Contents

1 Introduction

This serves as a brief starting point for understanding how mathematical proofs can be formalized in Lean, as well as being an introduction to the language itself. Lean is both a **functional programming language** and a **theorem prover**. We'll focus primarily on its role as a theorem prover. But what does this mean, and how can that be achieved?

A programming language defines a **set of rules, semantics, and syntax** for writing programs. To achieve a goal, a programmer must write a program that meets given specifications. There are two primary approaches: **program derivation** and **program verification** ([NPS90] Section 1.1). In **program verification**, the programmer first writes a program and then proves it meets the specifications. This approach checks for errors at **run-time** when the code executes. In **program derivation**, the programmer writes a proof that a program with certain properties exists, then extracts a program from that proof. This approach enables specification checking at **compilation-time**, catching errors while typing, thus, before execution. This distinction corresponds to **dynamic** versus **static type systems**. Most programming languages combine both approaches; providing basic types for annotation and compile-time checking, while leaving the remaining checks to be performed at runtime.

Example 1.1 *In a dynamically typed language, like JavaScript, variables can change type after they are created. For example, a variable defined as a number can later be reassigned to a string: TypeScript is a statically typed superset of JavaScript. Unlike JavaScript, it performs type checking at compile time. This means we can prevent the previous behavior, while writing our code, simply by adding a type annotation: Now converting a variable with type annotation number to a string results into a compile time error.*

Nonetheless, TypeScript, even though it has a sophisticated type system, cannot fully capture complex mathematical properties. As well as for the most programming languages, program specifications can only be enforced at runtime. Lean, by contrast, uses a much more powerful type system that enables it to express and verify mathematical statements with complete rigor fully during compilation time. This makes it particularly suitable as a **theorem prover** for formalizing mathematics.

Lean's type system is based on **dependent type theory**, specifically the **Calculus of Inductive Constructions** (CIC) with various extensions. It's important to note that **type theory** is not a single, unified theory, but rather a family of related theories with various extensions, ongoing developments, and rich historical ramifications.

Type theory emerged as an alternative foundation for mathematics, addressing paradoxes that arose in naive set theory. Consider Russell's famous paradox: let $S = \{x \mid x \notin x\}$ be the set of all sets that do not contain themselves. This construction is paradoxical, leading to the contradiction $S \in S \iff S \notin S$. Type theory resolves such issues by working with **types** as primary objects rather than sets, and by restricting which constructions are well-formed.

Dependent type theory, the framework underlying Lean, extends basic type systems by allowing types to depend on values. For instance, one can define the type of vectors of length n, where n is itself a value. This capability makes dependent type theory particularly expressive for formalizing mathematics.

Various proof assistants have been developed based on different variants of type theory, including Agda, Coq, Idris, and Lean. Each system makes different design choices regarding which rules and features to include. Lean adopts the **Calculus of Inductive Constructions**, which extends the Calculus of Constructions, introduced in Coq, with **inductive types**. Inductive types allow for the definition of structures such as natural numbers, lists, and trees.

A fundamental design feature of Lean is its **universe hierarchy** of types, with **Prop** (the proposition type) as a distinguished universe at the bottom. The **Prop** universe exhibits two special properties: **impredicativity** and **proof irrelevance**. Proof irrelevance means that all proofs of the same proposition are considered equivalent. What matters is whether a proposition can be proven, not which specific proof is given. This separation between propositions (**Prop**) and data types (**Type**) was first introduced in **N.G. de Bruijn's AUTOMATH system** (1967) ([Tho99]).

For our purposes, we do not need to delve deeply into the theoretical foundations; instead, we will introduce the relevant concepts as needed while working with Lean. The practical aspects of writing proofs in Lean will be our primary focus, and the theoretical machinery will be explained only insofar as it aids understanding of how to formalize mathematics effectively.

1.1 Lean first steps

In the language of type theory, and by extension in Lean, we write `x : X` to mean that x is a **term** of type X. For example, `2 : ℕ` annotates 2 as a natural

number, or more precisely, as a term of the natural number type. Lean has internally defined types such as `Nat` or $\mathbb{N}$ (you can type `\Nat` to get the Unicode symbol). The command `#check` allows us to inspect the type of any espression, term or variable.

Example 1.2

```
#check 2 -- 2 : Nat
#check 2 + 2 -- 2 + 2 : Nat
```

[Try this example in Lean Web Editor] By following the link, you can try out the code in your browser. Lean provides a dedicated ***infoview*** *panel on the right side. Position your cursor after* `#check 2`*, and the infoview will display the output* `2 : Nat`*. This dynamic interaction, where the infoview responds to your cursor position, is what makes Lean an* ***interactive theorem prover****. As you move through your code, the infoview continuously updates, showing computations, type information, and proof states at each location.*

At first glance, one might be tempted to view the colon notation as analogous to the membership symbol $\in$ from set theory, treating types as if they were sets. While this intuition can be helpful initially, type theory offers a fundamentally richer perspective. The crucial insight is the **Curry-Howard correspondence**, also known as the **propositions-as-types** principle. This correspondence establishes a deep connection between mathematical proofs and programs: **propositions correspond to types**, and **proofs correspond to terms** inhabiting those types. Under this interpretation, a term $x : X$ can be understood in two more ways:

- As a **computational object**: x is a program or data structure of type X
- As a **logical object**: x is a proof of the proposition X

Lean is a concrete realization of the propositions-as-types principle, proving a theorem, within the language, amounts to constructing a term of the appropriate type. When we write `theorem_name : Proposition`, we are declaring that `theorem_name` is a proof (term) of `Proposition` (type). For example, consider proving that $2 + 2 = 4$:

Example 1.3

```
theorem two_plus_two_eq_four : 2 + 2 = 4 := rfl
```

Lean's syntax is designed to resemble the language of mathematics. Here, we use the `theorem` *keyword to encapsulate our proof, of the statement/proposition* $2+2=4$*, giving it the name* `two_plus_two_eq_four`*. This allows us to reference and reuse this result later in our code. After the semicolon,* `:`*, we introduce the statement;* `2 + 2 = 4`*. The* `:=` *operator expects the proof term that establishes the theorem's validity. The proof itself consists of a single term:* `rfl` *(short for* ***reflexivity****). This is a proof term that works by* ***definitional equality****,*

Lean's kernel automatically reduces both sides of the equation to their normal (definitional) form and verifies they are identical. Since $2+2$ *computes to* 4*, the proof succeeds immediately. We can now use this theorem in subsequent proofs. For instance:*

```
example : 1 + 1 + 1 + 1 = 4 := two_plus_two_eq_four
```

Well, this example is simple enough for Lean to evaluate by itself: `1 + 1 + 1 + 1 = 2 + 2 = 4` *and conclude with* `two_plus_two_eq_four`*. Actually,* `rfl` *would solve the equation similarly, so this is just applying* `rfl` *again (it's a bit of cheating). Here, I used* `example`*, which is handy for defining anonymous expressions for demonstration purposes. Before diving into the discussion, here is another keyword,* `def`*, used to introduce definitions and functions.*

```
def addOne (n : Nat) : Nat := n + 1
```

This definition expects a natural number as its parameter, written `(n : Nat)` *and returns a natural number. [Run in browser]*

Let's now turn to how logic is handled in Lean and how the Curry-Howard isomorphism is reflected concretely.

2 Logic and Proposition as Types

Logic is the study of reasoning, branching into various systems. We refer to **classical logic** as the one that underpins much of traditional mathematics. It's the logic of the ancient Greeks (not fair) and truthtables, and it remains used nowadays for pedagogical reasons. We first introduce **propositional logic**, which is the simplest form of classical logic. Later we will extend this to **predicate (or first-order) logic**, which includes **predicates** and **quantifiers**. In this setting, a **proposition** is a statement that is either true or false, and a **proof** is a logical argument that establishes the truth of a proposition. Propositions can be combined with logical **connectives** such as "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$),"false" ($\bot$), ,"true" ($\top$) "implies" ($\Rightarrow$), and "if and only if" ($\Leftrightarrow$). These connectives allow the creation of complex or compound propositions.

