Day - 5 : Overview of Common Tactics

Touseef Haider

May 16, 2025

Study Mathematics In Lean(MIL) Section 2.3

5.1. The `intro` Tactic: Introducing Hypotheses

The `intro` tactic (or `intros` for multiple introductions) is your go-to tool when your goal is an *implication* (like `P → Q`) or a *universally quantified statement* (like `∀ x : T, P x`).

For Implication `P → Q`

If your goal is $\vdash P \to Q$ (read as "prove that P implies Q"), it means you need to show that *if* you assume `P` is true, then you can prove `Q`. The `intro hP` tactic does exactly this:

1. It takes the `P` from the left side of the '→'.
2. It adds `hP : P` to your list of hypotheses (your local context). So now you have `P` as an assumption named `hP`.
3. It changes your goal to `⊢ Q`. Now you need to prove `Q`, but you have the extra help of assuming `P` (via `hP`).

Example:

Suppose your Lean state is:

```
1 goal
P Q : Prop
⊢ P → Q
```

If you apply the tactic `intro hP`, the state becomes:

```
1 goal
P Q : Prop
hP : P
⊢ Q
```

You've"introduced" the hypothesis `P` into your context.

For Universal Quantifiers (`∀ x : T, P x`):

If your goal is `⊢ ∀ x : T, P x` (read as "prove that for all `x` of type `T`, the property `P x` holds"), it means you need to show this for an *arbitrary* `x` of type `T`. The `intro x` tactic does this:

1. It introduces an arbitrary `x` of type `T` into your local context.
2. It changes your goal to `⊢ P x`. Now you need to prove `P x` for this arbitrary `x`.

Example:

Suppose your Lean state is:

```
1 goal
P : Nat → Prop
⊢ ∀ n : Nat, P n
```

If you apply the tactic `intro n_val`, the state becomes:

```
 1 goal
 P : Nat → Prop
 n_val : Nat
 ⊢ P n_val
```

You've introduced an arbitrary natural number `n_val` and now must prove `P` holds for it.

Why is `intro` useful?

It allows you to peel off the "if..." parts of implications or the "for all..." parts of universal statements, turning them into assumptions you can use in later steps of your proof (often with `rw` or other tactics).

Think of it like this: If someone asks you to prove "If it's raining, then the ground is wet," your natural first step is to say, "Okay, *assume* it's raining..." That's what `intro` does!

Let's see a quick example in Lean code. Consider this simple (and perhaps obvious) theorem: If `P` is true, then `P` is true.

```
theorem if_P_then_P (P : Prop) : P → P := by
  -- What tactic would you use first here, based on what we just discussed?
  -- And what would the goal become?
```

What do you think? Your initial goal is :

```
1 goal
P : Prop
⊢ P → P
```

If you apply the tactic `intro h` (where `h` is the name you give to the new hypothesis, the Infoview will change to :

```
1 goal
P : Prop
h : P
⊢ P
```

Now you need to prove `P`. Look at the current state:

- You need to prove `P`.
- You have a hypothesis `h` which is `P`.

How would you tell Lean, "Hey, the thing I need to prove is exactly this hypothesis `h` that I already have"?

5.2. The `exact` Tactic: Pointing to the Proof

The `exact` tactic is used when your current goal is exactly matched by one of your hypotheses or by a known lemma/theorem that directly states the goal.

- If you goal is `⊢ G`
- And you have a hypothesis `h : G` (meaning `h` is a proof of `G`)
- Or you know a lemma `my_lemma : G`
- Then `exact h` (or `exact my_lemma`) will solve the goal.

It's like telling Lean, " The proof you are looking for? It is this thing right here!". So for our `if_P_then_P` theorem, the complete proof is :

```
theorem if_P_then_P ( P : Prop ) : P → P := by
    intro hP -- Goal becomes ⊣ P, with hP : P in hypotheses
    exact hP -- Solves the goal using the hypothesis hP
```

You have now learned :

- `intro` : to assume the premise of an implication (or introduce an arbitrary variable for a $\forall$
- `exact` : To solve a goal when you have a hypothesis or lemma that is precisely what the goal states.

