category

Timothee Aubourg

2025-02-10

Table of contents

Preface

There are notes on Category Theory.

1 Understanding Category Theory, Type Theory, and Logic

Welcome to the fascinating world of **category theory**, **type theory**, and **logic**. These foundational concepts form the backbone of modern mathematics and computer science, playing critical roles in everything from programming languages to mathematical proofs.

We'll start with something simple: **bytes**. At the most basic level, everything a computer does boils down to manipulating bytes—small units of digital information. However, when we zoom out, we begin to see that these raw bytes can be interpreted and organized into meaningful patterns using the concepts of **types**. We'll introduce you to type theory and how it evolved in response to paradoxes in logic. We'll also look at how **category theory** provides a unifying language for mathematics and computation. By the end of this lesson, you'll have a clear understanding of how these powerful ideas intersect.

1.1 The Low-Level Perspective

At the lowest level of computing, **everything is just bytes**—binary sequences of 0s and 1s. These bytes represent data—whether numbers, text, images, or sound—everything you interact with on your computer is ultimately made up of bytes.

For example, consider the byte `01001000`. In **binary** encoding, this byte represents the character `'H'` in the ASCII character set. Similarly, `01100101` represents `'e'`, and `01101100` represents `'l'`. A sequence of these bytes might represent an entire message, such as the word **"Hello"**.

Bytes are the **raw material** of computing, but they don't mean much on their own unless we impose some structure. That's where **types** come in. **Type theory** provides a way to give structure to these raw bytes. In a type system, we classify data into categories or types. This makes it easier to reason about and manipulate the data. For instance, we could classify `01001000` as a character or `01100101` as an integer, depending on the context.

1.2 Type Theory in Mathematics

Although type theory was originally developed to improve computing, its roots lie in **mathematics**. The history of type theory can be traced back to **logical paradoxes** in set theory. Let's explore one of the most famous of these paradoxes: **Russell's Paradox**.

1.2.1 Russell's Paradox: A Deeper Look

In 1901, Bertrand Russell discovered a paradox in **set theory**, which is the branch of mathematics that deals with collections of objects. The paradox arises when you try to define the set of all sets that do not contain themselves.

Let's break it down:

- Consider the set (R), defined as: "the set of all sets that do not contain themselves."

Now, ask yourself: **Does the set (R) contain itself?**

1. If (R) **contains itself**, then by its definition, it must **not** contain itself, because it only contains sets that do not contain themselves.
2. If (R) **does not contain itself**, then by its definition, it **must** contain itself because it is a set that does not contain itself.

This creates a **contradiction**.

To solve this, Russell proposed introducing a **type system** to prevent sets from containing themselves. In this system, we can categorize sets into levels or **types**, so that a set cannot be a member of itself, thus avoiding the paradox.

1.2.2 The Solution via Type Theory

Russell's type system was the **precursor to type theory** as we know it today. He proposed a hierarchy of sets—called **types**—to organize sets and avoid paradoxes like the one above.

1.2.2.1 Example: Types in Set Theory

- **Type 0**: Sets that do not contain themselves.
- **Type 1**: Sets that contain only sets from **Type 0**.
- **Type 2**: Sets that contain only sets from **Type 1**, and so on.

With this hierarchical structure, a set of **Type 0** cannot contain itself because it can only contain sets from **Type 1**, which are on a higher level. By categorizing sets in this way, Russell's system ensured that paradoxes like the one mentioned above would not occur.

This idea of **types** would go on to influence the development of **Martin-Löf Type Theory (MLTT)**, a more sophisticated type system that is widely used in both **mathematics** and **computer science**.

1.3 Martin-Löf Type Theory (MLTT)

1.3.1 Key Ideas in MLTT

1.3.1.1 Types as Sets

In MLTT, types are collections of objects, similar to sets in set theory. For example, the type `Nat` (natural numbers) contains objects like `0, 1, 2`, etc.

1.3.1.1.1 Example:

Let's define the type `Nat`: - `Nat = {0, 1, 2, 3, ...}`.

This means the elements of the type `Nat` are natural numbers.

1.3.1.2 Types as Propositions

In MLTT, types can also represent logical propositions. A type `A → B` represents the proposition "if `A` is true, then `B` is true."