Here how connectives are defined in Lean:

Example 2.1 (LogicaL connectives in Lean)

```
#check And (a b : Prop) : Prop
#check Or (a b : Prop) : Prop
#check True : Prop
#check False : Prop
#check Not (a : Prop) : Prop
#check Iff (a b : Prop) : Prop
```

`Prop` stands for proposition, and it is an essential component of Lean's type system. For now, we can think of it as a special type whose inhabitants are proofs; somewhat paradoxically, a type of types.

Logic is often formalized through a framework known as the **natural deduction system**, developed by Gentzen in the 1930s ([Wad15]). This approach brings logic closer to a computable, algorithmic system. It specifies rules for deriving **conclusions** from **premises** (assumptions from other propositions), called **inference rules**.

Example 2.2 (Deductive style rule) *Here is an hypothetical example of inference rule.*

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_n}{C}$$

Where the $P_1, P_2, \ldots, P_n$, above the line, are hypothetical premises and, the hypothetical conclusion C is below the line.

The inference rules needed are:

- **Introduction rules** specify how to form compound propositions from simpler ones, and
- **Elimination rules** specify how to use compound propositions to derive information about their components.

Type theory employs this porocedure too, by referring to deduction rules as judments, and each elemtns of a with terms ans types. here an examople:

Example 2.3 $$\frac{\Gamma, \qquad p_1 : P_1 \qquad p_2 : P_2 \qquad \cdots \qquad P_n}{C}$$

Notation 2.4 *We use $A \vdash B$ (called turnstile) to designate a deduction of B from A. It is employed in Gentzen's **sequent calculus** ([GTL89]) and moslty used in type theory. The square brackets around a premise $[A]$ mean that the premise A is meant to be **discharged** at the conclusion. The classical example is the introduction rule for the implication connective. To prove an implication $A \to B$, we assume A (shown as $[A]$), derive B under this assumption, and then discharge the assumption A to conclude that $A \to B$ holds without the assumption. The turnstile is predominantly used in judgments and type theory with the meaning of "entails that".*

Example 2.5 (Type Theory Judgments and Contexts) *In type theory,* ***judgments*** *are formal statements about the well-formedness of types and terms. There are typically four fundamental kinds of judgments []:*

1. $\Gamma \vdash A$ *type* – *"A is a well-formed type in context* Γ*"*
2. $\Gamma \vdash t : A$ – *"t is a term of type A in context* Γ*"*
3. $\Gamma \vdash A \equiv B$ *type* – *"types A and B are judgmentally equal in context* Γ*"*
4. $\Gamma \vdash t_1 \equiv t_2 : A$ – *"terms* t_1 *and* t_2 *are judgmentally equal of type A in context* Γ*"*

The ***context*** Γ *(Greek capital gamma) represents a finite list of type declarations for variables, formally written as* $\Gamma = x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ *(cite). The context encodes the* ***assumptions*** *or* ***hypotheses*** *under which a judgment is made.*

For example, consider these concrete judgments:

$$\vdash \mathbb{N} \text{ type} \quad \textit{(natural numbers form a type)} \tag{1}$$

$$x : \mathbb{N} \vdash x : \mathbb{N} \quad \textit{(variable } x \textit{ has type } \mathbb{N} \textit{ when declared)} \tag{2}$$

$$x : \mathbb{N}, y : \mathbb{N} \vdash x + y : \mathbb{N} \quad \textit{(addition of naturals yields a natural)} \tag{3}$$

$$\vdash \lambda x : \mathbb{N}.\ x + 1 : \mathbb{N} \to \mathbb{N} \quad \textit{(successor function)} \tag{4}$$

Let's look at how we can define some connectives. The inference rules needed are:

- **Introduction rules** specify how to form compound propositions from simpler ones, and
- **Elimination rules** specify how to use compound propositions to derive information about their components.

Conjunction ($\wedge$)

- Introduction

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-Intro}$$

- Elimination

$$\frac{A \wedge B}{A}\ \wedge\text{-Elim}_1 \qquad\qquad \frac{A \wedge B}{B}\ \wedge\text{-Elim}_2$$

Disjunction ($\vee$)

- Introduction

$$\frac{A}{A \vee B}\ \vee\text{-Intro}_1 \qquad\qquad \frac{B}{A \vee B}\ \vee\text{-Intro}_2$$

- Elimination (Proof by cases)

$$\frac{A \vee B \qquad [A] \vdash C \qquad [B] \vdash C}{C}\ \vee\text{-Elim}$$

Implication ($\rightarrow$)

- Introduction

$$\frac{[A] \vdash B}{A \rightarrow B} \rightarrow\text{-Intro}$$

- Elimination (Modus Ponens)

$$\frac{A \rightarrow B \qquad A}{B} \rightarrow\text{-Elim}$$

Let's look at how we can define some connectives.

Conjunction ($\wedge$)

- Introduction

$$\frac{A \qquad B}{A \wedge B} \wedge\text{-Intro}$$

- Elimination

$$\frac{A \wedge B}{A} \wedge\text{-Elim}_1 \qquad\qquad \frac{A \wedge B}{B} \wedge\text{-Elim}_2$$

Disjunction ($\vee$)

- Introduction

$$\frac{A}{A \vee B} \vee\text{-Intro}_1 \qquad\qquad \frac{B}{A \vee B} \vee\text{-Intro}_2$$

- Elimination (Proof by cases)

$$\frac{A \vee B \qquad [A] \vdash C \qquad [B] \vdash C}{C} \vee\text{-Elim}$$

Implication ($\rightarrow$)

- Introduction

$$\frac{[A] \vdash B}{A \rightarrow B} \rightarrow\text{-Intro}$$

- Elimination (Modus Ponens)

$$\frac{A \rightarrow B \qquad A}{B} \rightarrow\text{-Elim}$$

Notation 2.6 *We use $A \vdash B$ (called turnstile) to designate a deduction of B from A. It is employed in Gentzen's* ***sequent calculus*** *([GTL89]), whereas in natural deduction the corresponding symbol is*

$$\begin{array}{c} A \\ \vdots \\ B \end{array}$$

There are some minor differences, in fact, which I don't fully understand. The square brackets around a premise $[A]$ mean that the premise A is meant to be ***discharged*** *at the conclusion. The classical example is the introduction rule for the implication connective. To prove an implication $A \to B$, we assume A (shown as $[A]$), derive B under this assumption, and then discharge the assumption A to conclude that $A \to B$ holds without the assumption. The turnstile is predominantly used in judgments and type theory with the meaning of "entails that".*

Lean has its own syntax for connectives and their relative inference rules. For instance $A \wedge B$ can be presented as `And(A, B)` or `A \land B`, . Its untroduction rule is constructed by `And.intro _ _` or shortly `⟨_, _⟩` (underscore are placeholder for assumptions or "propositional functions"). The pair $A \wedge B$ can be then consumed using elimination rules `And.left` and `And.right`.

Example 2.7 *Let's look at our first Lean example*

```
example (H_A : A) (H_B : B) : (A \land B) := And.intro H_A H_B
```

Lean aims to resemble the language used in mathematics. For instance, when defining a function or expression, one can use keywords such as `theorem` or `def`. Here, I used `example`, which is handy for defining anonymous expressions for demonstration purposes. After that comes the statement to be proved:

```
(H_A : A) (H_B : B) : (A \land B)
```

Meaning given a proof of A and a proof of B we can form a proof of $(A \wedge B)$. The operator `:=` assigns a value (or return an expression) for the statement which "has to be a proof of it". `And.intro` is implemented as:

```
And.intro: p \to q \to (p \land q).
```

It says: if you give me a proof of p and a proof of q, then i return a proof of $p \wedge q$. We therefore conclude the proof by directly giving `And.intro H_A H_B`. Here another way of writing the same statment.

```
example (H_p : p) (H_B : B) : And(A, B) := ⟨H_p, H_B⟩
```

This system of inference rules allows us to construct proofs in an algorithmic and systematical way, organized in what is called a **proof tree**. To reduce complexity, we follow a **top-down** . This methodology forms the basis of **proof**

assistants like Lean, Coq, and Agda, which help verify the correctness of mathematical proofs by checking each step against these rules. We will see later that Lean, in fact, provides an info view of the proof tree which helps us understand and visualize the proof structure.

