Intro to Lean 4 by Formalizing Category Theory

Xingyu Ren[1,*]

1 Department of Physics, The Chinese University of Hong Kong, Hong Kong SAR, China
* xy.ren@link.cuhk.edu.hk

September 25, 2025

Abstract

This pedagogical note introduces Lean 4 as a coding language for mathematicians, with a gradual, context-driven approach centered on the formalization of category theory following the first three chapters of Emily Riehl's *Category Theory in Context*. We assume no prior Lean experience. Every mathematical definition, theorem, and example is presented rigorously in LaTeX with corresponding Lean 4 code blocks, accompanied by detailed commentary linking syntax, type theory, and proof tactics to mathematical practice. The document is self-contained, providing a structured, compilation-ready resource for mathematicians seeking both exposure to Lean 4 and hands-on experience with formal mathematical proof in the context of category theory.

Contents

1 Introduction

First of all, we list the official Lean 4 website [1] and community [2] here. In the Lean 4 community, you can almost find everything you need: installation, documentation (you can search for the source code in Mathlib), online Lean 4 editor (without installation), tutorial, and so on.

1.1 Formal mathematics

Formal mathematics has moved from a niche pursuit to a central pillar of modern mathematics and AI-assisted research. Lean 4 sits at the forefront of this movement, offering an expressive, dependently-typed language and a performant implementation that is widely used for formalization, proof engineering, and integration with machine learning workflows. The Lean community and its library, mathlib4, provide a collaborative ecosystem for formalizing domains including category theory and for building datasets used to train mathematical large language models (LLMs).

1.2 Lean 4's foundations: the computational trilogy

Lean 4's foundations are most naturally described using the "computational trilogy" viewpoint: the same core phenomena of computation and reasoning are seen equivalently from three complementary perspectives—type theory (logic), programming languages (computation), and category/topos theory (mathematical semantics). Presenting Lean 4 through this rosetta-stone helps explain why dependent types, constructive logic, and the lambda calculus are not isolated features but three faces of a single design goal: a system in which programs, proofs, and categorical semantics continuously inform one another.

- **Type-theoretic (logic) view.** Propositions are types and proofs are terms. Dependent type theory lets types depend on terms, giving fine-grained expressiveness (e.g., `Vector A n`). Universe hierarchies prevent paradoxes while permitting statements about objects at different levels of abstraction.
- **Programming-language (computation) view.** Programs are terms (lambda-terms); computation (beta/eta reduction) gives operational content to proofs. Constructive logic enforces that existence claims carry explicit witnesses, so proofs can be executed or extracted as algorithms.
- **Categorical (semantics) view.** Types and terms correspond to objects and generalized elements in suitable categories or $(\infty,1)$-toposes; dependent types correspond to objects in slice categories and dependent product/sum correspond to adjoints (pullback, pushforward). This semantics provides an organizing language for interpreting type-theoretic constructs and for transporting intuition between logic and geometry/topology.

Viewing Lean 4 through the trilogy yields a compact account of its core building blocks and why they were chosen:

- **Dependent type theory** supplies the syntax and formation rules for expressing rich mathematical structures and parametric families of objects (the logic face). In the categorical view these correspond to objects and display maps in slices; in the programming view they describe typed interfaces and specifications of programs.
- **Constructive logic** (intuitionistic/propositions-as-types) ensures proofs have computational content, so the same terms serve both as verified constructions and as executable artifacts. From the categorical side, constructive reasoning aligns with internal logics of toposes and with the existence of universal constructions that have computational interpretations.
- **Lambda calculus and computation** provide the operational semantics (evaluation, normalization) used by the kernel and elaborator. This is the programming-face realization of proofs-as-programs; categorically it corresponds to morphisms and composition that implement substitution and function application.

The computational trilogy extends naturally to the dependent homotopy-type theory (HoTT) setting: the categorical perspective is upgraded from ordinary categories to $(\infty,1)$-categories (locally cartesian closed $(\infty,1)$-toposes), and equalities are interpreted as paths (homotopies). In that enriched viewpoint:

- Dependent types become homotopy types; identity types are path spaces.
- Categorical semantics move to slice $(\infty,1)$-categories and higher categorical notions that interpret higher inductive types and univalence.

- The programming view must account for higher-dimensional equalities and correspondingly richer operational notions (e.g., cubical or parametric computation models).

Lean 4's mainstream kernel is engineered for a particular set of tradeoffs: a small, performant core supporting dependent types, constructive reasoning, and efficient computation. Those design choices make Lean 4 excellent for large-scale formalization, proof engineering, and extraction of computational content, while also allowing controlled extensions and experiments toward HoTT-style features via metaprogramming, emulation of higher inductives, or alternative kernels. In short, the trilogy explains both the strengths of Lean 4's current design and the exact sense in which a "full HoTT mode" would move the system into a different point in the same conceptual space: it upgrades the categorical face (from 1-categorical semantics to ∞-categorical semantics) and requires corresponding changes in how equality and computation are treated in the programming and logic faces.

