Contents

1 Introduction

This serves as a brief starting point for understanding how the Curry-Howard correspondence appears in Lean, as well as being an introduction to the language itslef. 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 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:*

```
let price = 100;
price = "100";   // valid in JavaScript
```

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 simply by adding a type annotation:

```
let price: number = 100;
price = "100";  // Error: Type 'string' is not assignable to type
'number'
```

Nonetheless, TypeScript, even though it has a quite sophisticated type system, cannot fully capture complex data structures. In many cases, program specifications can only be enforced at runtime. Lean, by contrast, is fully grounded in program derivation and static type checking, which is precisely what makes it a powerful **theorem prover**. At its core lies an advanced and flexible type system, capable of expressing and verifying a wide range of mathematical statements. This system is built on on **Type Theory**, a branch of marthematics and logic that aims to provide a foundation for all mathematics and wich is a (abstract) programming language itself.

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 hiostorical ramifications. Creating a language like Lean requirev careful consideration of which rules and features to include. We shall give a brief overview of the historical development of type theory, and an a introduction on what comes next.

(From [Car19] Introduction) Type theory emerged as a fundamental response to Bertrand Russell's paradox. Considers the set $S = \{x \mid x \notin x\}$ (the set of all sets that do not contain themselves). This is a paradoxical construction, leading to the contradiction $S \in S \iff S \notin S$. Ernst Zermelo and Fraenkel addressed the contradiction by introducing Zermelo-Fraenkel set theory (ZFC), which became the standard in modern mathematics. ZFC maintains Set theory and **classical logical** principles while avoiding paradoxes through careful axiomatization. Russell chose a fundamentally different path. He recognized expressions like $A(A)$ or $x \in x$ as ill-typed, introducing his theory of types. Hs's first systematic response was **Ramified Type Theory**, wich turned out to be problematic. In the 1930s, Alonzo Church developed **Lambda Calculus** as a foundation for mathematics, initially pursuing a type-free approach. However, Church's original untyped system suffered from inconsistencies similar to Russell's paradox ([Wad15]). To address these issues, Church introduced the **Simply Typed Lambda Calculus** in 1940 ([Chu40]). This system is a version of **Simple Type Theory**, a framework able to replace set-theory and propositional logic. Lambda calculus influenced the development of many programming languages as being a foundation for functional programming. Per Martin-Löf revolutionized type theory in the 1970s by introducing **dependent types** that can depend on values of other types. Think for instance of a of vectors of length n or a sequence of n elements. **Dependent Type Theory** extends the expressive power of type systems by allowing the direct representation of predicates and quantifiers (in the sense of Frege), powerful enough to replace set theory and predicate logic. Dependent Type Theory is a derivation of **Martin-Löf Type Theory** (also known as **Intuitionistic Type Theory**). Martin-Löf's system embraced constructive (intuitionistic) principles, requiring that the existence of mathematical objects be demonstrated through explicit construction rather than classical proof by contradiction. Martin-Löf Type Theory also introduced **identity types** to represent equality. In the 1980s, Thierry Coquand and Gérard Huet introduced the **Calculus of Constructions** (CoC), synthesizing insights from Martin-Löf's dependent type theory with higher-order **polymorphism**. The Calculus of Constructions served as the theoretical foundation for the Coq proof assistant, one of the most influential interactive theorem provers. The original CoC was later extended with

inductive types to form the **Calculus of Inductive Constructions** (CIC). Inductive types allow for the definition of data structures like natural numbers, lists, and trees. The Lean theorem prover, developed by Leonardo de Moura and others, is also based on CIC but incorporates several important refinements and differences from Coq's implementation.