Let's examine a concrete example of a proof.

Example 2.8 (Associativity of Conjunction) *We prove that $(A \wedge B) \wedge C$ implies $A \wedge (B \wedge C)$. First, from the assumption $(A \wedge B) \wedge C$, we can derive A:*

$$\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B} \wedge E_1}{A} \wedge E_1$$

Second, we can derive $B \wedge C$:

$$\dfrac{\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B} \wedge E_1}{B} \wedge E_2 \qquad \dfrac{\dfrac{(A \wedge B) \wedge C}{C} \wedge E_2}{}}{B \wedge C} \wedge I$$

Finally, combining these derivations we obtain $A \wedge (B \wedge C)$:

$$\dfrac{(A \wedge B) \wedge C \vdash A \qquad (A \wedge B) \wedge C \vdash B \wedge C}{A \wedge (B \wedge C)} \wedge I$$

Example 2.9 (Lean Implementation) *Let us now implement the same proof in Lean.*

```
theorem and_associative (a b c : Prop) : (a \land b) \land c \to a \land
    (b \land c) :=
fun h : (a \land b) \land c =>
-- First, from the assumption (a ∧ b) ∧ c, we can derive a:
have hab : a \land b := h.left -- extracts (derive) a proof of (a ∧ b)
    from the assumption
have ha : a := hab.left -- extracts a from (a ∧ b)
-- Second, we can derive b ∧ c (here we only extract b and c and
    combine them in the next step)
have hc : c := h.right
have hb : b := hab.right
-- Finally, combining these derivations we obtain a ∧ (b ∧ c)
show a \land (b \land c) from ⟨ha, ⟨hb, hc⟩⟩
```

We introduce the `theorem` *with the name* `and_associative`*. The type signature* `(a ∧ b) ∧ c → a ∧ (b ∧ c)` *represents our logical implication. Here, we construct tehe proof term using a function with the* `fun` *keyword. Why a function? We have already encountered the Curry-Howard correspondence in Lean previously, though without explicitly stating it. According to this correspondence, a proof of an implication can be understood as a function that takes a hypothesis as input and produces the desired conclusion as output. We will revisit this concept in more detail later. The* `have` *keyword introduces local lemmas within our*

proof scope, allowing us to break down complex reasoning into manageable intermediate steps, mirroring our natural deduction proof from before. Just before the keyword `show`*, the info view displays the following context and goal:*

```
a b c : Prop
h : (a \land b) \land c
hab : a \land b
ha : a
hc : c
hb : b
\vdash a \land b \land c
```

Finally, the `show` *keyword explicitly states what we are proving and verifies that our provided term has the correct type. In this case,* `show a ∧ (b ∧ c) from ⟨ha, ⟨hb, hc⟩⟩` *asserts that we are constructing a proof of* `a ∧ (b ∧ c)` *using the term* `⟨ha, ⟨hb, hc⟩⟩`*. The* `show` *keyword serves two purposes: it makes the proof more readable by explicitly documenting what is being proved at this step, and it performs a type check to ensure the provided proof term matches the stated goal up to* ***definitional equality****. Two types are definitionally equal in Lean when they are identical after computation and unfolding of definitions—in other words, when Lean's type checker can mechanically verify they are the same without requiring additional proof steps. Here, the goal* `⊢ a ∧ b ∧ c` *is definitionally equal to* `a ∧ (b ∧ c)` *due to how conjunction associates, so* `show` *accepts this statement. If we had tried to use* `show` *with a type that was only* ***propositionally*** *equal (requiring a proof to establish equality) but not definitionally equal, Lean would reject it.*

2.1 Predicate logic and dependency

To capture more complex mathematical ideas, we extend our system from propositional logic to **predicate logic**. A **predicate** is a statement or proposition that depends on a variable. In propositional logic we represent a proposition simply by P. In predicate logic, this is generalized: a predicate is written as $P(a)$, where a is a variable. Notice that a predicate is just a function. This extension allows us to introduce **quantifiers**: $\forall$ ("for all") and $\exists$ ("there exists"). These quantifiers express that a given formula holds either for every object or for at least one object, respectively. In Lean if α is any type, we can represent a predicate `P` on α as an object of type $\alpha \to$ `Prop`. Thus given an `x : α` (an element with type α) `P(x) : Prop` would be representative of a proposition holding for `x`.

of a variable name can be substituted without changing the meaning of the predicate or statement. This should feel familiar from mathematics, where the meaning of an expression does not depend on the names we assign to variables. We can give an informal reading of the quantifiers as infinite logical operations:

$$\forall x.\, A(x) \equiv A(a) \land A(b) \land A(c) \land \dots$$
$$\exists x.\, A(x) \equiv A(a) \lor A(b) \lor A(c) \lor \dots$$

The expression $\forall x.\, P(x)$ can be understood as generalized form of implication. If P is any proposition, then $\forall x.\, P$ expresses that P holds regardless of the choice of x. When P is a predicate, depending on x, this captures the idea that we can derive P from any assumption about x.

Example 2.10 *Lean expresses quantifiers as follow.*

```
∀ (x : X), P x
forall (x : X), P x -- another notation
```

Listing 1: For All

```
∃ (x : X), P x
exist (x : X), P x -- another notation
```

Listing 2: Exists

Where `x` *is a varible with a type* `X`*, and* `P x` *is a proposition, or predicate, holding for* `x`*.*

Example 2.11 (Existential introduction in Lean) *When introducing an* ***existential*** *proof, we need a* ***pair*** *consisting of a witness and a proof that this witness satisfies the statement.*

```
example (x : Nat) (h : x > 0) : ∃ y, y < x :=
  Exists.intro 0 h -- or shortly ⟨0, h⟩
```

Notice that `⟨0, h⟩` is product type holding a data (⟨witness) and a proof of it. The **existential elimination rule** (`Exists.elim`) performs the opposite operation. It allows us to prove a proposition Q from $\exists x, P(x)$ by showing that Q follows from $P(w)$ for an **arbitrary** value w.

Example 2.12 (Existential elimination in Lean) *The existential rules can be interpreted as an infinite disjunction, so that existential elimination naturally corresponds to a* ***proof by cases*** *(with only one single case). In Lean, this reasoning is carried out using* ***pattern matching****, a known mechanism in functional programming for dealing with cases, with* `let` *or* `match`*, as well as by using* `cases` *or* `rcases` *construct.*

```
example (h : ∃ n : Nat, n > 0) : ∃ n : Nat, n > 0 :=
  match h with
  | ⟨witness, proof⟩ => ⟨witness, proof⟩
```

Example 2.13 *The* ***universal quantifier*** *may be regarded as a generalized function. Accordingly, In Lean, universal elimination is simply function application.*

```
example : ∀ n : Nat, n ≥ 0 :=
  fun n => Nat.zero_le n
```

3 Describing and use properties

Functions are primitive objects in type theory. For example, it is interesting to note that a relation can be expressed as a function: `R : α → α → Prop`. Similarly, when defining a predicate (`P : α → Prop`) we must first declare `α : Type` to be some arbitrary type. This is what is called **polymorphism**, more specifically **parametrical polymorphism**. A canonical example is the identity function, written as $\alpha \to \alpha$, where α is a type variable. It has the same type for both its domain and codomain, this means it can be applied to booleans (returning a boolean), numbers (returning a number), functions (returning a function), and so on. In the same spirit, we can define a transitivity property of a relation as follows:

```
def Transitive (α : Type) (R : α → α → Prop) : Prop :=
  ∀ x y z, R x y → R y z → R x z
```

To use `Transitive`, we must provide both the type α and the relation itself. For example, here is a proof of transitivity for the less-than relation on $\mathbb{N}$ (in Lean `Nat` or $\mathbb{N}$):

```
theorem le_trans_proof : Transitive Nat (· ≤ · : Nat → Nat → Prop) :=
  fun x y z h1 h2 => Nat.le_trans h1 h2 -- this lemma is provided by Lean
```

