Contents

1 Logic and Proposition as Types **1**

2 Judgments and Propositions **4**
2.1 Predicate lgic and dependency 8
2.2 Describing and use properties 9

3 TODOs and Next Sections **16**

1 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 1.1 (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 1.2 (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 1.3 $$\frac{\Gamma, \quad p_1 : P_1 \quad p_2 : P_2 \quad \cdots \quad P_n}{C}$$

Notation 1.4 *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 1.5 (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 [**web:1**]:*

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 $\Gamma = x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ [**web:2**]. The context encodes the **assumptions** or **hypotheses** under which a judgment is made.*

For example, consider these concrete judgments:

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

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

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

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

Example 1.6 (Context Formation Rules) *The context itself must be well-formed according to specific rules [***web:41***]:*

$$\frac{}{\diamond\ ctx}$$

Empty context rule*: The empty context $\diamond$ (or $\emptyset$) is always well-formed.*

$$\frac{\Gamma\ ctx \qquad \Gamma \vdash A\ type}{\Gamma, x : A\ ctx}$$

Context extension rule*: If Γ is a well-formed context and A is a well-formed type in Γ, then extending Γ with a fresh variable x of type A yields a well-formed context.*

Example 1.7 (Typing Rules with Contexts) *Here are fundamental typing rules that show how contexts work in practice:*

$$\frac{\Gamma, x : A, \Delta\ ctx}{\Gamma, x : A, \Delta \vdash x : A}$$

Variable rule*: A variable x has its declared type A in any context where it appears.*

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

Application rule*: Function application preserves typing under the same context assumptions.*

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\ b : A \to B}$$

Lambda abstraction rule*: To form a function of type $A \to B$, assume $x : A$ and derive $b : B$.*

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)

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

2 Judgments and Propositions

Logic is often formalized through a framework that distinguishes clearly between **judgments** and **propositions**, following Martin-Löf's foundational approach ([**martin-lof-1983**], [**plato:intuitionistic-type-theory**]). A **judgment** represents something we may know — an object of knowledge that becomes evident once we have a proof of it.

The most fundamental form of judgment in logic is "*A is true*", where A is a proposition. This is formally written as A true. When we derive such judgments under assumptions, we write

$$\Gamma \vdash A \text{ true},$$

where Γ represents our hypothetical assumptions ([**pfenning-natded**]).

Example 2.1 (Judgment-Based Inference Rules) *All logical reasoning can be expressed through **introduction** and **elimination** rules for judgments. For instance, conjunction is characterized as follows.*

Introduction: *how to establish the judgment $A \wedge B$ true:*

$$\frac{A \textit{ true} \qquad B \textit{ true}}{A \wedge B \textit{ true}}\ \wedge\text{-}I$$

Elimination: *how to use the judgment $A \wedge B$ true:*

$$\frac{A \wedge B \textit{ true}}{A \textit{ true}}\ \wedge\text{-}E_1 \qquad\qquad \frac{A \wedge B \textit{ true}}{B \textit{ true}}\ \wedge\text{-}E_2$$

Notation 2.2 (The Turnstile Symbol) *The symbol $\vdash$ (the turnstile) separates assumptions from conclusions in judgments.*

$$\Gamma \vdash J$$

means that the judgment J follows from the assumptions Γ, or equivalently, that J is evident given evidence for Γ.

When assumptions are discharged during reasoning, we indicate this with square brackets $[A]$. For example, to establish an implication $A \to B$, we assume A (written $[A]$), derive B under this assumption, then discharge A:

$$\frac{[A\ \mathit{true}]^u \qquad \vdots \qquad B\ \mathit{true}}{A \to B\ \mathit{true}} \to\text{-}I^u$$

This judgment-based framework extends naturally to **type theory**, where additional judgment forms are introduced. While logic focuses on the judgment A true, type theory introduces several fundamental forms ([**plato:intuitionistic-type-theory**]):

$\Gamma \vdash A$ type	(A is a well-formed type)
$\Gamma \vdash t : A$	(t is a term of type A)
$\Gamma \vdash A \equiv B$ type	(types A and B are judgmentally equal)
$\Gamma \vdash t_1 \equiv t_2 : A$	(terms t_1 and t_2 are judgmentally equal)

The **context** Γ represents a list of assumptions about variables and their types:

$$\Gamma = x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n.$$

Example 2.3 (Unified Introduction/Elimination Pattern) *Both logical connectives and type constructors follow the same introduction/elimination pattern. For function types (corresponding to implication):*

Formation:

$$\frac{\Gamma \vdash A\ \mathit{type} \qquad \Gamma \vdash B\ \mathit{type}}{\Gamma \vdash A \to B\ \mathit{type}}$$

Introduction:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\, b : A \to B} \to\text{-}I$$

Elimination:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \to\text{-}E$$

This correspondence reveals a deep connection: logical implication and function types share the same structural rules, differing only in focus — whether on truth ($A \to B$ true) or on typing ($\lambda x.\, b : A \to B$).

