Day - 3 : Basic Tactics and First Proofs

Touseef Haider

May 13, 2025

Now that we have a sense of **propositions as types** and **proofs as objects**, let's get our hands dirty (figuratively speaking!) and see how to actually construct these proof objects in Lean. This brings us to:

3.1. The Concept of a Goal in a Proof

When you start writing a proof in Lean, you begin by stating the theorem you want to prove. For example:

Listing 1: Lean Theorem Example

```
import Mathlib.Data.Nat.Basic -- Or a simpler import if you don't have Mathlib fully set up
-- e.g., sometimes just 'import Lean' gives basic Nat definitions

theorem add_zero_example (n : Nat) : n + 0 = n := by
  -- Tactics will go here
  sorry -- 'sorry' is a placeholder tactic that "solves" any goal
        -- remove it to see the actual goal
```

Let's break this down:

- theorem add_zero_example: This declares that we're about to state and prove a theorem named add_zero_example.
- `(n : Nat)`: This says that our theorem will talk about some arbitrary natural number `n`. This `n` is a *hypothesis* or *assumption* for our proof—we get to assume `n` is a natural number.
- `: n + 0 = n`: This is the **proposition** we want to prove. In Lean's "propositions as types" view, this is the **type** for which we need to construct a term (our proof).
- `:=`: This means "is defined as" or "is proven by."
- `by`: This block is where we write the actual steps of our proof, using *tactics*.

The Goal

As soon as you write `by`, Lean's Infoview will display the "goal state." For the example above, before we write any tactics, the Infoview would show something like this:

```
1 goal
n : Nat
\dashv n + 0 = n
```

- `1 goal`: You have one thing you currently need to prove.
- `n : Nat`: This is your **local context** or **hypotheses**. It lists what you know or assume. Here, we know `n` is a natural number.
- $\dashv n + 0 = n$: The $\dashv$ symbol (often called a "turnstile") separates your knowns from what you need to prove. The expression $n + 0 = n$ is your current **goal**.

Your job as the prover is to use **tactics** to manipulate this goal state, simplify it, or match it with your hypotheses, until the goal disappears or is transformed into something obviously true. Each tactic you apply aims to construct a part of the underlying "proof object."

Think of it like a puzzle. The Infoview shows you the picture you need to complete (the goal) and the pieces you have (the hypotheses). Tactics are your tools for fitting those pieces together.

So, your Infoview shows:

```
1 goal
n : Nat
 n + 0 = n
```

This means Lean is asking you to prove that "for any natural number `n`, `n` plus `0` equals `n`."

3.2. The `rfl` Tactic: Proof by Reflexivity

Now, how do we prove this? In mathematics, `n + 0 = n` is often true *by definition* of addition with zero (it's part of how addition on natural numbers is defined, or a very basic property).

Lean has a tactic for exactly this kind of situation: `rfl`.

`rfl` stands for "reflexivity." It attempts to prove that the left-hand side of an equality is definitionally equal to the right-hand side. This means Lean will try to unfold definitions and simplify both sides to see if they become identical.

How to use it:

In your Lean file, on the line where your cursor is (where `sorry` used to be), type `rfl`:

Listing 2: Using the rfl Tactic

```
import Mathlib.Data.Nat.Basic -- Or your current import setup

theorem add_zero_example (n : Nat) : n + 0 = n := by
  rfl -- This is our proof!
```

What to Expect in the Infoview:

Once you type `rfl` (and Lean processes it, which is usually instant), look at the Infoview. It should now say something like:

```
goals accomplished
```

This means the `rfl` tactic successfully proved the goal! Lean was able to see that, according to its definitions (likely from `Mathlib.Data.Nat.Basic` or its core library), `n + 0` is indeed the same as `n`. You've constructed your first proof object!

Why this is important:

The statement `n + 0 = n` is a fundamental property often called the "identity element for addition." Using `rfl` tells Lean, "Hey, this is true by definition or by basic computation. Just check it!"

Go ahead and try adding `rfl` to your proof. Let me know what the Infoview says! This is your first step from stating a problem to solving it with Lean's help.