Here is the takeaway. Understanding Lean 4 as an instance of the computational trilogy clarifies: types, programs, and categorical semantics are not separate engineering choices but mutually constraining perspectives. Practical formalization work in Lean 4 benefits from keeping all three faces in view: type constructs guide programs, programs realize proofs, and categorical semantics provide the conceptual maps for transporting ideas across domains and for reasoning about extensions such as HoTT-style

1.3 mathlib4

mathlib4 is the community-driven Lean 4 mathematical library. It organizes definitions and theorems across algebra, analysis, topology, and category theory, emphasizing reuse via universe polymorphism and highly-general statements. The library is the primary source of formal content used by researchers, students, and AI systems working with Lean. Practical notes:

- Category theory in mathlib4 aims for maximal generality; newcomers should expect a learning curve but will gain powerful reusable abstractions.
- The library continually expands; documentation and search tools (LeanSearch, LeanExplore) are essential for navigation.
- Gaps persist (some undergraduate-level results, informal alignment), which motivates community contributions and targeted dataset creation.

1.4 AI for Math

Lean 4 changes how mathematics can be authored and taught:

- proofs become living, machine-checked artifacts open to refactor, verification, and reuse;
- provenance and reproducibility improve as formal results include machine-verifiable dependencies;
- pedagogical tools (Natural Number Game, tutorials, interactive exercises) help students learn proofs and computation simultaneously.

Lean formalizations (mathlib4) are now used to build datasets for mathematical LLMs and to evaluate proof-generating systems. These efforts have produced benchmarks and pipelines that map between natural language and formal Lean statements, and that train models to assist with autoformalization, proof generation, and proof engineering.

1.5 Outlook and challenges

Lean 4 and mathlib4 offer a laboratory for the future of mathematical research and AI collaboration. Key opportunities include AI assisted formalization at scale, conjecture generation, automated proof engineering, and improved semantic search in large formal libraries. Challenges include scaling formal content to cover broad mathematics, aligning LLM proof output with human standards, and extending kernels or tools to accommodate alternative foundational systems (like HoTT).

2 Getting Started

2.1 Installing Lean 4 and a Productive Development Environment

The canonical way to use Lean 4 is via a modern code editor (Visual Studio Code, VS Code) and Lean's official extension, which provides real-time feedback and interactive proof development. The essential steps are:

1. **Install VS Code**: Download and install from `https://code.visualstudio.com/`.
2. **Install Lean 4 extension:** Launch VS Code, search for "Lean4" in the extensions panel, and install it. This triggers additional guided installation (using Lean's *elan* version manager) for first-time users.
3. **Initialize a Lean project:** In the VS Code terminal, run `lake init <projectname>` to create a new Lean 4 project. You may also use `lake init <projectname> math` to include Mathlib, Lean's mathematical library.

Testing your setup: Open a file `Test.lean` and evaluate:

```
#eval Lean.versionString
```

This should output the installed Lean version in the VSCode "InfoView". We recommend to use Linux instead of Windows system, you can install a Window Subsystem for Linux (WSL) if you are using Windows.

Here is a nice Chinese Lean learning community [3], where you can try the online Lean editor for a first contact [4].

For more details and platform-specific troubleshooting, consult the official documentation [1].

2.2 Integrating Lean 4 Code into LaTeX Documents

To highlight Lean code in LaTeX (as we do below), we recommend the `minted` package, which uses `pygments` for syntax highlighting. For full color support and best compatibility:

- Install `pygments` for your Python installation.
- Compile your LaTeX source using `xelatex` or `lualatex` with the `--shell-escape` flag.
- The code block is the following form

```
        -- Lean 4 code here
```

For inline code, adjusting parameters for readability.

This note is formatted to support source code listings via minted.

3 Lean 4 Fundamentals

3.1 Basic Syntax and Expressions

Lean 4's syntax is reminiscent of other functional programming languages, but is governed by a rigorous type system, designed for both programming and formal proof. Below we introduce some core concepts.

3.1.1 Literals, Variables, and Comments

A *type* in Lean restricts what values a variable or function can take. Example types: `Nat` (natural numbers), `String`, `Prop` (propositions).

```lean
-- This is a comment
/- This is a multi-line comment. Useful for longer notes. -/

def x : Nat := 5 -- Defines variable x with value 5 and type Nat

#check x          -- Prints 'x : Nat' in the infoview

def mySum (a b : Nat) : Nat := a + b
#eval mySum 2 3  -- Evaluates to 5
```

Lean's **`#check`** inspects types, **`#eval`** evaluates expressions, and comments (`--` or `/- -/`) document code.

3.1.2 Functions and Lambda Syntax

Functions may be defined by name or anonymously:

```lean
def add (a b : Nat) : Nat := a + b
#check add                 -- add : Nat → Nat → Nat

#check fun (a b : Nat) ⇒ a + b  -- anonymous function, same type
#check λ (a b : Nat) ⇒ a + b    -- same, using Unicode λ (\lambda)
```

Curried functions and right-associative arrows (`A → B → C`) play a central role, especially in category theory formalization.