Looking at this code, we immediately notice that explicitly passing the type argument `Nat` is somewhat repetitive. Lean allows us to omit it by letting the type inference mechanism fill it in automatically. This is achieved by using **implicit arguments** with curly brackets:

```
def Transitive {α : Type} (R : α → α → Prop) : Prop :=
  ∀ x y z, R x y → R y z → R x z
theorem le_trans_proof : Transitive (· ≤ · : Nat → Nat → Prop) :=
  fun x y z h1 h2 => Nat.le_trans h1 h2
```

Lean's type inference system is quite powerful: in many cases, types can be completely inferred without explicit annotations. For instance, (NEED TO EXPLAIN TYPE INFERENECE). Let us now revisit the transitivity proof, but this time for the less-than-equal relation on the rational numbers (`Rat` or $\mathbb{Q}$) instead.

```
import Mathlib

theorem rat_le_trans : Transitive (· ≤ · :   Rat → Rat → Prop) :=
  fun _ _ _ h1 h2 => Rat.le_trans h1 h2
```

Here, `Rat` denotes the rational numbers in Lean, and `Rat.le_trans` is the transitivity lemma for $\leq$ on rational numbers, provided by Mathlib. We import Mathlib to access `Rat` and `le_trans`. Mathlib is the community-driven mathematical library for Lean, containing a large body of formalized mathematics and ongoing development. It is the defacto standard library for both programming and proving in Lean [Com20], we will dig into it as we go along. Notice that we used a function to discharge the universal quantifiers required by transitivity. The underscores indicate unnamed variables that we do not use later. If we had named them, say `x y z`, then: `h1` would be a proof of `x ≤ y`, `h2` would be a proof of `y ≤ z`, and `Rat.le_trans h1 h2` produces a proof of `x ≤ z`. The `Transitive` definition is imported from Mathlib and similarly defined as before.

Example 3.1 *The code can be made more readable using tactic mode. In this mode, you use tactics—commands provided by Lean or defined by users—to carry out proof steps succinctly, avoid code repetition, and automate common patterns. This often yields shorter, clearer proofs than writing the full term by hand.*

```
import Mathlib

theorem rat_le_trans : Transitive (· ≤ · : Rat → Rat → Prop) := by
intro x y z hxy hyz
exact Rat.le_trans hxy hyz
```

This proof performs the same steps but is much easier to read. Using `by` *we enter Lean's tactic mode, which (together with the info view) shows the current goal and context. Move your cursor just before* `by` *and observe how the info view changes. The goal is initially displayed as* $\vdash$ `Transitive fun x1 x2 ↦ x1 ≤ x2`*. The tactic* `intro` *is mainly used to introduce variables and hypotheses corresponding to universal quantifiers and assumptions into the context (essentially deconstructing universal quantifiers and implications). Now position your cursor just before* `exact` *and observe the info view again. The goal is now* $\vdash$ `x ≤ z`*, with the context showing the variables and hypotheses introduced by the previous tactic. The* `exact` *tactic closes the goal by supplying the term* `Rat.le_trans hxy hyz` *that exactly matches the goal (the specification of* `Transitive`*). You can hover over each tactic to see its definition and documentation.*

3.1 Exploring Mathlib (The Rat structure)

In these examples we cheated and have used predefined lemmas such as `Nat.le_trans` and `Rat.le_trans`, just to simplify the presentation. We can now dig into the implementation of these lemmas. Let's look at the source code of `Rat.le_trans`. The Mathlib 4 documentation website is at `https://leanprover-community.`

`github.io/mathlib4_docs`, and the documentation for `Rat.le_trans` is at `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Algebra/Order/Ring/Unbundled/Rat.html#Rat.le_trans`. Click the "source" link there to jump to the implementation in the Mathlib repository. In editors like VS Code you can also jump directly to the definition (Ctrl+click; Cmd+click on macOS). Another way to check source code is by using `#print Rat.le_trans`.

```
variable (a b c : Rat)
protected lemma le_trans (hab : a ≤ b) (hbc : b ≤ c) : a ≤ c := by
  rw [Rat.le_iff_sub_nonneg] at hab hbc
  have := Rat.add_nonneg hab hbc
  simp_rw [sub_eq_add_neg, add_left_comm (b + -a) c (-b), add_comm (b +
    -a) (-b), add_left_comm (-b) b (-a), add_comm (-b) (-a),
    add_neg_cancel_comm_assoc, ← sub_eq_add_neg] at this
  rwa [Rat.le_iff_sub_nonneg]
```

The proof uses several tactics and lemmas from Mathlib. The `rw` or `rewrite` tactic is very common and sintactically similar to the mathematical practice of rewriting an expression using an equality. In this case, with `at`, we use it to rewrite the hypotheses `hab` and `hbc` using the another Mathlib's lemma `Rat.le_iff_sub_nonneg`, which states that for any two rational numbers `x` and `y`, `x ≤ y` is equivalent to `0 ≤ y - x`. Thus we now have the hypotheses tranformerd to :

```
hab : 0 ≤ b - a
hbc : 0 ≤ c - b
```

The `have` tactic introduces an intermediate result. If you omit a name, Lean assigns it the default name `this`. In our situation, from `hab : a ≤ b` and `hbc : b ≤ c` we can derive that `b - a` and `c - b` are nonnegative, hence their sum is nonnegative:

```
this : 0 ≤ b - a + (c - b)
```

The most involved step uses `simp_rw` to simplify the expression via a sequence of other existing Mathlib's lemmas. The tactic `simp_rw` is a variant of `simp`: it performs rewriting using the simp set (and any lemmas you provide), applying the rules in order and in the given direction. Lemmas that `simp` can use are typically marked with the `@[simp]` attribute. This is particularly useful for simplifying algebraic expressions and equations. After these simplifications we obtain:

```
this : 0 ≤ c - a
```

Clearly, the proof relies mostly on `Rat.add_nonneg`. Its source code is fairly involved and uses advanced features that are beyond our current scope. Nevertheless, it highlights an important aspect of formal mathematics in Mathlib. Mathlib defines `Rat` as an instance of a linear ordered field, implemented via a normalized fraction representation: a pair of integers (numerator and denominator) with positive denominator and coprime numerator and denominator [Lea25b]. To achieve this, it uses a **structure**. In Lean, a structure is a dependent record (or product type) type used to group together related fields or properties as a single data type. Unlike ordinary records, the type of later fields may depend on the values of earlier ones. Defining a structure automatically introduces a constructor (usually mk) and projection functions that retrieve (deconstruct) the values of its fields. Structures may also include proofs expressing properties that the fields must satisfy.

```
structure Rat where
  /-- Constructs a rational number from components.
  We rename the constructor to ‘mk’‘ to avoid a clash with the smart
   constructor. -/
  mk' ::
  /-- The numerator of the rational number is an integer. -/
  num : Int
  /-- The denominator of the rational number is a natural number. -/
  den : Nat := 1
  /-- The denominator is nonzero. -/
  den_nz : den ≠ 0 := by decide
  /-- The numerator and denominator are coprime: it is in "reduced
  form". -/
  reduced : num.natAbs.Coprime den := by decide
```

In order to work with rational numbers in Mathlib, we use the `Rat.mk'` constructor to create a rational number from its numerator and denominator, if omitted the default would be `Rat.mk`. The fields `den_nz` and `reduced` are proofs that the denominator is nonzero and that the numerator and denominator are coprime, respectively. These proofs are automatically generated by Lean's `decide` tactic, which can solve certain decidable propositions (to be discussed in the next section).

Example 3.2 *Here is how we can define and manipulate rational numbers in Lean.*

```
def half : Rat := Rat.mk' 1 2
def third : Rat := Rat.mk' 1 3
```

When working with rational numbers, or more generally with structures, we must provide the required proofs as arguments to the constructor (or Lean must be able to ensure them). For instance `Rat.mk' 1 0` or `Rat.mk' 2 6` would be rejected. In the case of rationals, Mathlib unfolds the definition through `Rat.numDenCasesOn`. This principle states that, to prove a property of an arbitrary rational number, it suffices to consider numbers of the form `n /. d` in canonical (normalized) form, with `d > 0` and `gcd n d = 1`. This reduction allows mathlib to transform proofs about $\mathbb{Q}$ into proofs about $\mathbb{Z}$ and $\mathbb{N}$, and then lift the result back to rationals.