A central insight in type theory is the **Curry-Howard correspondence**, which establishes a profound connection between logic and computation. Also known as the **propositions-as-types** principle, this correspondence represents one of the most elegant discoveries in the foundations of mathematics and computer science. It serves also well as a good introduction to type theory, and will be used in this discussion. Nontheless it continuelsy shows new applications and interpretations in modern type theories. The Curry-Howard correspondence was independently discovered by multiple researchers. **Haskell Curry** (1934) first observed the connection between combinatory logic and Hilbert-style proof systems. **William Alvin Howard** (1969) significantly extended the correspondence to natural deduction and the simply typed lambda calculus in his seminal work "The Formulae-as-Types Notion of Construction." The correspondence was further developed through **N.G. de Bruijn's AUTOMATH system** (1967), which was the first working proof checker and demonstrated the practical viability of mechanical proof verification. Amongst its technical innovations are a discussion of the irrelevance of proofs when working in a classical context, which is one of the reasons advanced by de Bruijn for the separation between the notions of **type** and **prop** in the system [Tho99]. Lean also adopts this separation. **Per Martin-Löf's type theory** extended the correspondence to dependent types, allowing for the representation of quantifiers and identity types. Modern proof assistants like Coq, Lean, Agda, and Idris all leverage variants of the Curry-Howard correspondence to enable formal verification of mathematical theorems and software correctness properties.

2 Propositional and Predicate Logic

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 quantifiers and predicates. 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 are constructed via **formulas** built from **propositional variables** (also called atomic propositions) combined with logical **connectives** such as "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$), "implies" ($\Rightarrow$), and "if and only if" ($\Leftrightarrow$). These connectives allow the creation of complex or compound propositions.

Definition 2.1 (Propositional Formula) *([Tho99]) A* ***propositional formula*** *is either:*

- *A* ***propositional variable:*** $X_0, X_1, X_2, \ldots,$ *or*
- *A* ***compound formula*** *formed by combining formulas using connectives:*

$$(A \wedge B), \quad (A \Rightarrow B), \quad (A \vee B), \quad \bot, \quad (A \Leftrightarrow B), \quad (\neg A)$$

  where A and B are formulas themselves.

We are going to describe classical logic though a formal framework called **natural deduction system** developed by Gentzen in the 1930s ([Wad15]). 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.

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 ($\to$)

- Introduction

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

- Elimination (Modus Ponens)

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

Notation 2.3 *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*

$$\begin{matrix} A \\ \vdots \\ B \end{matrix}$$

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 ∧ 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 2.4 *Let's look at our first Lean example*

```
example (H_A : A) (H_B : B) : (A ∧ 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 ∧ 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 → q → (p ∧ 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** approach (see [Tho99] and [NPS90]). 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 2.5 (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 $B \wedge C$:

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

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