3.2 Structures and Records: Mathematical Abstraction

A **structure** in Lean is analogous to a record or struct in other programming languages: it bundles together several named fields, each with specified types, into a single object. Mathematically, you can think of a structure as a tuple with named components, together with convenient syntax for construction, projection, and update. Every structure in Lean has the following ingredients:

1. **Name**: the identifier of the structure (e.g. `Point`).
2. **Fields**: a list of named components, each with a type (e.g. `x : Float`, `y : Float`).

3. **Constructors**: Lean automatically generates a constructor function that builds a structure from its fields.

4. **Projections**: Lean automatically generates projection functions to access each field (e.g. `Point.x`, `Point.y`).

5. **Namespace**: Lean creates a namespace with the same name as the structure, so you can refer to its fields unambiguously.

A 2D point example:

```
structure Point where
  x : Float
  y : Float
deriving Repr
```

This defines a structure `Point` with two fields:

- `x : Float`
- `y : Float`

We can construct a `Point` using record syntax:

```
def origin : Point := { x := 0.0, y := 0.0 }
```

We can access fields using dot notation:

```
#eval origin.y   -- yields 0.0
```

Lean supports record update syntax, which copies a structure while changing some fields:

```
def translateX (p : Point) (dx : Float) : Point :=
  { p with x := p.x + dx }

#eval translateX origin 5.0
-- { x := 5.0, y := 0.0 }
```

Here `{ p with x := ... }` means: copy all fields from `p`, but replace the field `x` with a new value.

Lean allows certain typeclasses to be *derived* automatically for structures. The command

```
deriving Repr
```

tells Lean to generate an instance of the `Repr` typeclass, which controls how values are printed ("represented") when evaluated. Without this, `#eval` would not know how to display a `Point`. With `deriving Repr`, Lean can pretty-print points:

```
#eval origin
-- prints "⟨Point.mk 0.0 0.0⟩" or "{ x := 0.0, y := 0.0 }"
```

Other common derivable classes include `BEq` (boolean equality), `DecidableEq` (decidable equality), and `Inhabited` (default element).

Thus, a structure in Lean is a powerful way to package data:

1. It has *fields* with types.

2. Lean automatically provides *constructors* and *projections*.

3. You can use *record syntax* to build and update values.

4. You can *derive* useful behaviors like printing or equality.

This mechanism underlies almost all mathematical definitions in Lean: groups, rings, categories, functors, and so on are all defined as structures bundling together data and axioms. And this usage is foundational for defining categorical objects, functors, cones, and more, faithfully mirroring abstract mathematical data collections.

3.3 Typeclasses: Overloading, Algebraic Abstraction

A **typeclass** in Lean is a mechanism for bundling together operations and axioms associated with a type, and for automatically retrieving canonical instances of those operations when needed. Mathematically, typeclasses correspond to *structures with implicit arguments*, such as groups, rings, or topological spaces. Every typeclass in Lean has several key ingredients:

1. **Name**: the identifier of the class (e.g. `MyAdd`).

2. **Parameters**: the type(s) the class is about (e.g. `A : Type`).

3. **Fields**: the operations or properties required (e.g. an addition function).

4. **Instances**: concrete implementations of the class for specific types (e.g. natural numbers with their usual addition).

5. **Instance search**: Lean's mechanism for automatically finding the right instance when a function requires it.

Typeclasses allow us to:

1. Write generic definitions that work for any type with the required structure.

2. Reuse algebraic theory: once we prove a theorem about all groups, it applies to every instance of `Group`.

3. Encode categories "with structure" (e.g. topological groups, ordered rings) by layering typeclasses.

We define a typeclass `MyAdd` that provides an addition operation on a type.

```
-- Define a typeclass with one operation: addition
class MyAdd (A : Type) where
  myadd : A → A → A
```

This says: for any type `A`, if `A` has an instance of `MyAdd`, then we know how to add two elements of `A`. We can now declare that the natural numbers form an instance of `MyAdd`, using their usual addition:

```
-- Provide an instance for natural numbers
instance : MyAdd Nat where
  myadd := Nat.add
```

Here `Nat.add` is the built-in addition on natural numbers. Now we can write generic functions that require only the existence of a `MyAdd` instance:

```
-- A generic function that doubles an element
def double {A : Type} [MyAdd A] (x : A) : A :=
  MyAdd.myadd x x

#eval double 7   -- works because Nat has a MyAdd instance
```

More explanation on this code:

- The curly braces `{A : Type}` mean that the type argument is implicit.
- The square brackets `[MyAdd A]` mean: "assume Lean can find an instance of `MyAdd A`." This triggers *instance search*.
- When we call `double 7`, Lean automatically inserts `A = Nat` and finds the `MyAdd Nat` instance we defined.

Finaly, we make some comments on usage:

- Typeclasses are Lean's way of encoding algebraic hierarchies. For example, `AddMonoid`, `Group`, `Ring`, and `Field` are all typeclasses in `mathlib4`.
- You can layer typeclasses: a `Group` extends `Monoid`, which extends `Semigroup`.

Thus, a typeclass in Lean has:

1. A *declaration* bundling operations and axioms.
2. *Instances* providing concrete implementations.
3. *Automatic instance search* that allows generic code to be reused across many types.