Example 3.3 *We present a simplified implementation of addition non-negativity for rationals (`Rat.add_nonneg`), maintaining a similar approach: projecting everything to the natural numbers and integers first. To illustrate the proof technique clearly, we avoid using existing lemmas from the Rat module in Mathlib. Mathlib is indeed organized into modules by mathematical domain (e.g., Nat, Int, Rat).*

We start by defining helper lemmas needed in the main proof. Given a natural number (which in this case represents the denominator of a rational number) that is not equal to zero, we prove it must be positive. This follows directly by applying the Mathlib lemma `Nat.pos_of_ne_zero`:

```
import Mathlib

lemma nat_ne_zero_pos (den : ℕ) (h_den_nz : den ≠ 0) : 0 < den :=
  Nat.pos_of_ne_zero h_den_nz
```

The naming convention follows Mathlib best practices aiming to be descriptive by indicating types and properties involved.

The following lemma is slightly more involved. It states that if a rational number (num / den) is non-negative, then its numerator must also be non-negative:

```
lemma rat_num_nonneg {num : ℤ} {den : ℕ} (hden_pos : 0 < den)
    (h : (0 : ℚ) ≤ num / den) : 0 ≤ num := by
  contrapose! h
  have hden_pos_to_rat : (0 : ℚ) < den := Nat.cast_pos.mpr hden_pos
  have hnum_neg_to_rat : num < (0 : ℚ) := Int.cast_lt.mpr h
  exact div_neg_of_neg_of_pos hnum_neg_to_rat hden_pos_to_rat
```

The lemma requires the denominator to be positive as well as the non-negativity of the rational number, expressed as `num / den` *where the types of* `num` *and* `den` *are inferred. First, notice the type annotation* `(0 : ℚ)`*. This explicit type annotation on zero forces the entire equation to be casted into rational numbers. Without this annotation, Lean would infer* `0` *as a natural number by default. However, since the main theorem we are proving concerns rational numbers, we must ensure all comparisons occur in* ℚ*. The tactic* `contrapose!` *does what you might expect: it proves a statement by contraposition. According to the documentation:*

- `contrapose` *turns a goal* `P → Q` *into* `¬ Q → ¬ P`
- `contrapose!` *turns a goal* `P → Q` *into* `¬ Q → ¬ P` *and pushes negations inside* `P` *and* `Q` *using* `push_neg`
- `contrapose h` *first reverts the local assumption* `h`*, then uses* `contrapose` *and* `intro h`
- `contrapose! h` *first reverts the local assumption* `h`*, then uses* `contrapose!` *and* `intro h`

In our case, `contrapose! h` *transforms the goal from proving* `0 ≤ num` *to assuming* `num < 0` *and proving* `num / den < 0`*. We then introduce two local hypotheses. The first,* `hden_pos_to_rat`*, proves that the denominator is positive when cast to rationals, using* `Nat.cast_pos`*. The suffix* `.mpr` *selects the "modus ponens reverse" direction of the biconditional (the* ← *direction of the* ↔*). Next, we introduce* `hnum_neg_to_rat`*, which expresses that the numerator is negative when cast to rationals, using* `Int.cast_lt` *with* `.mpr` *again. Finally, we apply* `div_neg_of_neg_of_pos`*, which states that dividing a negative number by a positive number yields a negative result, thus completing the proof by contraposition. Note that we are allowing ourselves to use existing lemmas from Mathlib, such as* `div_neg_of_neg_of_pos` *from the* `Field` *module, but not from the* `Rat` *module, to keep the presentation clear and focused on the main proof techniques.*

Now we can prove the main result:

```
lemma rat_add_nonneg (a b : Rat) : 0 ≤ a → 0 ≤ b → 0 ≤ a + b := by

  intro ha hb
  cases a with | div a_num a_den a_den_nz a_cop =>
  cases b with | div b_num b_den b_den_nz b_cop =>
  -- Goal: ⊢ 0 ≤ ↑a_num / ↑a_den + ↑b_num / ↑b_den
  rw[div_add_div] -- applies the addition formula requiring two new goals
  · sorry
  · sorry
  · sorry
```

We first introduce the two hypotheses `ha` *and* `hb` *into the context using* `intro`*. As mentioned earlier, a structure can be viewed as a product type or a record type with a single constructor. The tactic* `cases a with` *exposes the fields of* `Rat`*: the numerator (*`a_num`*), denominator (*`a_den`*), the proof that the denominator is non-zero (*`a_den_nz`*), and the coprimality condition (*`a_cop`*). Notice how the goal transforms the rationals* `a` *and* `b` *into:*

```
⊢ 0 ≤ ↑a_num / ↑a_den + ↑b_num / ↑b_den
```

where ↑ *denotes type coercion from* $\mathbb{Z}$ *or* $\mathbb{N}$ *to* $\mathbb{Q}$*. Now we rewrite the goal using* `rw [div_add_div]`*, a theoprem from the* `Field` *module, which applies the addition formula for division. Let us briefly examine the source code of this theorem:*

```
variable [Semifield K] {a b d : K}

theorem div_add_div (a : K) (c : K) (hb : b ≠ 0) (hd : d ≠ 0) :
    a / b + c / d = (a * d + b * c) / (b * d) := ...
```

The type `K` *here is assumed to be a* `Semifield`*. The* `variable` *keyword is a way to declare parameters that are potentially used across multiple theorems or definitions. We will explore Lean's powerful algebraic hierarchy and the meaning of the square brackets* `[ ]` *in a later section. Using this rewrite is particularly time-saving, since otherwise one would have to establish the well-definedness of rational addition in terms of the underlying structure (a non-trivial task). This theorem requires proofs* `(hb : b ≠ 0)` *and* `(hd : d ≠ 0)`*, generating two additional side goals. We handle each goal separately using the focusing bullet* ·*. The first bullet addresses the main goal (proving the sum is non-negative), while the subsequent bullets discharge the non-zero denominator conditions. I have omitted the actual proofs, here, using* `sorry`*, which we haven't mentioned before.* `sorry` *is a useful feature of Lean that tells the system to accept an incomplete proof for the time being, allowing you to continue development without proving every detail immediately. We can now tackle the remaining goals:*

```
· -- Goal: ⊢ 0 ≤ (↑a_num * ↑b_den + ↑a_den * ↑b_num) / (↑a_den * ↑b_den)
  have hnum_nonneg : (0 : ℚ) ≤ a_num * b_den + a_den * b_num := by
    have ha_num_nonneg := by
      have ha_den_pos := nat_ne_zero_pos a_den a_den_nz
      exact rat_num_nonneg ha_den_pos ha
    have hb_num_nonneg := by
      have hb_den_pos := nat_ne_zero_pos b_den b_den_nz
      exact rat_num_nonneg hb_den_pos hb
    apply add_nonneg -- works for any OrderedAddCommMonoid
    · apply mul_nonneg -- works for any OrderedSemiring
      · exact Int.cast_nonneg.mpr ha_num_nonneg
      · exact Nat.cast_nonneg b_den
    · apply mul_nonneg
      · exact Nat.cast_nonneg a_den
      · exact Int.cast_nonneg.mpr hb_num_nonneg

  have hden_nonneg : (0 : ℚ) ≤ a_den * b_den := by
    rw [← Nat.cast_mul]
    exact Nat.cast_nonneg (a_den * b_den)
  exact div_nonneg hnum_nonneg hden_nonneg

· exact Nat.cast_ne_zero.mpr a_den_nz -- Goal: ⊢ ↑a_den ≠ 0
· exact Nat.cast_ne_zero.mpr b_den_nz -- Goal: ⊢ ↑b_den ≠ 0
```