```
theorem and_associative : (A ∧ B) ∧ C → A ∧ (B ∧ C) :=
  fun h : (A ∧ B) ∧ C =>
    -- First, from the assumption (A ∧ B) ∧ C, we can derive A:
    have ab : A ∧ B := h.left -- extracts (derive) a proof of (A ∧ B)
    from the assumption
    have a : A := ab.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 c : C := h.right
    have b : B := ab.right
    -- Finally, combining these derivations we obtain A ∧ (B ∧ C)
    show A ∧ (B ∧ C) from ⟨a, ⟨b, c⟩⟩
```

Listing 1: Associativity of Conjunction in Lean

We introduce the `theorem` with the name `and_associative`, which can be referenced in subsequent proofs. The type signature `(A ∧ B) ∧ C → A ∧ (B ∧ C)` represents our logical implication. The `:=` operator introduces the proof term that establishes the theorem's validity. In the previous code example this proof was directly given. Here, we construct it 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 of before. Finally, the `show` keyword presents our final result.

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 α → `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. `Prop` stands for *proposition*, and it is an essential component of Lean's type system. Thus given an `x :` α (an element with type α) `P(x) : Prop` would be representative of a proposition holding for `x`.

When introducing variables into a formal language we must keep in mind that the specific choice 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. Some variables are **bound** (constrained), while others remain **free** (arbitrary, in programming often called "dummy" variables). When substituting variables, it is important to ensure that this distinction is preserved. This phenomenon, called **variable capture**, parallels familiar mathematical practice: if $f(x) \coloneqq \int_1^2 \frac{dt}{x-t}$, then $f(t)$ equals $\int_1^2 \frac{ds}{t-s}$, not the ill-defined $\int_1^2 \frac{dt}{t-t}$. The same principle applies to predicate logic. For example, consider

$$\exists y.\, (y > x).$$

This states that for a given x there exists a y such that $y > x$. If we naively substitute $y + 1$ for x, we would obtain

$$\exists y.\, (y > y + 1),$$

where the y in $y + 1$ has been **captured** by the quantifier $\exists y$. This transforms the original statement from "there exists some y greater than the free variable

x" into the always-false statement "there exists some y greater than itself plus one." To avoid the probelm, in the above example, we would first rename the bound variable to something fresh say z, obtaining $\exists z. (z > x)$, and then safely substitute to get $\exists z. (z > y + 1)$.

Notation 2.7 *We use the notation $\phi[t/x]$ for **substitution**, meaning all occurrences of the free variable x in formula (or expression) ϕ are replaced by term t.*

We can now present the inference rules for quantifiers.

Universal Quantification ($\forall$)

- Introduction

$$\frac{A}{\forall x.\, A}\ \forall I$$

  The variable x must be arbitrary in the derivation of A. This rule captures statements like $\forall x \in \mathbb{N}$, x has a successor, but would not apply to $\forall x \in \mathbb{N}$, x is prime (since we cannot derive this for an arbitrary natural number).

- Elimination

$$\frac{\forall x.\, A}{A[t/x]}\ \forall E$$

  The conclusion $A[t/x]$ represents the substitution of term t for variable x in formula A. From a proof of $\forall x.\, A(x)$ we can infer $A(t)$ for any term t.

Existential Quantification ($\exists$)

- Introduction

$$\frac{A[t/x]}{\exists x.\, A}\ \exists I$$

  The substitution premise means that if we can find a specific term t for which $A(t)$ holds, then we can introduce the existential quantifier. The introduction rule requires a witness t for which the predicate holds.

- Elimination

$$\frac{\exists x.\, A \qquad [A] \vdash B}{B}\ \exists E$$

  To eliminate an existential quantifier, we assume A holds for some witness and derive B without making any assumptions about the specific witness.

We can give an informal reading of the quantifiers as infinite logical operations:

$$\forall x.\, A(x) \equiv A(a) \wedge A(b) \wedge A(c) \wedge \ldots$$
$$\exists x.\, A(x) \equiv A(a) \vee A(b) \vee A(c) \vee \ldots$$

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. Morover, there is a duality between universal and existential quantification. We shall develop all this dicussions further after exploring their computational (type theoretical) meaning.

Example 2.8 *Lean espresses quantifiers as follow.*

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

Listing 2: For All

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

Listing 3: Exists

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

Example 2.9 (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⟩
```

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 2.10 (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 2.11 *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
```

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**. 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, in earlier examples, Lean automatically inferred that the types of A, B, and C were `Prop`. 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 de facto 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 2.12 *The code can be made more readable using tactic mode. This mode comes with an info view showing the goal to solve and proof structure. 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 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 (with the info view) shows the current goal and context. Move your cursosr just before* `by` *and observe the info view change. The goal is shwon displayed* `⊢ Transitive fun x1 x2 ↦ x1 ≤ x2` *at first. The tactic* `intro` *introduces the variables and hypotheses corresponding to the universal quantifiers and assumptions. Now position your cursor just before* `exact` *and observe the info view again. The goal is now* `⊢ x ≤ z`*, with the context showing the variables ans hypothesis introduced by the previous tactic.* `exact` *closes the goal by supplying the term* `Rat.le_trans hxy hyz` *that matches with the goal (the specification of* `Trqansitive` *). You can over each tactic to see its definition and documentation.*

In these examples we havec used predeined lemmas such as `Nat.le_trans` and `Rat.le_trans`, just to simplify the presentation. We can now dig in to 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). Lean has built-in types `Nat` (natural numbers), `Int` (integers), and `Rat` (rational numbers). While Lean provides basic arithmetic and order relations for these types, many advanced properties and theorems live in Mathlib.