This is one of Lean's most powerful features, enabling us to write mathematics once and apply it to all relevant structures. This mechanism is essential for category theory: categories, functors, and natural transformations are all defined as typeclasses or structures with typeclass fields.

3.4 Universes: Types of Types

To avoid paradoxes such as Russell's paradox, Lean organizes types into a hierarchy of *universe levels*. Each universe is a "type of types" at a certain level. For example:

- `Type 0` (often written simply as `Type`) is the universe of small types, such as `Nat`.
- `Type 1` is the universe containing `Type 0` itself.
- `Type 2` contains `Type 1`, and so on.

Thus we have an infinite ascending hierarchy

$$\texttt{Type 0} : \texttt{Type 1} : \texttt{Type 2} : \cdots \tag{1}$$

This prevents self-reference paradoxes like "the type of all types that do not contain themselves. In Lean we declare universe variables with the keyword `universe`:

```
universe u v
```

This introduces two universe levels `u` and `v`, which can be used as parameters in type definitions.

Suppose we want to define the type of functions from a type `A` in universe `u` to a type `B` in universe `v`. The resulting function type must live in the larger of the two universes, written `max u v`:

```
def Fun (A : Type u) (B : Type v) : Type (max u v) :=
  A → B
```

Explanation:

- `A : Type u` means A is a type living in universe u.
- `B : Type v` means B is a type living in universe v.
- The function type `A → B` lives in universe `max u v`, since it must be large enough to contain both A and B.

In category theory we often distinguish between:

- *Small categories*: categories whose objects and morphisms form sets in some universe.
- *Large categories*: categories whose objects form a proper class, but still live in a higher universe.

For example: The category **Set** of small sets lives in some universe `u`. The category of all small categories lives in a higher universe `v`.

In Lean, this distinction is tracked explicitly by universe levels. This ensures that we cannot naively form "the category of all categories" without specifying which universe of categories we mean. We can define the category of types in a given universe:

```
universe u

instance : Category (Type u) where
  Hom X Y := X → Y
  id := fun x ⇒ x
  comp := fun f g x ⇒ g (f x)
```

Here:

- `Type u` is the collection of all types in universe `u`.
- This gives us the category **Type_u**, the category of u-small types.

In summary:

1. Lean has an infinite hierarchy of universes: `Type 0`, `Type 1`, `Type 2`, ...
2. Universe variables (`u`, `v`, etc.) let us write definitions that are polymorphic in size.
3. Function types and categories automatically live in the maximum of the universes of their components.

This machinery is essential for formalizing category theory safely, distinguishing between small and large categories.

4 Tactics and Formal Proofs

A distinctive feature of Lean and other interactive theorem provers is the separation between *declarative* and *tactic* proof styles:

- In **declarative style**, one writes a proof term directly, much like writing down a λ-term or explicit construction.
- In **tactic style**, one writes a sequence of proof steps, each transforming the current goal into simpler subgoals, until all are solved.

For mathematicians, tactic proofs echo the stepwise reasoning familiar from blackboard arguments: "assume this," "apply that lemma," "rewrite by this equality," etc.

Here are two basic examples.

```
theorem sum_comm (a b : Nat) : a + b = b + a := by
  apply Nat.add_comm    -- Applies the commutativity of addition

theorem and_comm (p q : Prop) : p ∧ q → q ∧ p := by
  intro h              -- Assume p ∧ q
  constructor          -- To prove q ∧ p, prove each part
  . exact h.right      -- First goal: prove q
  . exact h.left       -- Second goal: prove p
```

We now describe the most common tactics, with intuitive explanations and examples.

intro / intros Introduce hypotheses or variables into the context. Mathematically: "Suppose x is arbitrary," or "Assume p holds."

```
theorem imp_self (p : Prop) : p → p := by
  intro h         -- Assume p
  exact h         -- Goal is p, which is exactly h
```

exact Supply a term that directly solves the goal. Mathematically: "This is exactly what we need."

```
theorem true_intro : True := by
  exact True.intro
```

apply Use a theorem or lemma to reduce the goal to its premises. Mathematically: "To prove Q, it suffices to prove the hypotheses of lemma L."

```
theorem add_comm (a b : Nat) : a + b = b + a := by
  apply Nat.add_comm   -- Lean knows this lemma already
```

rw (rewrite) Rewrite the goal using a known equality or equivalence. Mathematically: "Replace the left-hand side by the right-hand side."

```
theorem add_zero (n : Nat) : n + 0 = n := by
  rw [Nat.add_zero]     -- Rewrites using the lemma Nat.add_zero
```

constructor When the goal is a conjunction ($p \wedge q$) or an inductive type with multiple constructors, this tactic splits the goal into subgoals. Mathematically: "To prove $p \wedge q$, prove p and q separately."

```
theorem and_comm (p q : Prop) (h : p ∧ q) : q ∧ p := by
  constructor
  . exact h.right
  . exact h.left
```