These two work together: `intro` to get hypotheses, and then other tactics (including `exact` or `rw`) to use those hypotheses to solve the new goal.

5.3. The `apply` Tactic : Matching Conclusions and Creating Subgoals

The `apply` tactic is similar to `exact`, but it is more flexible. You use `apply` when you have a hypothesis or lemma whose *conclusion* matches your current goal, but the hypothesis/lemma itself might have some prerequisites (premises).

How it works:

- You have a goal, say $\dashv$ `G`.
- You have a hypothesis or lemma, say `h :` $P_1 \to P_2 \to \cdots \to P_n \to G$. (This mean if you can prove P_1, P_2, $\cdots$, P_n, then you get `G`).
- If you use tactic `apply h`, Lean sees that the conclusion of `h` (which is `G`) matches your goal `G`.
- Lean then says, "Okay, I can use `h` to prove `G`, provided you can give me proofs for all of `h`'s premises."
- Your original goal $\dashv$ `G` is replaced by `n` new subgoals : $\dashv P_1, \quad \dashv P_2, \cdots, \quad \dashv P_n$.

Contrast with `exact`:

- `exact foo` works if `foo` is directly a proof of your goal.
- `apply foo` works if `foo` is a function/implication that results in your goal. `apply` essentially does "modus ponens" in reverse : to prove `G` using `P` $\to$ `G`, it asks you to prove `P`.

Example

Let's say you have the following state:

```
1 goal
P Q R : Prop
hPQ : P → Q
hQR : Q → R
hP : P
⊣ R
```

Your goal is `R`.

- You cannot `exact hQR` because `hQR` is `Q` $\to$ `R` not `R`.
- You cannot `exact hPQ` because `hPQ` is `P` $\to$ `Q`, not `R`.

But notice that `hQR` concludes with `R` (i.e. `Q` $\to$ `R`). So, you can `apply hQR`. If you type `apply hQR`, the state becomes :

```
1 goal
P Q R : Prop
hPQ : P → Q
hQR : Q → R
hP : P
⊢ Q -- New goal! This was the premise of hQR
```

Now your goal is `Q`.

- You can `apply hPQ` because `hPQ` concludes with `Q` (i.e. `P → Q`). If you type `apply hPQ`, the state becomes:

```
1 goal
P Q R : Prop
hPQ : P → Q
hQR : Q → R
hP : P
⊢ P -- New goal! This was the premise of hPQ
```

Now your goal is `P`. And you have `hP : P`. So, how would you finish this proof? This chaining (`apply` . . .then `apply` . . . then often `exact`) is very common for proving implications.

Let's try to write out the full proof for the state above. Suppose the theorem was stated like this:

```
example (P Q R : Prop) (hPQ : P → Q) (hQR : Q → R) (hP : P) : R := by
  -- How would you prove this step-by-step using apply and then the final tactic?
```

```
example (P Q R : Prop) (hPQ : P → Q) (hQR : Q → R) (hP : P) : R := by
  -- Initial state:
  -- P Q R : Prop
  -- hPQ : P → Q
  -- hQR : Q → R
  -- hP : P
  -- ⊢ R

  apply hQR  -- The conclusion of hQR (Q → R) is R, matching the goal.
             -- New goal becomes the premise of hQR.
  -- New state:
  -- P Q R : Prop
  -- hPQ : P → Q
  -- hQR : Q → R
  -- hP : P
  -- ⊢ Q

  apply hPQ  -- The conclusion of hPQ (P → Q) is Q, matching the current goal.
             -- New goal becomes the premise of hPQ.
  -- New state:
  -- P Q R : Prop
  -- hPQ : P → Q
  -- hQR : Q → R
  -- hP : P
  -- ⊢ P

  exact hP   -- The hypothesis hP directly proves the current goal P.
  -- Proof complete! Infoview shows ‘‘goals accomplished"
```

You have beautifully demonstrated how `apply` works backward from the goal, using implications, and how `exact` can then seal the deal when a premise is directly available as a hypothesis. This pattern of `apply`ing implications and then proving the new sub goals (perhaps with more `apply`, `rw`, `exact`, or `intro`s) is a cornerstone of proving in Lean.