1.3.1.2.1 Example:

Let `A` be the type of natural numbers greater than 0, and `B` be the type of even numbers: - `A = {1, 2, 3, 4, ...}`. - `B = {2, 4, 6, 8, ...}`.

The type `A → B` means "if a number is greater than 0, then it is an even number."

1.3.1.3 Proofs as Objects

In MLTT, a proof of a proposition is itself a term of a specific type. So, a proof of `A → B` is a function that takes an element of type `A` and returns an element of type `B`.

1.3.1.3.1 Example:

Let's define a function `f: A → B` that proves "if a number is greater than 0, then it is even." We could define `f` as follows: - `f(1) = 2`, `f(2) = 4`, `f(3) = 6`, etc.

Here, `f` is a function that constructs even numbers, proving the implication `A → B`.

1.3.1.4 Constructive Proofs

MLTT insists on constructive proofs. This means if we prove the existence of an object, we must provide a concrete example.

1.3.1.4.1 Example:

To prove "there exists an even number greater than 0," we must construct such a number: - `f: Nat` where `f` could return `2`, which is the smallest even number greater than 0.

1.3.2 Example in MLTT (Proving Implication)

Let's prove the implication `A → B`, where `A` and `B` are types (propositions).

1.3.2.1 Steps:

1. **Assume a function `f: A → B` exists.**
   This means that for every element of type `A`, the function `f` produces an element of type `B`.
2. **Prove `A → B`**
   To prove the implication, we must show that for any element `a` of type `A`, we can derive an element `b` of type `B`.
3. **Use `f` to transform elements of `A` into elements of `B`.**
   Since `f` is of type `A → B`, it provides us with a proof. For example, if `a` is an element of type `A`, applying `f(a)` gives us an element of type `B`.

1.3.2.2 Example:

- Let `A` be the type of natural numbers greater than 0.
- Let `B` be the type of even numbers.
- We assume `f: A → B` such that `f(a)` returns an even number for every `a > 0`.

The function `f` can be a function that returns the next even number greater than `a`.

1.4 The Curry-Howard-Lambek Isomorphism

1.4.1 The Correspondence

1.4.1.1 Logic Type Theory

- In logic, a proof is a sequence of steps that proves a statement.
- In type theory, a type represents a proposition, and a term (a function) is a proof of that type.

1.4.1.1.1 Example:

- Logical statement: `(A → B)` means "if `A` is true, then `B` is true."
- Type theory: `(A → B)` is a type, and a function that takes an element of type `A` and returns an element of type `B` is a proof of that type.

1.4.1.2 Type Theory Category Theory

- In category theory, objects are types, and morphisms (arrows) are functions between types.
- In type theory, a term (proof) is a morphism that transforms one object (type) into another.

1.4.1.2.1 Example:

- In category theory, a morphism from object `A` to object `B` represents a function from type `A` to type `B`.

1.4.2 Step-by-Step Example of the Curry-Howard-Lambek Isomorphism

Let's walk through how a logical statement, a type in type theory, and a morphism in category theory are related.

1.4.2.1 Logical Statement:

- Consider the logical statement `(A → B)`, which means "if `A` is true, then `B` is true."

1.4.2.2 In Type Theory:

- The logical statement `A → B` corresponds to the type `A → B` in type theory.
- A function of type `A → B` is a proof of this implication. It takes an element of type `A` and returns an element of type `B`.

1.4.2.3 In Category Theory:

- In category theory, the objects are types (`A`, `B`).
- The morphisms (arrows) between objects represent functions. So, a morphism from object `A` to object `B` represents a function that transforms elements of type `A` into elements of type `B`.

Thus, the Curry-Howard-Lambek Isomorphism tells us that: - The logical statement `(A → B)` is represented by a type `A → B`. - A proof of this logical statement corresponds to a term (function) of type `A → B`. - In category theory, this is captured as a morphism from object `A` to object `B`.