cases Perform case analysis on a hypothesis of inductive type (e.g. disjunctions, natural numbers, lists). Mathematically: "Consider cases: if p holds, or if q holds."

```
theorem or_comm (p q : Prop) : p ∨ q → q ∨ p := by
  intro h
  cases h with
  | inl hp ⇒ exact Or.inr hp   -- Case: p holds
  | inr hq ⇒ exact Or.inl hq   -- Case: q holds
```

induction Prove a statement by induction on a natural number or other inductive type. Mathematically: "Base case, then inductive step."

```
theorem add_zero_right (n : Nat) : n + 0 = n := by
  induction n with
  | zero ⇒ rfl
  | succ k ih ⇒ simp [Nat.add_succ, ih]
```

rfl Closes a goal if both sides are definitionally equal. Mathematically: "This is true by definition."

```
theorem refl_example (n : Nat) : n = n := by
  rfl
```

simp Simplifies the goal using a collection of rewrite lemmas tagged with `simp`. Mathematically: "Simplify using known identities."

```
theorem zero_add (n : Nat) : 0 + n = n := by
  simp
```

have Introduce an intermediate lemma or calculation inside a proof. Mathematically: "Let us first observe that..."

```
theorem double_even (n : Nat) : ∃ m, 2 * n = 2 * m := by
  use n
  have h : 2 * n = 2 * n := rfl
  exact h
```

let Introduce a local definition. Mathematically: "Define x to be ..."

```
theorem example_let (a b : Nat) : (fun x ⇒ x + b) a = a + b := by
  let f := fun x ⇒ x + b
  rfl
```

Summary of common tactics:

1. `intro` / `intros`: assume hypotheses.
2. `exact`: provide the required term.
3. `apply`: reduce goal using a lemma.
4. `rw`: rewrite using equalities.
5. `constructor`: split conjunctions or structured goals.
6. `cases`: case analysis on disjunctions or inductive types.

7. `induction`: induction on natural numbers or inductive types.
8. `rfl`: solve goals by definitional equality.
9. `simp`: simplify using rewrite lemmas.
10. `have`, `let`: introduce intermediate results or definitions.

These tactics correspond closely to moves in standard mathematical proofs:

1. `intro` ↔ “Suppose …”
2. `apply` ↔ “It suffices to prove …”
3. `rw` ↔ “Replace by …”
4. `constructor` ↔ “Prove each part separately.”
5. `cases` ↔ “Consider cases.”
6. `induction` ↔ “Proceed by induction.”

Once mathematical structures are defined and relevant lemmas imported, proofs in Lean can be as concise and natural as their blackboard counterparts.

5 Formalizing Category Theory

Below we start to formalize basic category theory following the textbook [5]. In this chapter, we will go through the chapter 1 of this book.

5.1 Defining Categories

Definition 5.1 (Category Riehl 1.1.1)**.** *A **category** $\mathcal{C}$ consists of:*

- *A collection of objects denoted $X, Y, Z, \ldots$.*
- *For each pair X, Y, a collection of morphisms* $\mathrm{Hom}(X, Y)$.
- *A composition law: For $f \in \mathrm{Hom}(X, Y)$, $g \in \mathrm{Hom}(Y, Z)$, a morphism $g \circ f \in \mathrm{Hom}(X, Z)$.*
- *For each object X, an identity morphism $\mathrm{id}_X \in \mathrm{Hom}(X, X)$.*
- ***Axioms:***
  1. Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$.
  2. Unit: *For all* $f : X \to Y$, $\mathrm{id}_Y \circ f = f$, $f \circ \mathrm{id}_X = f$.

The corresponding Lean code is following.

```
universe u v

structure CategoryStruct where
  Obj   : Type u
  Hom   : Obj → Obj → Type v
  id    : ∀ X, Hom X X
  comp  : ∀ {X Y Z}, Hom X Y → Hom Y Z → Hom X Z
```

```
id_comp'   : ∀ {X Y} (f : Hom X Y), comp f (id Y) = f
comp_id'   : ∀ {X Y} (f : Hom X Y), comp (id X) f = f
assoc'     : ∀ {W X Y Z} (f : Hom W X) (g : Hom X Y) (h : Hom Y Z),
                comp f (comp g h) = comp (comp f g) h
```

This structure directly encodes the "data" of a category:

- `Obj` is a type of objects;
- `Hom : Obj → Obj → Type v` gives the hom set/type between any two objects. (Universes u, v keep track of "size.")
- `id` and `comp` handle composition and identities.
- The last three fields (`id_comp'`, `comp_id'`, `assoc'`) are axioms, mirror the mathematical requirements.

Lean idiom: The use of `structure` not only introduces a new type but also automatically provides "namespace" scoping for projections (`C.Hom`, etc), and supports dot notation (C.id).

5.2 Examples: Sets as a Category

Example 5.2 (Category of Sets)**.** *The objects are sets, morphisms are functions, composition is function composition, and identities are identity functions.*

```
instance CategoryOfSets : CategoryStruct where
  Obj := Type u
  Hom := fun X Y ⇒ X → Y
  id := fun X ⇒ id
  comp := fun {X Y Z} (f : X → Y) (g : Y → Z) ⇒ g ⍰ f
  id_comp' := fun f ⇒ rfl
  comp_id' := fun f ⇒ rfl
  assoc' := fun f g h ⇒ rfl
```

Here, `rfl` is a *tactic* certifying that reflexivity holds (equality is by definitional computation), which is immediate for standard function composition.