5.4. The `simp` Tactic : The Simplifier

The `simp` tactic is one of Lean's most powerful and frequently used automation tools. Think of it as a skilled assistant that tries to simplify your goal (or hypotheses) using a predefined set of rewrite rules, definitions, and basic computational steps.

What it does:

- `simp` takes your current goal.
- It repeatedly applies lemmas that are tagged with the `@[simp]` attribute (these are called "simp lemmas"). These lemmas are equalities that are generally considered "simplifying" like `n + 0 = n`, `0 + n = n`, `¬(¬P) ⟺ P`, etc.
- It unfolds definitions that are marked as `@[reducible]` or sometimes just basic definitions.
- It performs computations (like `2 + 2` becomes `4`).
- It continues this process until no more such simplifications can be applied.

How to use it:

- `simp`: Tries to simplify the current goal.
- `simp at h` : Tries to simplify hypothesis `h`.
- `simp at *` : Tries to simplify all hypotheses and the goal.
- `simp [lemma1, lemma2]` : Simplifies using the default simp set plus `lemma1` and `lemma2` (even if they are not `@[simp]` lemmas).
- `simp only [lemma1, lemma2]` : Simplifies using only `lemma1` and `lemma2` (and not the default simp set). This gives you more control.
- `simp [-lemma_to_remove]` : Simplifies using the default simp set but *excludes* `lemma_to_remove`.

When is it useful?

- Cleaning up expressions: If your goal has things like `x+0` or `true && P`. `simp` can often clean these up to `x` or `P`.
- Unfolding definitions: if a definition is obscuring the structure of your goal, `simp` might unfold it to reveal something easier to work with.
- Applying many basic rewrites : Instead of `rw [lemma1], rw [lemma2], rw [lemma3]`, sometimes a single `simp` can do the job if those lemmas are part of its simp set.
- After `intro` or `cases` (another tactic we haven't covered yet, for breaking down inductive types), the goal might become messy, and `simp` can tidy it up.

Example

Suppose your goal is `⊣ ( x+ 0) + (y * 1) = x + y`. Lemmas like `Nat.add_zero (n : Nat) : n + 0 = n` and `Nat.mul_one (n: Nat) : n * 1 = n` are typically `@[simp]` lemmas.

If you apply `simp`, Lean would:

- See `x + 0` and rewrite it to `x` (using `Nat.add_zero`). Goal becomes `x + ( y * 1 ) = x + y`.
- See `y + 1` and rewrite it to `y` (using `Nat.mul_one`). Goal becomes `x + y = x + y`
- This is now `rfl`, and `simp` is often smart enough to solve such goals by reflexivity it it reaches that state. So, it might directly show "goal accomplished".

A Word of Caution (and Power!):

`simp` is powerful, but it can sometimes feel like a "black box". It might do a lot of steps, and if it does not solve the goal, the new goal might look quite different, and it might not be obvious how it got there. Let's try a small example theorem where `simp` can be very helpful :

```
import Mathlib.Data.Nat.Basic -- For Nat properties


theorem simp_example (P : Prop) (n m : Nat) (h : P) : (true  P) ∧ (n + 0 = m) → m = n := by
  -- What do you think simp will do to the goal if we apply it after an intro?
```

If you were to first use `intro h_imp` for the main implication, your goal would become `m = n` and you would have a hypothesis `h_imp : (true ∧ P ) ∧ (n + 0 = m)`. What do you think `simp at h_imp` would do to this hypothesis `h_imp`? Think about `true ∧ P` and `n + 0`.

For the hypothesis `h_imp : ( true ∧ P) ∧ ( n + 0 = m )` :

- `simp` would look at `true ∧ P`. Since `true ∧ P` is equivalent to `P` (often via simp lemma like `true_and P : true ∧ P ⟺ P`), it simplifies this part to `P`.
- `simp` would look at `n + 0 = m`. Since `n + 0` simplifies to `n` (often via a simp lemma like `Nat.add_zero n : n + 0 = n`), this part of the equality becomes `n=m`.