3.3. Practice Time

Let's try a slightly different, but still very similar, algebraic identity. How about proving that 0 + n = n for any natural number n?

```
import Mathlib.Data.Nat.Basic -- Or your current import setup

theorem zero_add_example (n : Nat) : 0 + n = n := by
  -- Your proof tactic goes here!
```

Definitional Equality vs. Propositional Equality:

- `rfl` works when two things are equal by their very definition or by a very simple computation that Lean can do directly by unfolding definitions. For natural numbers in Lean, addition $m + n$ is often defined by recursion on the second argument n. So $n + 0 = n$ is usually true by one of the base cases of this definition.
- However $0 + n = n$, while mathematically true, might not be directly how the + operation is defined. It is usually a theorem that itself needs to be proven from the basic definitions (often by induction, or using other established theorems).
- So, Lean is saying "I cannot just unfold definitions to see that $0 + n$ is the same as n. You need to show me why they are equal using other known facts."

This does not mean your mathematical intuition is wrong. It just means `rfl` is too simple for this case. We need a more powerful tool, or we need to use an existing theorem.

This is the perfect time to introduce our next tactic!

3.4. The `rw` (rewrite) Tactic

When `rfl` isn't enough, but you know an equality that *should* help, the `rw` tactic is your friend. `rw` stands for "rewrite."

You use `rw [some_lemma]` to rewrite the goal using `some_lemma`.

- If `some_lemma` is an equality like `A = B`, then `rw [some_lemma]` will try to find `A` in your goal and replace it with `B`.
- If `some_lemma` is `A = B`, then `rw [\leftarrow some_lemma]` (or `rw [Eq.symm some_lemma]`) will try to find `B` in your goal and replace it with `A` (it rewrites from right to left).

Solving `0 + n = n`

For natural numbers, there's a standard theorem that states exactly what you want to prove: `0 + n = n`. In Lean's `Mathlib`, this theorem is often called `Nat.zero_add`.

So, if we have access to `Nat.zero_add` (which `import Mathlib.Data.Nat.Basic` should provide), we can use it with `rw`.

Let's try this in your `zero_add_example`:

Listing 3: Using the rw Tactic

```
import Mathlib.Data.Nat.Basic

theorem zero_add_example (n : Nat) : 0 + n = n := by
  rw [Nat.zero_add]
```

What to expect

1. Your goal starts as $\vdash 0 + n = n$.
2. The lemma `Nat.zero_add` states `∀ (n : Nat), 0 + n = n`.
3. When you use `rw [Nat.zero_add]`, Lean sees the `0 + n` on the left side of your goal. It matches this with the left side of `Nat.zero_add`.
4. It then replaces `0 + n` in your goal with the right side of `Nat.zero_add`, which is `n`.
5. Your goal becomes $\vdash n = n$.

In fact, since `Nat.zero_add` is *exactly* the statement `0 + n = n`, applying `rw [Nat.zero_add]` should solve the goal directly! The Infoview should show "goals accomplished ".

Try it out!

Replace your `rfl` that failed with `rw [Nat.zero_add]`. What does the Infoview say?

This experience of `rfl` failing and then using `rw` with a known lemma is *very* common in Lean. It teaches you to distinguish between what's true by definition and what's true because of a theorem.

3.5. Practice with `rw` (and maybe `rfl`!)

Let's try a slightly more involved proof that might require you to think about how a rewrite changes the goal. Consider the associativity of addition for natural numbers:

$$(a + b) + c = a + (b + c)$$

In Lean's `Mathlib`, the lemma for this is `Nat.add_assoc`. It states:

```
Nat.add_assoc (a b c : Nat) : (a + b) + c = a + (b + c)
```

Your Task

Prove the following theorem:

Listing 4: Associativity of Addition Practice

```
import Mathlib.Data.Nat.Basic -- Or your current import setup

theorem add_assoc_practice (x y z : Nat) : (x + y) + z = x + (y + z) := by
  -- Your proof tactic(s) go here!
```

Hint:

- Your goal is `(x + y) + z = x + (y + z)`.
- The lemma `Nat.add_assoc` directly states this property.

What tactic (or tactics) do you think you'll need here? Give it a try and see what happens in the Infoview!