5.3 Opposite Category

Definition 5.3 (Opposite Category Riehl 1.2.1)**.** *Given a category $\mathcal{C}$, the* ***opposite category*** *$\mathcal{C}^{op}$ has the same objects, with morphisms reversed:* $\mathrm{Hom}_{\mathcal{C}^{op}}(X, Y) = \mathrm{Hom}_{\mathcal{C}}(Y, X)$. *Composition and identities are defined accordingly.*

```
def Opposite (C : CategoryStruct) : CategoryStruct where
  Obj := C.Obj
  Hom := fun X Y ⇒ C.Hom Y X
  id := C.id
  comp := fun {X Y Z} (f : C.Hom Y X) (g : C.Hom Z Y) ⇒ C.comp f g

  id_comp' := C.id_comp'
  comp_id' := C.comp_id'
  assoc'   := C.assoc'
```

Pedagogical note: This concrete encoding shows how general categorical constructions translate into type-theoretic ones; for more complex structures, keeping "explicit" universes and argument types makes definitions easier to relate to familiar mathematics.

5.4 Isomorphisms in a Category

Definition 5.4 (Isomorphism Riehl 1.1.5)**.** *A morphism $f : X \to Y$ in a category $\mathcal{C}$ is an* isomorphism *if there exists $g : Y \to X$ such that $g \circ f = \mathrm{id}_X$ and $f \circ g = \mathrm{id}_Y$.*

```
structure is_iso {C : CategoryStruct} {X Y : C.Obj} (f : C.Hom X Y) where
  inv : C.Hom Y X
  left_inv  : C.comp f inv = C.id X
  right_inv : C.comp inv f = C.id Y
```

This naturally mirrors the mathematical definition and can be used to endow categories with the structure and properties required for more advanced results.

5.5 Functors

Definition 5.5 (Functor Riehl 1.3.1)**.** *A (covariant)* ***functor*** *$F : \mathcal{C} \to \mathcal{D}$ between categories consists of:*

- *an assignment of objects $F(X)$ in $\mathcal{D}$ to each X in $\mathcal{C}$,*
- *an assignment of morphisms $F(f) : F(X) \to F(Y)$ to each $f : X \to Y$ in $\mathcal{C}$,*
- *such that $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$,*
- *and $F(g \circ f) = F(g) \circ F(f)$ for composable f, g.*

```
structure FunctorStruct (C D : CategoryStruct) where
  objMap : C.Obj → D.Obj
  morMap : {X Y : C.Obj} → C.Hom X Y → D.Hom (objMap X) (objMap Y)
  preserves_id : ∀ (X : C.Obj), morMap (C.id X) = D.id (objMap X)
  preserves_comp : ∀ {X Y Z : C.Obj} (f : C.Hom X Y) (g : C.Hom Y Z),
    morMap (C.comp f g) = D.comp (morMap f) (morMap g)
```

The functions `objMap` and `morMap` explicitly package the two roles "on objects" and "on morphisms", enforcing functoriality via the preservation laws.

5.6 Natural Transformations

Definition 5.6 (Natural Transformation Riehl 1.4.1)**.** *Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a* ***natural transformation*** *$\alpha : F \Rightarrow G$ is a collection of morphisms $\alpha_X : F(X) \to G(X)$ (one for each $X \in \mathcal{C}$) such that for every $f : X \to Y$ in $\mathcal{C}$, the following diagram commutes:*

$$\begin{tikzcd} F(X) \arrow[r, "F(f)"] \arrow[d, "\alpha_X"] & F(Y) \arrow[d, "\alpha_Y"] \\ G(X) \arrow[r, "G(f)"] & G(Y) \end{tikzcd}$$

```
structure NatTrans {C D : CategoryStruct}
  (F G : FunctorStruct C D) where
  app : (X : C.Obj) → D.Hom (F.objMap X) (G.objMap X)
  naturality : ∀ {X Y : C.Obj} (f : C.Hom X Y),
    D.comp (app X) (G.morMap f) = D.comp (F.morMap f) (app Y)
```

By indexing `app X` over the objects and enforcing `naturality`, we encode "diagrams commute" directly.

6 Representability and the Yoneda Lemma

In this chapter, we formalize the ideas from Chapter 2 of [5], focusing on representable functors and the Yoneda lemma. These concepts are central to categorical thinking and are elegantly expressible in Lean.