1.4.2.4 Concrete Example:

- Let `A = {1, 2, 3, ...}` and `B = {2, 4, 6, 8, ...}`.
- A function `f: A → B` that proves "if a number is natural, then it is even" could map each element `a` from `A` to the next even number `b` in `B`:
  - `f(1) = 2, f(2) = 4, f(3) = 6, ....`

In category theory, the function `f` is a morphism between objects `A` and `B`. In type theory, `f` is a term of type `A → B`. In logic, `f` is a proof of the statement "if `A`, then `B`."

1.5 The Nature of Mathematics: Discovery or Invention?

Mathematicians often debate whether mathematics is an **invention** of the human mind or a **discovery** of universal truths. Unlike physicists, who conduct experiments to uncover the natural laws of the universe, mathematicians engage in abstract reasoning and logical deduction. Despite this, different branches of mathematics often uncover **equivalent structures**, suggesting that there may be some underlying **objective reality** to these concepts.

1.6 The Limits of Human Cognition

Humans evolved primarily for survival—identifying threats, securing food, and navigating social relationships. Our brains are excellent at dealing with concrete, immediate problems, but **abstract reasoning** is a more recent development. To handle complexity, we often use **decomposition**—breaking complex problems into simpler, manageable parts.

This principle, found in everything from **science** to **programming**, is central to how we understand and solve problems.

1.7 Is the Universe Fundamentally Composable?

In physics, scientists have long adhered to a **reductionist** approach—breaking matter down into smaller and smaller components. For instance, atoms are composed of protons, neutrons, and electrons, which are themselves composed of quarks. However, recent developments in physics challenge this reductionist view:

- **Quantum mechanics** shows that particles don't behave like simple building blocks.
- **String theory** suggests that the fundamental particles are not points, but tiny vibrating strings.

These discoveries raise a profound question: is the universe inherently **composable**, or is this just a human cognitive strategy?

1.8 Category Theory as a Study of Human Thought

Category theory, often seen as a highly abstract branch of mathematics, may not be as concerned with the intrinsic nature of the universe as we might think. Instead, it provides a framework for understanding how **humans reason** about complexity. It describes patterns and structures that our minds impose on the problems we encounter.

In this sense, category theory might be more about **epistemology**—how we understand the world—than **ontology**—what the world actually is.

Part I

Fundamentals

2 Associativity and its Role in Categories

2.1 Is it Possible to Have Non-Associative Theories?

In category theory, associativity is an important property. When combining elements in a category, you can associate them in different ways, but as long as associativity holds, the result will be the same. This property makes working with categories manageable and consistent.

But what if we didn't require strict associativity? Is it still possible to have a valid theory? The answer is yes. There are areas in mathematics where associativity is **weakened**: the two different ways of combining things aren't identical, but they are **isomorphic**. This means they are related in a way that one can be transformed into the other.

2.1.1 Example: Floating Point Arithmetic

For example, in floating point arithmetic, the order of operations may result in slightly different results due to the finite precision of calculations. This does not form a valid category since strict associativity is required for categories.

2.2 Categories vs. Groups

The structure described in category theory shares some similarities with **groups**, but it is more general. A **group** is a type of **monoid** with an added property: every element has an inverse.

If every morphism (arrow) in a category has an inverse, the category becomes a **groupoid**. A **groupoid** is more general than a group because a group only has one object with its operations, whereas a category can have multiple objects with morphisms between them.

The distinction between groups and categories is that in a category, not all objects can be composed with each other. The ability to compose objects depends on whether the end of one morphism matches the start of another.

3 Programming and Category Theory

3.1 The Category of Types

A practical example of category theory comes from programming languages such as **Haskell** and **ML**. In these languages, types can be treated as **objects**, and **functions** between types are the **morphisms** of the category.

In these programming categories, a function from type `A` to type `B` is considered an arrow or morphism from object `A` to object `B`.

3.1.1 Special Case in Haskell

In **Haskell**, things are slightly more complicated due to the concept of **laziness**. In Haskell, every type contains an undefined value (`bottom`) which represents non-terminating computations. This introduces a level of complexity that is not typically considered in pure category theory, as categories don't account for time or infinite loops in computation.

3.2 Types as Sets

In some programming languages like **ML**, types can be viewed as **sets** of values. The functions between these types are simply **functions between sets**. This simplification works well in languages that don't have infinite loops or non-terminating computations, unlike Haskell.