We introduce two key hypotheses, `hnum_nonneg` *and* `hden_nonneg`*, which will be required by* `div_nonneg` *from the* `GroupWithZero` *module. This lemma provides us with a term that directly validates our statement. Note that* `div_nonneg` *is a generalized lemma that applies not only to rational numbers but to all ordered groups with zero that are also partially ordered. The hypothesis* `hnum_nonneg` *proves that the numerator is non-negative by working with the coerced expressions in* $\mathbb{Q}$*. It uses* `add_nonneg` *and* `mul_nonneg`*, which are general theorems that work for any ordered additive commutative monoid and ordered semiring, respectively. The actual reasoning is done using integer-related theorems (via* `Int.cast_nonneg`*) for the numerators and natural number theorems (via* `Nat.cast_nonneg`*) for the denominators. The hypothesis* `hden_nonneg` *proves that the denominator is non-negative by working entirely with natural numbers. We use the rewrite* `rw [← Nat.cast_mul]`*, which moves the coercion (in this case from* $\mathbb{N}$ *to* $\mathbb{Q}$*) inside the multiplication:* `↑(m * n) = ↑m * ↑n`*. The* $\leftarrow$ *symbol means that we want the transformation from right to left (i.e., we apply the equality in reverse to move the cast inward). Type casts and coercions require these kinds of rewrite rules, not only for multiplication but also for addition and other operations, and similarly for* $\mathbb{Z}$ *or other numerical types. These lemmas, such as* `Nat.cast_mul`*,* `Int.cast_add`*, etc., ensure that algebraic operations commute with type coercions.*

3.2 Coercions and Type Casting

We extensively used type casting and coercions in this proof, which requires some explanation [LM20]. Lean's type system lacks subtyping, means that types like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ are distinct and do not have a subtype relationship. In order to translate between these types, we need to use explicit type casts or rely on automatic coercions. For example, natural numbers ($\mathbb{N}$) can be coerced to integers ($\mathbb{Z}$), and integers can be coerced to rational numbers ($\mathbb{Q}$). Casting and coercion are related but distinct concepts:

- **Casting** refers to the explicit conversion of a value from one type to another, typically using functions like `Int.cast` or `Nat.cast`. These functions have accompanying lemmas that preserve properties across type conversions, such as `Int.cast_lt` and `Nat.cast_pos`.
- **Coercion**, on the other hand, is a more general mechanism that allows Lean to automatically convert between types when needed. More generally, in expressions like `x + y` where `x` and `y` are of different types, Lean will automatically coerce them to a common type. For example, if `x : ℕ` and `y : ℤ`, then `x` will be coerced to $\mathbb{Z}$.

The notation ↑ denotes an explicit coercion (in between cast and coercion). To illustrate the expected behavior of coercion simplification, consider the expression `↑m + ↑n < (10 : ℤ)`, where `m, n : ℕ` are cast to $\mathbb{Z}$. The expected normal form is `m + n < (10 : ℕ)`, since `+`, $<$, and the numeral `10` are polymorphic (i.e., they can work with any numerical type such as $\mathbb{Z}$ or $\mathbb{N}$). The simplification should proceed as follows:

1. Replace the numeral on the right with the cast of a natural number: `↑m + ↑n < ↑(10 : ℕ)`
2. Factor ↑ to the outside on the left: `↑(m + n) < ↑(10 : ℕ)`
3. Eliminate both casts to obtain an inequality over $\mathbb{N}$: `m + n < (10 : ℕ)`

Lean provides tactics like `norm_cast` to simplify expressions involving such coercions. The `norm_cast` tactic normalizes casts by pushing them outward and eliminating redundant coercions, often simplifying proofs significantly by reducing goals to their "native" types.

3.3 Type Classes and Algebraic Hierarchy in Lean

In our proof of `rat_add_nonneg`, we used many generalized lemmas from Mathlib, such as `add_nonneg`, `mul_nonneg`, and `div_nonneg`, which apply to a wide range of types beyond just rational numbers. Similarly, in our earlier work with natural numbers, we used `Nat.le_trans`, a theorem specifically for natural numbers that is part of Lean's core library (`lean/Init/Prelude.lean`). Mathlib is built on top of this base library. However, the transitivity property holds not only for naturals but also for integers, reals, and, in fact, for any partially ordered set.

Rather than duplicating this theorem for each type, Mathlib provides a general lemma `le_trans` that works for any type α endowed with a partial ordering. This is achieved through **type classes**, Lean's mechanism for defining and working with abstract algebraic structures in an ad hoc polymorphic manner. Type classes provide a powerful and flexible way to specify properties and operations that can be shared across different types, thereby enabling polymorphism and code reuse. Ad hoc polymorphism arises when a function or operator is defined over several distinct types, with behavior that varies depending on the type. A standard example [WB89] is overloaded multiplication: the same symbol `*` denotes multiplication of integers (e.g., `3 * 3`) and of floating-point numbers (e.g., `3.14 * 3.14`). By contrast, parametric polymorphism occurs when a function is defined over a range of types but acts uniformly on each of them. For instance, the `List.length` function applies in the same way to a list of integers and to a list of floating-point numbers.

Under the hood, a type class is a structure. An important aspect of structures, and hence type classes, is that they support hierarchy and composition through inheritance. For example, mathematically, a monoid is a semigroup with an identity element, and a group is a monoid with inverses. In Lean, we can express this by defining a `Monoid` structure that extends the `Semigroup` structure, and a `Group` structure that extends the `Monoid` structure using the `extends` keyword:

```
-- A semigroup has an associative binary operation
structure Semigroup (α : Type*) where
  mul : α → α → α
  mul_assoc : ∀ a b c : α, mul (mul a b) c = mul a (mul b c)

-- A monoid extends semigroup with an identity element
structure Monoid (α : Type*) extends Semigroup α where
  one : α
  one_mul : ∀ a : α, mul one a = a
  mul_one : ∀ a : α, mul a one = a

-- A group extends monoid with inverses
structure Group (α : Type*) extends Monoid α where
  inv : α → α
  mul_left_inv : ∀ a : α, mul (inv a) a = one
```

The symbol `*` in `(α : Type*)` indicates a universe variable (we will discuss universes later). Sometimes, to avoid inconsistencies between types (such as Girard's paradox), universes must be specified explicitly. This is an example of universe polymorphism. Thus we have now seen all the polymorphism flavors in Lean: parametric, ad hoc, and universe polymorphism.

Type classes are defined using the `class` keyword, which is syntactic sugar for defining a structure. Thus, the previous example can be rewritten using type classes:

```
-- A semigroup has an associative binary operation
class Semigroup (α : Type*) where
  mul : α → α → α
  mul_assoc : ∀ a b c : α, mul (mul a b) c = mul a (mul b c)

-- A monoid extends semigroup with an identity element
class Monoid (α : Type*) extends Semigroup α where
  one : α
  one_mul : ∀ a : α, mul one a = a
  mul_one : ∀ a : α, mul a one = a

-- A group extends monoid with inverses
class Group (α : Type*) extends Monoid α where
  inv : α → α
  mul_left_inv : ∀ a : α, mul (inv a) a = one
```

The main difference is that type classes support **instance resolution**. We use the keyword `instance` to declare that a particular type is an instance of a type class, which inherits the properties and operations defined in the type class. Instances can be automatically inferred by Lean's type inference system, allowing for concise and expressive code. For example, we can declare that $\mathbb{Z}$ is a group under addition:

```
instance : Group ℤ where
  mul := Int.add
  one := 0
  inv := Int.neg
  mul_assoc := Int.add_assoc
  one_mul := Int.zero_add
  mul_one := Int.add_zero
  mul_left_inv := Int.neg_add_cancel
```

Now, any theorem proven for an arbitrary `Group` α automatically applies to $\mathbb{Z}$ without any additional work. This mechanism is particularly useful for defining and working with order structures like preorders and partial orders. Mathematically, a preorder consists of a set `P` and a binary relation $\leq$ on `P` that is reflexive and transitive [Lea25a].

```
-- A preorder is a reflexive, transitive relation `≤` with `<` defined
    in terms of `≤`
class Preorder (α : Type*) extends LE α, LT α where
  le_refl : ∀ a : α, a ≤ a
  le_trans : ∀ a b c : α, a ≤ b → b ≤ c → a ≤ c
  lt := fun a b => a ≤ b ∧ ¬ b ≤ a
  lt_iff_le_not_ge : ∀ a b : α, a < b ↔ a ≤ b ∧ ¬ b ≤ a := by intros;
    rfl

instance [Preorder α] : Lean.Grind.Preorder α where
  le_refl := Preorder.le_refl
  le_trans := Preorder.le_trans _ _ _
  lt_iff_le_not_le := Preorder.lt_iff_le_not_ge _ _
```

Listing 3: Preorder Type Class in Lean

The `class Preorder` declares a type class over a type α, bundling the $\leq$ and `<` relations (inherited via `extends LE α, LT α`) with the preorder axioms: reflexivity (`le_refl`) and transitivity (`le_trans`). The field `lt` provides a default definition of strict inequality in terms of $\leq$, and the theorem `lt_iff_le_not_ge` characterizes this relationship, proved automatically via reflexivity (`by intros; rfl`). The `instance` declaration connects the `Preorder` class to Lean's `Grind` tactic automation, which allows automatic reasoning with preorder properties during proof search. Returning to our rational number proof, this explains why lemmas like `add_nonneg` and `mul_nonneg` work seamlessly: $\mathbb{Q}$ is an instance of `OrderedSemiring`, which extends `Preorder` and other algebraic structures, automatically providing all their theorems.