6.1 Representable Functors

Definition 6.1 (Representable Functor Riehl 2.1.1)**.** *Let $\mathcal{C}$ be a locally small category and $F : \mathcal{C}^{op} \to \mathbf{Set}$ a functor. We say F is* representable *if there exists an object $c \in \mathcal{C}$ and a natural isomorphism*

$$F \cong \mathrm{Hom}_{\mathcal{C}}(-, c). \tag{2}$$

That is, F is naturally isomorphic to the hom-functor $\mathrm{Hom}(-, c)$*.*

Note: Riehl uses contravariant functors for representability. In Lean, we encode this using the opposite category.

```
-- First, the Hom-functor itself
def HomFunctor (C : CategoryStruct) (c : C.Obj) :
    FunctorStruct (Opposite C) CategoryOfSets :=
{ objMap := λ X ⇒ C.Hom X c,
  morMap := λ {X Y} f g ⇒ C.comp g f,
  preserves_id := by intros; rfl,
  preserves_comp := by intros; rfl }

-- A natural isomorphism is a natural transformation with an inverse
structure NatIso {C D : CategoryStruct}
  (F G : FunctorStruct C D) where
  toNat    : NatTrans F G
  invNat   : NatTrans G F
  leftInv : ∀ X, D.comp (toNat.app X) (invNat.app X) = D.id (F.objMap X)
  rightInv: ∀ X, D.comp (invNat.app X) (toNat.app X) = D.id (G.objMap X)

-- Now, representability packages the data:
structure Representable {C : CategoryStruct}
  (F : FunctorStruct (Opposite C) CategoryOfSets) where
  obj : C.Obj
  iso : NatIso F (HomFunctor C obj)
```

Explanation:

- `HomFunctor C c` is the canonical hom-functor $\mathrm{Hom}(-, c)$.
- `NatIso F G` encodes a natural isomorphism between functors F and G.
- `Representable F` asserts that F is representable by providing:
  1. an object `obj : C.Obj`,
  2. a natural isomorphism `iso : NatIso F (HomFunctor C obj)`.

The Lean definition now matches the textbook: a representable functor is not just the Hom-functor, but a functor *together with* an isomorphism to a Hom-functor.

6.2 Yoneda Lemma

Theorem 6.2 (Yoneda Lemma Riehl 2.2.4). *Let $\mathcal{C}$ be a locally small category, $F : \mathcal{C}^{op} \to$ **Set** a functor, and $c \in \mathcal{C}$. Then there is a natural bijection:*

$$\mathrm{NatTrans}(\mathrm{Hom}(-, c), F) \cong F(c)$$

We now formalize this in Lean. First, we define the forward direction of the bijection.

```
def yonedaToFun {C : CategoryStruct}
  (F : FunctorStruct (Opposite C) CategoryOfSets)
  (c : C.Obj)
  (α : NatTrans (HomFunctor C c) F) : F.objMap c :=
  α.app c (C.id c)
```

Explanation:

- Given a natural transformation α, we evaluate it at c and apply it to id_c.
- This yields an element of $F(c)$.

Next, we define the inverse direction.

```
def yonedaInvFun {C : CategoryStruct}
  (F : FunctorStruct (Opposite C) CategoryOfSets)
  (c : C.Obj)
  (x : F.objMap c) : NatTrans (HomFunctor C c) F :=
{
  app := λ X f ⇒ F.morMap f x,
  naturality := by
    intros X Y f; dsimp;
    rw [F.preserves_comp]; rfl
}
```

Explanation:

- Given $x \in F(c)$, we define a transformation $\alpha_X(f) := F(f)(x)$.
- Naturality follows from functoriality of F.

Finally, we prove the bijection.

```
theorem yoneda {C : CategoryStruct}
  (F : FunctorStruct (Opposite C) CategoryOfSets)
  (c : C.Obj) :
  (NatTrans (HomFunctor C c) F) ≃ F.objMap c :=
{
  toFun := yonedaToFun F c,
  invFun := yonedaInvFun F c,
  left_inv := by
    intro α; ext X f;
    dsimp [yonedaToFun, yonedaInvFun];
    rw [←F.preserves_comp]; rfl,
  right_inv := by
    intro x; dsimp [yonedaToFun, yonedaInvFun];
    rfl
}
```

Lean idiom: The use of `≃` denotes a bijection (equivalence of types). The `ext` tactic allows us to prove equality of structures by componentwise reasoning.

6.3 Summary

We have:

- Defined representable functors as those naturally isomorphic to $\mathrm{Hom}(-, c)$.
- Constructed the hom-functor in Lean using the opposite category.
- Proved the Yoneda lemma by constructing a bijection between natural transformations and elements of $F(c)$.

This completes our formalization of Chapter 2 of [5]. The Yoneda lemma is a cornerstone of category theory and its Lean formalization showcases the power of type-theoretic reasoning.

7 Formalizing Chapter 3 — Limits, Colimits, and Universal Constructions

7.1 Limits and Universal Properties

Definition 7.1 (Limit of a Diagram). *Given a diagram $F : J \to \mathcal{C}$, a* ***limit*** *is an object $L \in \mathcal{C}$ with a cone $\lambda : L \to F$, such that for any cone $\mu : C \to F$, there is a unique morphism $u : C \to L$ making the diagram commute.*

```
structure Cone {J C : CategoryStruct} (F : FunctorStruct J C) where
  vertex : C.Obj
  edge   : ∀ (j : J.Obj), C.Hom vertex (F.objMap j)
  comm   : ∀ {j k : J.Obj} (f : J.Hom j k),
    C.comp (edge j) (F.morMap f) = edge k
```

Lean note: This code defines a cone as a vertex and a family of legs (edges), requiring the triangle identities (commutativity) for each morphism $f : j \to k$ in the diagram.