```
  -- mathlib4/Mathlib/Algebra/Order/Ring/Unbundled/Rat.lean
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. You can follow the proof step by step using the info view in tactic mode, by piositioning the cursor on each line and observing the changes in the goal and context. 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 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 2.13 *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
-- #eval evaluate the expression and print the result
#eval half.den -- outputs 2
#eval half + third -- outputs 5/6
-- #check prints the type of an expression
#check half.den -- outputs : Nat
#check half -- outputs : Rat
#check half + third -- outputs : Rat
```

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 2.14 *I will present a simplified version of this implementation.*

```
import Mathlib
open Rat

lemma add_nonneg_simplified : 0 ≤ a → 0 ≤ b → 0 ≤ a + b := by
  intro ha hb
  -- Convert hypotheses to numerator conditions
  rw [← num_nonneg] at ha hb
  -- Express rationals in divInt form and apply addition formula
  rw [← num_divInt_den a, ← num_divInt_den b, divInt_add_divInt]
  -- Use divInt_nonneg_iff_of_pos_right to reduce to integer arithmetic
  · rw [divInt_nonneg_iff_of_pos_right]
    · -- Prove numerator ≥ 0
      exact Int.add_nonneg (Int.mul_nonneg ha (Int.natCast_nonneg _))
                          (Int.mul_nonneg hb (Int.natCast_nonneg _))
    · -- Prove denominator > 0
      norm_cast
      exact Nat.mul_pos (Nat.pos_of_ne_zero a.den_nz)
     (Nat.pos_of_ne_zero b.den_nz)
  · norm_cast; exact a.den_nz
  norm_cast; exact b.den_nz
```

In this version, we open the `Rat` **namespace** *to access its definitions and lemmas directly (Notice that i use* `num_nonneg` *instead of* `Rat.num_nonneg` *in the next line). The proof begins by introducing the hypotheses* `ha` *and* `hb` *that* `a` *and* `b` *are nonnegative. The* `rw ... at` *tactic rewrites these hypotheses using the lemma* `num_nonneg`*, which states that a rational number is nonnegative if and only if its numerator is nonnegative. We use ← to indicate the direction of rewriting (from right to left). Next, we express the rational numbers in terms of their numerator and denominator using* `num_divInt_den`*, and apply the addition formula for rational numbers represented as* `divInt_add_divInt`*. The goal is then to prove that the resulting numerator is nonnegative.* `num_divInt_den` *transofms a rational number* `r` *into the form* `r.num /. ↑r.den`*. The ↑ symbol denotes the coercion from natural numbers to integers (remember in our definition of Rat, the numerator is an integer and the denominator a natural number, but here we need to translate everything to integers). We now have 3 goals. We use* `divInt_nonneg_iff_of_pos_right` *to reduce this to proving that the numerator is nonnegative, given that the denominator is positive. This requires two subgoals: proving the numerator is nonnegative and the denominator is positive. For the numerator, we use* `Int.add_nonneg` *to show that the sum of two nonnegative integers is nonnegative. For the denominator, we first translate the problem from integers to natural numbers, using* `norm_cast`*. and use* `Nat.mul_pos` *to show that the product of two positive natural numbers is positive. Finally, we use* `norm_cast` *to handle the necessary type casts between integers and natural num-*

bers automatically and close the remaining goals with the nonzero denominator conditions given from the Rat structure (`den_nz`). Lean encourages to separate subgoals with · and proper indentation, making the corresponding proof more readable.

We made extensive use of type casting and coercions in this proof, handled by the `norm_cast` tactic,wich requires some explanation ([LM20]). Lean type system lack of subtypes means that types like `Nat`, `Int`, and `Rat` are distinct and do not have a subtype relationship. In order to translate between these types, we need to use explicit type casts or coercions. For example, natural numbers (`Nat`) can be coerced to integers (`Int`) and integers can be coerced to rational numbers (`Rat`). The `norm_cast` tactic simplifies expressions involving such coercions by normalizing them, making it easier to reason about mixed-type expressions. It will be otherwise a long and tedious process to manually insert and manage these coercions throughout the proof. `norm_cast` is another example of a tactic that leverages Lean's metaprogramming capabilities to automate common proof patterns. (I CAN DISCUSS THIS FURTHER IF NEEDED).

The theorem previously used with natural numbers, `Nat.le_trans`, is part of Lean's internal library at /lean/Init/Prelude.lean. Mathlib is built on top of this base library. More generally, the transitivity property holds not only for naturals but also for integers, reals, and, in fact, for any partially ordered set. Mathlib provides a general lemma `le_trans` for any type α endowed with 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 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 length function applies in the same way to a list of integers and to a list of floatingpoints.

Under the hood, a type class is a structure. An important aspect of structures, and hence type classes, is that they are powered by hierarchy and composition. For example, 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 `*` on `(`α `: Type*)` indicates a universe variable (we will discuss universes later). Sometimes, in order to avoid inconsistencies between types (like Girard's paradox), universes must be specified explicitly. This is an example of universe polymorphism, thus we have seen all the polymorphism flavors in Lean. Type classes are defined using the `class` keyword, which is syntactic sugar for defining a structure. The differnce is that type classes support **instance resolution**, using the keyword `instance` to declare that a particular type is an instance of a type class, wich inherits the properties and operations defined in the type class. Morover a class can extend other classes, allowing for the composition of properties and operations. For example, we can define a type class for a preorder, which is a set equipped with a reflexive and transitive relation derived from the less-than-or-equal and less-than type classes.

Instances of a type class can be automatically inferred by Lean's type inference system, allowing for concise and expressive code. This mechanism is particularly useful for defining and working with algebraic structures, such as groups, rings, and fields, as well as order structures like preorders and partial orders. Mathematically, a partially ordered set consists of a set `P` and a binary relation $\leq$ on P that is transitive and reflexive ([Lea25a] Structures)