An important aspect of type classes is that they can be parameterized by other type classes, enabling the construction of complex hierarchies of algebraic structures. Topological spaces in Mathlib are represented using type classes and they can be derived metric spaces and normed spaces. For example, the `TopologicalSpace` type class defines the structure of a topological space, and it can be extended to define metric spaces and normed spaces, which are topological spaces with additional structure. This hierarchical organization allows for the reuse of definitions and theorems across different mathematical domains, facilitating a modular and scalable approach to formalization.

4 Example in Topology: A Connected but not Path-Connected Space

As part of my thesis work, with the help and revision from Prof David Loeffler, I have formalized a well-known counterexample in topology: the **topologist's sine curve**. This classic example illustrates a space that is **connected** but not **path-connected**. It demonstrates the use of type classes, structures, and several tactics in Lean that were discussed earlier. In my presentation, I will provide a high-level overview along with some specific examples, of my original code, wich is way longer than the final result. My original proof follows Conrad's

paper ([Con]), with a few modifications and some differences from the final formalization **Counterexamples – Topologist's Sine Curve**. The topologist's sine curve is defined as the graph of $y = \sin(1/x)$ for $x \in (0, 1]$, together with the origin $(0, 0)$. We define three sets in $\mathbb{R}^2$:

- S: the oscillating curve $\{(x, \sin(1/x)) : x > 0\}$
- Z: the singleton set $\{(0, 0)\}$
- T: their union $S \cup Z$

The proof merged into the Mathlib library takes Z as $\{0\} \times [-1, 1]$ instead of $\{(0, 0)\}$, resulting in a stronger and more general result. In Lean, this is expressed as follows:

```
open Real Set
def pos_real := Ioi (0 : ℝ)
noncomputable def sine_curve := fun x ↦ (x, sin (x⁻¹))

def S : Set (ℝ × ℝ) := sine_curve '' pos_real
def Z : Set (ℝ × ℝ) := { (0, 0) }
def T : Set (ℝ × ℝ) := S ∪ Z
```

We open the `Real` and `Set` namespaces to avoid prefixing real number and set operations with `Real.` and `Set.`, respectively. We define the interval $(0, \infty)$ as `pos_real`, using the predefined notation `Ioi 0`, from `Set`. The function `sine_curve` maps a positive real number to a point on the topologist's sine curve in $\mathbb{R}^2$. Here, `''` denotes the image of a set under a function. It's noncomputable because it involves the sine function, which is not computable in Lean's core logic. The sets `S`, `Z`, and `T` are defined using set operations. , and `{ (0, 0) }` denotes the singleton set containing the point $(0, 0)$.

`Set` is the type of sets, defined as predicates (i.e., functions from a type to `Prop`). The sets are subsets of the product space $\mathbb{R}^2$, represented as $\mathbb{R} \times \mathbb{R}$. The sin function `sin` is defined in the `Real`.

The goal is to prove that T is connected but not path-connected. Let's start with connectedness.

4.1 T is connected

First of all one can directlly see that S is connected, since it is the image of the set $((0, \infty))$ under the continuous map $x \mapsto (x, \sin(1/x))$ and a interval in $\mathbb{R}$ is connected. Moreover, the closure of S is T, and the closure of a connected subset of a topological space is always connected, so T is connected. This is how a mathematician would argue informally, using known facts. However, in a formal proof, one must justify each step. For instance, justifying that S is connected requires proving that the map $x \mapsto (x, \sin(1/x))$ is continuous on $(0, \infty)$ and that $(0, \infty)$ is connected.

As we have seen, even showing that a rational number is non-negative requires several steps and the use of various lemmas from Mathlib. Similarly,

proving that a set is connected can involve multiple steps for the newer programmer.

We can use the structure `IsConnected` to set up the statement and see if we can argue similarly in Lean.

```
lemma S_is_conn : IsConnected S := by sorry
```

In the file where `IsConnected` is defined, `Topology/Connected/Basic.lean`, we see that it requires S to be nonempty and preconnected. You can verify this by unfolding `IsConnected` in the goal.

```
lemma S_is_conn : IsConnected S := by
  unfold IsConnected
  ⊢ S.Nonempty ∧ IsPreconnected S
  sorry
```

Following the definition of `IsPreconnected`, we see that it captures the usual definition of preconnectedness: that S cannot be partitioned into two nonempty disjoint open sets. This trivially requires nonemptiness to make sense. The `unfold` tactic helps to expand definitions; one can use it to expand the definition of S or `pos_real`, as well as other Mathlib expressions. Reflecting our argument, we can check if Mathlib includes the fact that every interval is connected and that connectedness is preserved under continuous maps. Indeed, in `Topology/Connected/Interval.lean`, we find the theorem `isConnected_Ioi.image`, stating that the image of an interval of the form (a, ∞) under a continuous map is connected.

```
lemma S_is_conn : IsConnected S := by
  apply isConnected_Ioi.image
  -- ⊢ ContinuousOn sine_curve (Ioi 0)
  sorry
```

The `apply` tactic applies the theorem similar to `exact`, but it also tries to unify the goal with the conclusion of the theorem. The theorem `isConnected_Ioi.image` requires proving the continuity of the map on the interval $(0, \infty)$, which is expressed as `ContinuousOn sine_curve (Ioi 0)`. The predicate `ContinuousOn f S` expresses that a function f is continuous on a set S, which is exactly what we need to prove. The function $x \mapsto (x, \sin(1/x))$ is continuous on $(0, \infty)$ as the product of two functions continuous on the given domain: the identity map $x \mapsto x$ and the map $x \mapsto \sin(1/x)$. Here is the full proof in Lean:

```
lemma inv_is_continuous_on_pos_real : ContinuousOn (fun x : ℝ => x⁻¹)
    (pos_real) := by
  apply ContinuousOn.inv₀
  · exact continuous_id.continuousOn
  · intro x hx; exact ne_of_gt hx

lemma sin_comp_inv_is_continuous_on_pos_real : ContinuousOn
 (sine_curve) (pos_real) := by
  apply ContinuousOn.prodMk continuous_id.continuousOn
  apply continuous_sin.comp_continuousOn
  exact inv_is_continuous_on_pos_real
```

The theorem `ContinuousOn.prodMk` states that the product of two functions continuous on a set is continuous on that set, requiring a proof of the continuity of each component. The first component is the identity map, which is continuous on any set. Mathlib provides `continuous_id.continuousOn` for this purpose. The second component is the composition of the sine function with the inverse function. The sine function is continuous everywhere, and for this we can use `continuous_sin`. The method `comp_continuousOn` is accessible from the fact that `continuous_sin` gives an instance of a continuous map and is generalized in the `ContinuousOn` module. The theorem `Continuous.comp_continuousOn` states that the composition of a continuous function with a function that is continuous on a set is continuous on that set, and requires proof of the continuity on the set of the inner function. We separate the proof that the inverse function is continuous on the positive reals into the auxiliary lemma `inv_is_continuous_on_pos_real`. The theorem `continuousOn_inv₀` states that if a function is continuous and non-zero on a set, then its inverse is continuous on that set. We left the proof of the continuity of the identity map wich have proved already. The second argument requires proving that $x \neq 0$ for all x in $(0, \infty)$.

```
  · intro x hx
    exact ne_of_gt hx
```