So, after `simp at h_imp`, the hypothesis `h_imp` would effectively become `P ∧ (n = m )` .

Let's see how we could complete the proof of the `simp_example` theorem:

```
import Mathlib.Data.Nat.Basic -- For Nat properties

theorem simp_example (P : Prop) (n m : Nat) (_h_given_P : P) : (true ∧ P) ∧ (n + 0 = m) → m
    = n := by
  intro h_imp          -- h_imp : (true ∧ P) ∧ (n + 0 = m)
                       -- Goal: ⊣ m = n

  -- Simplify the hypothesis h_imp
  simp at h_imp --
                       -- After this, h_imp effectively means P is true AND n = m.
                       -- In Lean 4, h_imp will likely become: h_imp : P ∧ n = m

  -- Now h_imp is ‘P ∧ (n = m)‘. We need to use the ‘n = m‘ part.
  -- We can extract the parts of the conjunction.
  -- A common way is to use the ‘cases‘ tactic or dot notation if available.
  -- Let’s assume h_imp is now ‘P ∧ n = m‘. We can get ‘n = m‘ from it.
  -- If h_imp is h_imp : P ∧ n = m
  -- We can get ‘h_eq : n = m‘ from ‘h_imp.right‘ (or ‘h_imp.2‘ in some versions)
  -- or by using ‘cases h_imp with hP_internal h_eq_actual‘
  cases h_imp with hp_from_h_imp h_eq_from_h_imp
                       -- Now we have:
                       -- hp_from_h_imp : P
                       -- h_eq_from_h_imp : n = m

  -- Our goal is ⊣ m = n. We have h_eq_from_h_imp : n = m.
  -- We need to show m = n. This is the *symmetric* of n = m.
  -- We can rewrite or use ‘Eq.symm‘.
  rw [h_eq_from_h_imp] -- This rewrites n to m in the goal, making it m = m
                       -- Or, if it rewrites m to n, it becomes n = n.
                       -- Let’s assume it rewrites n in the goal if possible, or m.
                       -- If goal is m = n and h_eq_from_h_imp is n = m.
                       -- ‘rw [h_eq_from_h_imp]‘ might try to replace n with m in the goal.
                       -- This would make goal ‘m = m‘.
                       -- Or, ‘rw [← h_eq_from_h_imp]‘ would replace m with n, making it ‘n
    = n‘.
  -- A more direct way if you have ‘h_eq_from_h_imp : n = m‘ to prove ‘m = n‘ is:
  exact Eq.symm h_eq_from_h_imp -- ‘Eq.symm‘ turns ‘n=m‘ into ‘m=n‘

  -- So, a cleaner finish after ‘simp at h_imp‘ which results in ‘h_imp : P ∧ n = m‘:
  -- exact (h_imp.right).symm -- Assuming .right gives n=m

```

```
  -- Let's try the cases approach which is common:
  -- (after `simp at h_imp` resulted in `h_imp : P ∧ n = m`)
  -- cases h_imp with unused_P_part actual_equality -- `actual_equality` is `n = m`
  -- exact actual_equality.symm -- This uses the .symm property of an equality.
```

A very common and often more robust way in Lean 4 after `simp at h_imp` (if `h_imp` becomes `P ∧ n = m`) is:

```
theorem simp_example_lean4 (P : Prop) (n m : Nat) (_hP_given : P) : (true ∧ P) ∧ (n + 0 = m) → m = n := by
  intro h_imp
  simp (config := { contextual := true }) [Nat.add_zero, and_true] at h_imp -- contextual simp might be more powerful
  -- Assuming h_imp is now P ∧ n = m
  -- We can use dot notation to access parts of a conjunction or an equality:
  rw [h_imp.right] -- If h_imp.right is n = m, this will rewrite the goal m = n to m = m if n is found, or n=n if m is found.
  rfl               -- Or more directly, if h_imp.right is n=m
  -- exact h_imp.right.symm -- This should work

  -- Let's assume `simp at h_imp` simplifies it effectively to give us `n = m`.
  -- One way `simp` can be very powerful is using `simp [*] at *`.
  -- This uses all hypotheses to simplify each other and the goal.
  -- Consider this alternative approach using that:
```