Thus, we can model a category of types as a **category of sets**. In this model, functions are the morphisms, and types are sets of values. A function from type `A` to type `B` becomes a mathematical function between sets `A` and `B`.

4 Building a Category from Sets

4.1 Set Theory and Categories

To build a category, you start with **sets**. In set theory, each set contains elements, and functions between sets are simply mappings from elements of one set to another. From this perspective, you can form a category where:

- The **objects** are the sets.
- The **morphisms** are the functions between the sets.

4.1.1 Forgetting Internal Structure

Once the category is created, the internal structure of the sets is forgotten. All that remains are the **morphisms** that relate the sets to each other. The **composition** of these morphisms follows the rules of category theory, with **associativity** and **identity** being central properties.

4.1.2 Composition of Functions

Function composition in set theory is simply the act of applying one function to the result of another. If `f: A -> B` and `g: B -> C`, then the composition `g  f` is a function that takes `A` to `C`.

- If `x` is an element of set `A`, `f(x)` is an element of set `B`.
- Then `g(f(x))` is an element of set `C`.

4.1.3 Identity Function

Each set also has an **identity function**. This is a function that maps each element of a set to itself. The identity function acts as a neutral element in function composition: when composed with any other function, it leaves that function unchanged.

- In a category, the identity function corresponds to an **identity morphism**.

- For any morphism `f`, composing it with the identity morphism gives back the original morphism: `f  id_A = f` and `id_B  f = f`.

4.2 The Category of Sets (Set Theory)

This category, called **Set**, consists of sets and functions between them. The creation of the category **Set** follows a process where the internal structure of the sets and their elements is forgotten. We only care about the **morphisms** (functions) and their composition.

4.2.1 Forgetting the Structure

Once you form a category, you forget about the internal structure of the objects. The objects (sets) become "atoms" with no internal properties, and you focus solely on how they are connected to other objects through morphisms (functions).

- In the category **Set**, sets are objects, and the morphisms are functions between them.
- You build a **multiplication table** (composition table) that defines how these morphisms can be composed, and this composition follows the axioms of category theory (associativity, identity).

4.2.2 Composition Table

In the category of sets, you can think of each function between two sets as an arrow. The composition of these arrows is simply the composition of the functions. The identity function for each set ensures that the composition works according to the rules of category theory.

4.3 Abstraction and Data Hiding

Category theory represents a higher level of abstraction. Once you form a category from sets, you forget about the internal elements of the sets. You are left with the **interface** of each object — how it connects to other objects through morphisms.

This abstraction represents the **end of the road for data hiding**: you don't need to know what the internal structure of a set is, only how it connects with other sets. This is similar to the idea of **data hiding** in programming, where you only expose the relevant interface of a data structure.

4.3.1 The Ultimate in Abstraction

Category theory provides a powerful tool for studying mathematical structures at a high level of abstraction. It allows us to work with objects and morphisms without worrying about the specifics of their internal construction. It is, in a sense, the ultimate language for abstraction.

5 Conclusion

Category theory offers a framework for thinking about mathematical structures and their relationships in an abstract, high-level way. By focusing on morphisms and their composition, we can study the properties of objects without needing to understand their internal details. This approach to abstraction leads to new insights into mathematics and computation, ultimately enabling a deeper understanding of how various structures are interconnected.

6 Relations and Functions

In mathematics and computer science, functions and relations play a crucial role in structuring information and defining transformations between sets. This lesson explores the concepts of relations, functions, and their properties, including invertibility and isomorphism.

7 Relations

7.1 What is a Relation?

A **relation** is a set of ordered pairs where elements from one set are associated with elements from another set.

For example, if we have two sets:

- $A = \{1, 2, 3\}$
- $B = \{a, b, c\}$

A relation between them could be:

$$R = \{(1, a), (2, b), (3, c)\}$$

7.2 Relations and Cartesian Products

A relation can be understood through the **Cartesian product** of two sets. The Cartesian product $A \times B$ consists of all possible pairs:

$$A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$$

A relation is simply a subset of this Cartesian product.

8 Functions

8.1 What is a Function?

A **function** is a special type of relation that has a well-defined **directionality**: each input from the **domain** is mapped to exactly one output in the **codomain**.