```
-- A preorder is a reflexive, transitive relation ‘≤‘ with ‘a ≤ b‘
    defined in the obvious way.
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 4: 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 alpha, LT alpha`) with the preorder axioms: reflexivity (`le_refl`) and transitivity (`le_trans`). The theorem `lt_iff_le_not_ge` provides a characterization of the strict order, proved automatically (`by intros; rfl`). The `instance` declaration connects the `Preorder` class to Lean's `Grind` tactic automation, which allows automatic reasoning with preorder properties.

This design pattern is the foundation of Lean's powerful mathematical library, allowing complex abstract algebraic and order structures to be expressed succinctly and compositionally.

3 Constructive Mathematics

Mathematicians have traditionally worked within **classical logic**, using **sets** as the primary means of structuring mathematical objects. In contrast, **type theory** does not take sets as its primitive notion, nor is it built by first applying logic and then adding structure. Instead, logic is internal to type theory and is based on constructive (intuitionistic) logic, introduced by Brouwer, formalized by Heyting (see, e.g., [GTL89]). A major point of departure from classical logic is that, in constructive logic, statements cannot simply be classified as true or false; their truth depends on whether a proof exists. There are many conjectures, such as the Riemann Hypothesis, for which we do not yet know whether a proof or disproof exists, so we cannot say whether they are true or false. Consequentely constructiver logic does not universally accept principles such as the **the axiom of choice** or **law of excluded middle** (every proposition is either true or false) as axioms. As a consequence a proof by contraddiction will not work in this setting. Constructive logic emphasizes that a statement is only considered true if we can explicitly construct a proof or provide a **witness** for it. This is what makes constructive mathematics inherently **computable**, and it has important consequences for how we work in type theory and, by extension, in Lean. We already touched on this concept in the previous section. In particular, we presented the logical connectives via the Brouwer–Heyting–Kolmogorov (BHK) interpretation and emphasized that, constructively, a proof of existence consists of a pair: a witness together with a proof that the stated property holds for that witness.

Example 3.1 *We give a constructive proof in Lean that there exist natural numbers a and b such that $a + b = 7$:*

```
example : ∃ a b : Nat, a + b = 7 := by
  use 3
  use 4
```

*use is a tactic that must be imported from the external library **Mathlib**. It assigns the value 3 to a and 4 to b. Lean will then automatically evaluate the expression and verify that both sides are equal. This example is simple enough for Lean to infer the final step on its own.*

This example should be readable even if you have never worked with Lean. It reads: $\exists a, b \in \mathbb{N}$ such that $a + b = 7$. The operator `:=` expect a proof of such a statment. Using `by` we enter Lean **tactic mode**. `use` is a tactic used for dealing with existential quantifiers imported from Mathlib. In classical mathematics, one might attempt a proof by contradiction. However, this approach is not directly accepted in constructive mathematics, as it doesn't provide explicit witnesses for the claimed objects. Nonetheless, while constructive at its core, Lean allows users to invoke classical principles, such as contraposition or proof by contradiction, through tactics like `exfalso`. Thanks to these functionalities, Lean ensures that such reasoning can be translated into a constructive form aka term mode.