Let's show a more direct proof using `simp`'s power:

```
import Mathlib.Data.Nat.Basic


theorem simp_example_direct (P : Prop) (n m : Nat) (_h_given_P : P) : (true ∧ P) ∧ (n + 0 = m) → m = n := by
  intro h_imp         -- h_imp : (true ∧ P) ∧ (n + 0 = m)
                      -- Goal: ⊢ m = n
  simp at h_imp -- h_imp becomes P ∧ n = m
  exact h_imp.right.symm -- h_imp.right is n = m, so h_imp.right.symm is m = n
```

This version is clean and shows `simp` simplifying the hypothesis, and then we use the parts of the simplified hypothesis. The `.right` gets the second part of the∧ (which is `n = m`), and `.symm` gives the symmetric equality `m=n`, which `exact`ly matches the goal.

The key takeaway for `simp` is that it is a powerful tool for automatically applying many known simplifying rewrite rules.

Now you have `rfl, rw, intro, exact, apply`, and `simp`. That is a very solid set of basic tactics.

5.5. When to Use Which Tactic : A Quick Summary

`rfl`

- When : Your goal is an equality `A=B` (or an equivalence `P ⟺ Q`) where `A` and `B` (or `P` and `Q`) are *definitionally* equal. Lean can see they are the same just by unfolding definitions.
- Example : ⊢ `x + 0 = x` (if `add_zero` is definitional), ⊢ `2 + 2 = 4`.

`rw [lemma_or_hyp]`

- When : Your goal contains a subexpression that matches one side of an equality stated in `lemma_or_hyp`, and you want to replace it with the other side.
- Use `rw [← lemma_or_hyp]` to rewrite from right-to-left.
- Example: Goal ⊢ `(a + b) +c = (b + a) +c`, use `rw [Nat.add_comm a b]` to change `a+b` to `b+a`.

`intro h` **(or** `intro h1 h2 ...`**):**

- When : Your goal is an implication `P` $\rightarrow$ `Q` or a universal qualification $\forall$ `x : T, P x`.
- Effect: Adds `P` (as `h : P`) or `x : T` to your hypotheses and changes the goal to `Q` or `P x`.
- Example: Goal $\vdash$ `P` $\rightarrow$ `(Q` $\rightarrow$ `R)` , `intros hP hQ` gives hypotheses `hP : P`, `hQ : Q` and goal $\vdash$ `R`.

`exact h` **(or** `exact lemma_name`**):**

- When : Your current goal $\vdash$ `G` is *exactly* what hypothesis `h` states (i.e. , `h : G`) or what `lemma_name` states.
- Example : Goal $\vdash$ `P`, hypothesis `hP ; P`. Use `exact hP`.

`apply h`**(or** `apply lemma_name`**):**

- When : The conclusion of hypothesis `h` (or `lemma_name`) matches your goal, but `h` itself might have premises.
- Effect : Solves the current goal using `h` and creates new subgoals for each premise of `h`.
- Example : Goal $\vdash$ `R`, hypothesis `hQR : Q` $\rightarrow$ `R`. Use `apply hQR`. New goal becomes $\vdash$ `Q`.

`simp`

- When : You want to simplify the goal or hypotheses using a collection of known rewrite rules (like `x + 0 = x`, `true` $\wedge$ `P` $\iff$ `P`, etc.) and definitions.
- Can be used as `simp, simp at h, simp [*] at *` (uses hypotheses to simplify too).
- Example : goal $\vdash$ `(x + 0) + (y * 1) = z`, `simp` might change it to $\vdash$ `x + y =z`.

This summary should give a good starting point for choosing your tactics. Often, a proof involves a sequence of these.