8.2 Key Properties of Functions

- A function **must** assign exactly **one** output to each input.
- The **domain** is the set of all possible inputs.
- The **codomain** is the set of possible outputs (though not necessarily all elements in the codomain are used).
- The actual values a function maps to are called the **image** of the function.

For example, if we define a function $f : A \to B$ as:

$$f(x) = x^2, \quad A = \{1, 2, 3\}, \quad B = \{1, 4, 9, 10\}$$

Then:

$$f(1) = 1, \quad f(2) = 4, \quad f(3) = 9$$

8.3 Total Functions vs. Partial Functions

- A **total function** maps **every** element of the domain to an element in the codomain.
- A **partial function** may leave some elements of the domain unmapped.

9 Invertibility of Functions

9.1 When is a Function Invertible?

A function is **invertible** if there exists another function that can reverse its effect. That is, given an output, we can determine the unique input that produced it.

Example: If $f(x) = x + 2$, the inverse function is $g(x) = x - 2$, because applying g after f returns the original input:

$$g(f(x)) = (x + 2) - 2 = x$$

Not all functions are invertible! A function must be **bijective** (both injective and surjective) to have an inverse.

10 Isomorphism: A Special Kind of Invertibility

10.1 What is an Isomorphism?

An **isomorphism** is a function that has a perfect inverse—it maps one set to another in a way that preserves structure.

Given two functions:

- $f : A \to B$
- $g : B \to A$

f and g are isomorphic if:

$$g \circ f = id_A \quad \text{and} \quad f \circ g = id_B$$

where id_A and id_B are identity functions that return their input unchanged.

10.2 Why are Functions Non-Isomorphic?

There are two main reasons why a function might not be an isomorphism:

10.2.1 1. Loss of Information: Non-Injective Functions

A function is **not injective** if multiple inputs map to the same output (collapsing elements).

Example:

$$f(x) = x^2$$

Here, both $f(2) = 4$ and $f(-2) = 4$, so the function is not invertible because we lose information about whether the input was positive or negative.

- **Opposite Property:** A function is **injective** (one-to-one) if no two inputs map to the same output.
- **Related concept: Monomorphism (monic functions).**

10.2.2 2. Incomplete Coverage: Non-Surjective Functions

A function is **not surjective** if it does not cover the entire codomain.

Example: Suppose $f : \mathbb{R} \to \mathbb{R}$ is defined by:

$$f(x) = e^x$$

Since the function never produces negative numbers, it is not surjective onto $\mathbb{R}$.

- **Opposite Property:** A function is **surjective** (onto) if it covers the entire codomain.
- **Related concept: Epimorphism (epic functions).**

11 Bijections and Isomorphism

A function is:

- **Injective** if it preserves uniqueness (one-to-one).
- **Surjective** if it covers the entire codomain (onto).
- **Bijective** if it is both injective and surjective, meaning it has a perfect inverse.

11.1 Why is a Bijective Function an Isomorphism?

A **bijective function** is an **isomorphism** because it:

1. Preserves uniqueness (no collapsing).
2. Fully maps the codomain (no gaps).
3. Has an inverse function that undoes its effect.

Example:
The function $f(x) = x + 3$ is bijective because:

- It is injective (each x gives a unique output).
- It is surjective (covers all real numbers).
- It has an inverse $g(x) = x - 3$, making it an isomorphism.

12 Summary and Key Takeaways

Property	Definition	Opposite Property	Related Concept
Relation	A set of ordered pairs	–	Cartesian Product
Function	A relation where each input maps to exactly one output	–	–
Injective	No two inputs map to the same output	Non-injective (collapses elements)	Monomorphism
Surjective	Covers the entire codomain	Non-surjective (leaves gaps)	Epimorphism
Bijective	Both injective and surjective	Non-bijective	Isomorphism
Isomorphism	A function with a perfect inverse	Non-isomorphic	–

13 Conclusion

Understanding functions and relations helps in many fields, from mathematics to computer science. Knowing when a function is invertible, injective, or surjective allows us to determine how information is preserved, mapped, and structured.

By mastering these concepts, you can better analyze transformations, abstractions, and models in various domains.

References