Definition 7.2 (Limit). *A* limit *of F is a cone (L, λ) such that for any other cone (C, μ), there is a unique $u : C \to L$ making all the diagrams commute.*

```
structure IsLimit {J C : CategoryStruct} {F : FunctorStruct J C}
  (c : Cone F) where
  lift : ∀ (s : Cone F), C.Hom s.vertex c.vertex
  fac : ∀ (s : Cone F) (j : J.Obj),
           C.comp (lift s) (c.edge j) = s.edge j
  uniq : ∀ (s : Cone F) (m : C.Hom s.vertex c.vertex),
           (∀ j, C.comp m (c.edge j) = s.edge j) → m = lift s
```

Discussion: This explicitly encodes the universal property—the lift and its uniqueness—crucial to all limit-like definitions in category theory.

7.2 Colimits: Dual Notion

By reversing arrows, one defines cocones and colimits in an analogous fashion.

—

8 Pedagogical Strategies: How to Read and Write Lean as a Mathematician

8.1 Linking Mathematical Notation and Lean Syntax

Correspondence Table: Mathematical vs Lean 4

Mathematical Notion	LaTeX	Lean 4
Object in a category	$X \in \mathrm{Ob}(\mathcal{C})$	`X : C.Obj`
Morphism	$f : X \to Y$	`f : C.Hom X Y`
Composition	$g \circ f$	`C.comp f g`
Identity	id_X	`C.id X`
Functor on objects	$F(X)$	`F.objMap X`
Functor on morphisms	$F(f)$	`F.morMap f`

By internalizing these translations, mathematicians can confidently read and write proofs using Lean.

8.2 Best Practices

- **Write explicit types** to avoid type ambiguity.
- **Use whitespace and comments** to clarify definitions, especially for complex constructions.
- **Use Lean's InfoView** and `check`, `print` to inspect structures.
- **Develop small examples first**, e.g., categories of finite sets, to validate intuition.

—

9 Advanced Topics and Further Reading

9.1 Universes and Category Theory in Lean

Lean's universe system replaces set-theoretic size issues; "small" and "locally small" categories can be parameterized by universe variables (`u`, `v`), and creating "the category of all small categories" requires careful formulation to avoid paradoxes. Consult the Lean reference and mathematics-in-Lean project for real-world workarounds and idioms.

9.2 Mathlib and Category Theory Libraries

Though this note restricts itself to core Lean 4, the broader community mathematical library Mathlib contains advanced formalizations of categories, functors, limits, and much more, with extensive use of typeclasses. However, as discussed in pedagogical reviews, Mathlib's high level of abstraction can be challenging for newcomers. Beginners are encouraged to first work through minimal and concrete Lean examples.

9.3 LaTeX or Markdown Output for Lean Code Documentation

Beyond minted, documentation generation tools and literate programming modes (such as Lean's markdown integration and Jupyter-based environments) can help bridge the gap between readable mathematics and formal code.

—

10 Conclusion

This note has guided the reader through the installation, syntax, and core structures of Lean 4, always in the context of formalizing the basic elements and results of category theory, as developed in Riehl's text. By systematically pairing each mathematical definition and theorem with its Lean implementation—and by detailing the connecting logic for each step—mathematicians can both deepen their understanding of modern proof assistants and gain practical skills in writing and verifying formal mathematics.

The approach advocated here does not replace traditional learning from books and peer mathematicians, but rather offers a complementary, interactive mode of exploration and verification that is rapidly gaining importance in the mathematical community. As Lean and its libraries continue to grow, the tools and strategies introduced in this primer should remain directly relevant.

Acknowledgements

Author contributions: A.N.A. designed, implemented, and wrote all sections of the manuscript.

Funding information No external funding was used for this pedagogical project.

A Lean 4 Syntax Cheat Sheet

A compact reference of Lean 4 syntax, core constructs, and tactics can be found in [**?**, **?**].

B Glossary of Terms

- **Category** – Collection of objects and morphisms with composition and identities
- **Functor** – Structure-preserving mapping between categories
- **Natural Transformation** – Morphism between functors
- **Typeclass** – Lean mechanism for polymorphism over algebraic structures
- **Universe** – Level in Lean's hierarchy of types
- **Tactic** – Step or command for constructing proofs interactively

References

[1] *Lean Programming Language*, `https://lean-lang.org/`, [Accessed 24-09-2025].

[2] *Lean community*, `https://leanprover-community.github.io/`, [Accessed 25-09-2025].

[3] *leanprover.cn*, `https://www.leanprover.cn/#lean-zh`, [Accessed 24-09-2025].

[4] *GitHub - Lean-zh/GlimpseOfLean*, `https://github.com/Lean-zh/GlimpseOfLean`, [Accessed 24-09-2025].

[5] E. Riehl, *Category Theory in Context*, Courier Dover Publications, ISBN 9780486809038 (2016).