Example 3.2 *Here an example of proving something by contraddiction:*

```
example (p : Prop) ( h : False ) : p := by
  exfalso
  · exact h
```

*The **example** takes a proposition p to prove and a false hypothesis h. If we move our cursor just after the **by** keyword, Lean's infoview will show that the current goal is to prove the proposition p, i.e., $\vdash p$. The tactic **exfalso** transforms the goal into $\vdash$ False, which means we now need to derive a contradiction, in other words, we must provide something that leads to False. For the sake of the example, we already have a false hypothesis h, which can be "applied" using the **exact** tactic.*

Some key principles of constructive math include: (From Naive Type Theory)

- $P \vee \neg P$ is not allowed in general.
- $\forall(x : \mathbb{N}).(\text{isPrime}(x) \vee \neg\text{isPrime}(x))$ works because primality is a decidable property.
- $\forall(x : \mathbb{N}).(\text{isProgram}(x) \vee \neg\text{isProgram}(x))$ is not decidable.
- $\neg\neg P \to P$ is allowed because it can be constructively proven (Negative Translation).

I can connect the discssion to Mathlib and Lean: - computability and decidability in Lean (the decidable typeclass in Mathlib) decide tactics noncomputable keyword - constructive proofs and classical reasoning in Lean - constructive analysis in Lean (real numbers in Mathlib, constructive reals in Lean 4) - discuss Set in Lean 4 and Mathlib - inverse function and axiom of choise - extensional vs intensional? - FinSet in Mathlib that relates to computability and decidability

Reasoning about finite sets computationally requires having a procedure to test equality on α, which is why the snippet below includes the assumption `[DecidableEq` α`]`. For concrete data types like N, Z, and Q, the assumption is satisfied automatically. When reasoning about the real numbers, it can be satisfied using classical logic and abandoning the computational interpretation

References

[Car19] Mario Carneiro. *The Type Theory of Lean*. Manuscript. Accessed: September 20, 2025. Apr. 2019. URL: https://github.com/digama0/lean-typetheory/.

[Chu40] Alonzo Church. *A Formulation of the Simple Theory of Types*. Vol. 5. 2. Association for Symbolic Logic, 1940, pp. 56–68.

[Com20] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*. New Orleans, LA, USA: ACM, 2020, pp. 367–381. DOI: 10.1145/3372885.3373824. URL: https://leanprover-community.github.io/papers/mathlib-paper.pdf.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Vol. 7. Cambridge Tracts in Theoretical Computer Science. Translated and adapted from the French book "Le Point Aveugle". Cambridge University Press, 1989.

[Lea25a] Lean Mathematical Library community. *Mathematics in Lean*. https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf. Version v4.23.0-rc2. 2025. URL: https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf.

[Lea25b] Lean Mathematical Library community. *mathlib — The Lean Mathematical Library*. https://github.com/leanprover-community/mathlib. Version v4.23.0-rc2. 2025. URL: https://github.com/leanprover-community/mathlib.

[LM20] Robert Y. Lewis and Paul-Nicolas Madelaine. "Simplifying Casts and Coercions". In: *arXiv preprint arXiv:2001.10594v2* (2020). arXiv: 2001.10594 [cs.PL]. URL: https://arxiv.org/abs/2001.10594.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Now available from https://www.cse.chalmers.se/research/group/logic/book/book.pdf. Oxford, UK: Oxford University Press, 1990. URL: https://www.cse.chalmers.se/research/group/logic/book/book.pdf.

[Tho99] Simon Thompson. *Type Theory & Functional Programming*. March 1999. Computing Laboratory, University of Kent, 1999.

[Wad15] Philip Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407. URL: https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf.

[WB89] Philip Wadler and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *16th ACM Symposium on Principles of Programming Languages (POPL)*. Austin, TX, USA: ACM, Jan. 1989, pp. 1–12. URL: `https://web.engr.oregonstate.edu/~walkiner/teaching/cs583-sp21/files/Wadler-TypeClasses.pdf`.