The hypothesis `hx` states that x is in $(0, \infty)$, which implies that $x > 0$. The theorem `ne_of_gt` states that if a real number is greater than zero, then it is non-zero, which completes the proof. Thus the final proof goes as follows:

```
lemma S_is_conn : IsConnected S := by
  apply isConnected_Ioi.image
  · exact sin_comp_inv_is_continuous_on_pos_real
```

When writing a proof, one starts by working out the informal argument on paper. Then one tries to translate it into Lean, step by step, looking for theorems in Mathlib. Afterwards, one can try to optimize the proof by removing unnecessary steps or refactoring it. Proving properties like continuity and connectedness is very common, and there are obviously ways to achieve this with less work. Let's showcase a refactoring of the entire proof. First, the auxiliary lemmas can be reduced to one-liners.

```
lemma inv_is_continuous_on_pos_real : ContinuousOn (fun x : ℝ => x⁻¹)
    (pos_real) :=
  ContinuousOn.inv₀ (continuous_id.continuousOn) (fun _ hx =>  ne_of_gt
    hx)

lemma sin_comp_inv_is_continuous_on_pos_real : ContinuousOn
 (sine_curve) (pos_real) :=
 ContinuousOn.prodMk continuous_id.continuousOn <|
  Real.continuous_sin.comp_continuousOn <|
    (inv_is_continuous_on_pos_real)
```

We removed the `by` keyword since we can provide a term that directly proves the statement. In `inv_is_continuous_on_pos_real`, we directly apply `ContinuousOn.inv₀` with the two required arguments. Notice that we can use a lambda function `fun _ hx =>  ne_of_gt hx` to prove that $x \neq 0$ for all x in $(0, \infty)$, recall the propositions-as-types correspondence. In the next lemma, we use the `<\|` reverse application operator, which allows us to avoid parentheses by changing the order of application. This means that `f <| g <| h` is equivalent to `f (g h)`. We can inline these two lemmas into the main proof to get a final one-liner:

```
lemma S_is_conn : IsConnected S :=
  isConnected_Ioi.image sine_curve <| continuous_id.continuousOn.prodMk
    <|
    continuous_sin.comp_continuousOn <|
    ContinuousOn.inv₀ continuous_id.continuousOn (fun _ hx => ne_of_gt
    hx)
```

Notice again the use of the **pipe** operator. Reading from left to right, we are building up the proof by successive applications:

- We start with `isConnected_Ioi.image sine_curve`, which states that the image of $(0, \infty)$ under `sine_curve` is connected if we can prove the function is continuous.
- We then apply `continuous_id.continuousOn.prodMk`, which constructs the product of two continuous functions.
- Next, `continuous_sin.comp_continuousOn` provides the continuity of the sine composition.
- Finally, `ContinuousOn.inv₀ continuous_id.continuousOn (fun _ hx => ne_of_gt hx)` proves the continuity of the inverse function on positive reals.

The entire chain can be read as building the continuity proof from the innermost function (the inverse) outward to the complete sine curve function, which is then used to prove that S is connected.

Since the intersection of Z and S is empty, we cannot directly conclude that T is connected from the connectedness of its components alone. However, we can use the fact that every subset between a connected set and its closure is connected.

Theorem 4.1 *Let C be a connected topological space, and denote $\overline{C}$ as its closure. It follows that every subset $C \subseteq S \subseteq \overline{C}$ is connected.*

In Mathlib, this theorem is available as `IsConnected.subset_closure`. We can set up the statement and progress from there.

```
theorem T_is_conn : IsConnected T := by
  apply IsConnected.subset_closure
  · exact S_is_conn -- ⊢ IsConnected ?s
  · tauto_set -- ⊢ S ⊆ T
  · sorry -- ⊢ T ⊆ closure S
```

The theorem requires three goals:

1. That S is connected, which was already proved in `S_is_conn`.
2. That $S \subseteq T$, which is a trivial set operation. The tactic `tauto_set` handles this kind of set tautologies.
3. That $T \subseteq \overline{S}$ (the closure of S), which requires proof.

Let's continue with the final point.

```
lemma T_sub_cls_S : T ⊆ closure S := by
  intro x hx
  cases hx with
  | inl hxS => exact subset_closure hxS
  | inr hxZ =>
      sorry
```

Proving that one set is contained in another can be done naively in a pointwise manner. We introduce an element $x \in \mathbb{R}^2$ together with the proof that $x \in T$. Since T is a union, we use `cases` to separate the two cases:

- When $x \in S$, which is trivially solved by `exact subset_closure hxS`, since any set is contained in its closure.
- When $x \in Z$, which requires more work.

Now a trick. One of the most painful issue in fomralizing math in Lean is the use of existing theorems. One can use several ways to look for the exact theorem. Let's try using the `apply?` tactic to see what the infoview suggests:

```
lemma T_sub_cls_S : T ⊆ closure S := by
  intro x hx
  cases hx with
  | inl hxS => exact subset_closure hxS
  | inr hxZ =>
      apply?
      sorry
```

Depending on the previous work in the file, Lean can already unify the goal with available theorems and suggest the next step. Similar tactics include:

- `exact?` for finding an exact match to close the goal
- `rw?` for suggesting rewrites and definitionally equal replacements
- `simp?` for suggesting simplifications

Another useful tool is Loogle (similar to Haskell's Hoogle), which helps you find theorems by their type signature or name patterns. You can access it at `https://loogle.lean-lang.org/` or use it directly in VS Code. The best approach is to think first about how you would tackle the problem on a piece of paper, as mentioned earlier with informal reasoning. We know that the closure of a set contains all its limit points. To show that the point $(0, 0)$ is contained in the closure of S, we need to show that it is a limit point of S. Thus, one can define a sequence in S tending to $(0, 0)$, and we are done.

For this purpose, we need to work with Lean's Filter library, a powerful tool for dealing with limits and other aspects of topology such as neighborhoods (`𝓝`) and convergence (`Tendsto`). Filters provide a unified framework for expressing notions like "eventually" and "frequently," which are essential for working with limits in a formal setting.

If you are a one-liner enthusiast like me, you don't mind trying to combine bits and pieces to get a clean final result. We can simplify the final theorem as follows initially:

```
theorem T_is_conn : IsConnected T :=
  IsConnected.subset_closure S_is_conn (by tauto_set) T_sub_cls_S
```

With a bit of courage, we can also inline the proof of `S_is_conn` (while `T_sub_cls_S` is way too long to inline) to get a more self-contained one-liner:

```
theorem T_is_conn : IsConnected T :=
  IsConnected.subset_closure (isConnected_Ioi.image sine_curve <|
    continuous_id.continuousOn.prodMk <|
    Real.continuous_sin.comp_continuousOn <|
    ContinuousOn.inv₀ continuous_id.continuousOn
    (fun _ hx => ne_of_gt hx)) (by tauto_set) T_sub_cls_S
```

Here is the link to the entire first part of the proof: (link to Lean live)

4.2 T is not path-connected

References

[] *Placeholder for web:1.*

[Com20] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020.

[Con] Keith Conrad. *Spaces that are connected but not path-connected*. `https://kconrad.math.uconn.edu/blurbs/topology/connnotpathconn.pdf`.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Lea25a] Lean Mathematical Library community. *Mathematics in Lean*. `https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf`. 2025.

[Lea25b] Lean Mathematical Library community. *mathlib — The Lean Mathematical Library*. `https://github.com/leanprover-community/mathlib`. 2025.

[LM20] Robert Y. Lewis and Paul-Nicolas Madelaine. "Simplifying Casts and Coercions". In: *arXiv preprint arXiv:2001.10594* (2020).

[NPS90] Bengt Nordstr"om, Kent Petersson, and Jan M. Smith. *Programming in Martin-L"of's Type Theory: An Introduction*. Oxford University Press, 1990.

[Tho99] Simon Thompson. *Type Theory And Functional Programming*. University of Kent, 1999.

[Wad15] Philip Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84.

[WB89] Philip Wadler and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *16th ACM Symposium on Principles of Programming Languages*. 1989.