References

Per Martin-Löf, *Intuitionistic Type Theory*, 1983–1984.
P. Dybjer, *Intuitionistic Type Theory*, *Stanford Encyclopedia of Philosophy*, 2016.
Frank Pfenning, *Logical Frameworks and Natural Deduction*, lecture notes.
Additional formal presentations and lecture notes on type theory and natural deduction, CMU (various).

rules `And.left` and `And.right`.

Example 2.4 *Let's look at our first Lean example:*

```
example (a b : Prop) (ha : a) (hb : b) : (a ∧ b) := And.intro ha hb
```

This illustrates the ***introduction rule*** *for conjunction in Lean. The declaration*

```
(a b : Prop) (ha : a) (hb : b) : (a ∧ b)
```

means that, given proofs `ha` *and* `hb` *of the propositions* `a` *and* `b`*, we can construct a proof of* `(a ∧ b)`*. Try placing the cursor just before* `And.intro`*. Observe how the infoview updates to display:*

```
a b : Prop
ha : a
hb : b
⊢ a ∧ b
```

Here we see again the turnstile symbol (⊢), introduced earlier. Everything before it represents the ***context*** *(Γ)—our variables and hypotheses— while everything after it denotes the* ***goal*** *to be proved. Lean implements the constructor* `And.intro` *as:*

```
And.intro : p → q → (p ∧ q)
```

This means: given a proof of `p` *and a proof of* `q`*, Lean can produce a proof of* `(p ∧ q)`*. We therefore complete the proof by directly providing* `And.intro ha hb`*. Alternatively, we can express the same statement more compactly:*

```
example (a b : Prop) (ha : a) (hb : b) : (a ∧ b) := ⟨ha, hb⟩
```

Here, `⟨ha, hb⟩` *is a* ***pair*** *(or product) term— you can think of it as a Cartesian pair combining the two proofs into one.*

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]). 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 : Prop) : (a ∧ b) ∧ c → a ∧ (b ∧ c) :=
fun h : (a ∧ b) ∧ c =>
-- First, from the assumption (a ∧ b) ∧ c, we can derive a:
have hab : a ∧ 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 ∧ (b ∧ 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 ∧ b) ∧ c
hab : a ∧ b
ha : a
hc : c
hb : b
⊢ a ∧ b ∧ 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 lgic 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**: ∀ ("for all") and ∃ ("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`.

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.

Example 2.7 *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 2.8 (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 2.9 (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 2.10 *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
```

2.2 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 2.11 *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* ⊢ `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* ⊢ `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.*

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

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 2.13 *We present a simplified implementation of addition non-negativity for rationals, 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). Lemmas are typically namespaced (e.g., Rataddnonneg) and often marked protected to prevent namespace pollution. 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
```

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
```

```
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
  have hnum_nonneg : (0 : ℚ) ≤ a_num * b_den + a_den * b_num := by
    apply add_nonneg
    · apply mul_nonneg
      · 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 :=
    Nat.cast_nonneg _
  exact div_nonneg hnum_nonneg hden_nonneg
```

Let us examine this proof step-by-step. 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 ℤ or ℕ to ℚ. Now we rewrite the goal using `rw [div_add_div]`, 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) := sorry
```

Notice that we are not using a theorem from `Rat` specifically, but rather from `Field`. 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 `[ ]` (which denote typeclass instances) in the next section. I have omitted the actual proof 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. 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 involving quotient types. 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 · (entered by typing `\.` or `\cdot`). The first bullet addresses the main goal

(proving the sum is non-negative), while the subsequent bullets discharge the non-zero denominator conditions.

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.

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. On the other hand, Type classes are defined using the `class` keyword, which is syntactic sugar for defining a structure. Thus the previuous example can be rewritten similarly, using type classes: The main diffrence 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.

Let's look at a more concrete example, say we would like to define a a generic `Transitive` structure with field `le_trans`: Now in overload differnt typeds such as Rat, INtn and NAt we would define:

This idea is extensively used in Mathlib to define and work with algebraic structures.

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)

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 TODOs and Next Sections

- Revisit Section 2
  - Make it more practical by removing most of the rules and type theory details.
  - Add more Lean code examples and emphasize the Curry–Howard correspondence.
- Example on Rational Numbers
- Overview of `mathlib`
- Lean Tactics
- Type Coercions
- Structures and Type Classes
- Constructive Mathematics (briefly):
  - Decidable and inherited type classes
  - `noncomputable` theorems/lemmas vs. `def`, etc.
- Construction of Rationals and the Classical Logic Module
- Sets in Lean
- Topology in Lean
- Filters
- Describe the code example submitted (this will start in the previous sections)
- This is the core part of the work; if needed, I can expand it with more type-theoretical discussion, as I spent quite some time studying it.

References

[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.

[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.