Introduzione alla formalizzazione della matematica in Lean
Lesson 3/10 Tactics

Yoh Tanimoto

Corso di dottorato, University of Rome “Tor Vergata”

17 Dicembre 2025

Tactic: apply, exact

We saw a proof using a tactic “apply” and “exact”.

```
example (p q r : Prop) (hp : p) (hpq : p → q)
    (hqr : q → r) : r := by
  apply hqr
  apply hpq
  exact hp
```

link

- “by” introduces the tactic mode. You can use all the tactics below.
- The goal is `r`.
- “apply” works if we have an assumption or theorem with the conclusion `r`.
- We “`apply hqr`”. The goal becomes `q`.
- We “`apply hpq`”. The goal becomes `p`.
- We have the assumption `hp : p`. So it is `exact hp`.

Tactic: show_term

This is a "backward proof". The final goal is `r`, and we went backwards to the assumption `hp : p` by using other assumptions.

```
example (p q r : Prop) (hp : p) (hpq : p → q)
    (hqr : q → r) : r := by show_term
  apply hqr
  apply hpq
  exact hp
```

link

- By writing `show_term`, one can see what the tactic mode produces.
- In this example, it is `hqr (hpq hp)`.
- One can add tactics, but they only produce these **proof terms**.
- The kernel of Lean verifies whether the proof term gives the correct type, so the kernel remains a small program.
- One can write independent type checker when s/he does not trust the kernel.

Tactic: have

The tactic `have` can introduce an intermediate proposition to perform a "forward proof".

```
example (p q r : Prop) (hp : p) (hpq : p → q)
    (hqr : q → r) : r := by show_term
  have hq : q := by
    exact hpq hp
  exact hqr hq
```

link

- The line `have` defines a proof of `q`.
- The line `have` behaves exactly as in the main proof, except that it shold be indented. One can use `by` and all the tactics.
- The proof of `q` produced here is named `hq` and we can use it in the remaining proof.

Tactic: intro

The tactic `intro` can add an assumption and transform the goal

```
example (p q : Prop) : p → q → p := by show_term
  intro hp
  intro hq
  exact hp
```

link

- The line `p → q → p` should be read as `p → (q → p)`.
- The line `intro hp` puts a proof `hp` of `p` among the assumptions.
- The remaining goal is now `q → p`.
- The line `intro hq` puts a proof `hq` of `q` among the assumptions.
- The remaining goal is now `q → p`. `exact hp` closes the goal.

Tactic: constructor

When the goal is `p ∧ q`, we can prove it separately using `constructor`.

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

link

- In Lean, `p ∧ q` is defined as a `structure`, which contains two propositions `left` and `right`.
- `constructor` splits the construction of a structured object into components.
- There are many other things that are `structure`.

```
example (p q : Prop) (hpq : p ∧ q) : p := by exact
hpq.left
```

Tactic: left/right

When the goal is `p ∨ q`, we can prove it from one of them using `left` or `right`.

```
example (p q : Prop) (hp : p) : p ∨ q := by
  left
  · exact hp
```

link

- In Lean, `p ∨ q` is defined as an inductive type, similar to that of `Nat`: there are introduction rules, and there is no other way to construct.

Tactic: rcases

The tactic `rcases` can split cases in the assumption `p ∨ q`.

```
example (p q r : Prop) (hpq : p ∨ q)
    (hpr : p → r) (hqr : q → r) : r := by
  rcases hpq with hp | hq
  · exact hpr hp
  · exact hqr hq
```

link

- Recall that `p ∨ q` is an inductive type with elements `inl` and `inr`.
- The line `rcases hpq` declares that we split the cases, either `p` (coming from `inl`) or `q` (coming from `inr`).
- Their names `hp` or `hq` follow `with`.
- The rest is proven using these new assumptions. We can also use `cases` with slightly different grammer.

Tactic: refine

The tactic `refine` allows one to leave some holes and fill in them later.

```
example (p q r : Prop) (hpq : p → q → r)
    (hp : p) (hq : q) : r := by
  refine hpq ?_ ?_
  · exact hp
  · exact hq
```

link

- This could be proven by `exact hpq hp hq`, but in practice `hp` and `hq` can be long.

Tactic: refine

The tactic `refine` allows one to leave some holes and fill in them later.

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) :
    p ∧ q ∧ r := by
  refine <?_, ?_, ?_>
  · exact hp
  · exact hq
  · exact hr
```

link

- This could be proven by `constructor` twice, but this is shorter.

Tactic: intro/by_contra for proof by contradiction

The tactic `intro` can used to prove `¬ p`

```
example (p q : Prop) (hpq : p → q) (hnq : ¬ q) :
    ¬ p := by
  intro hp
  have hq : q := by
    exact hpq hp
  exact hnq hq
```

link

- When the goal is `¬ p`, `intro hp` switches the goal to `False` and puts `hp : p` as an assumption.
- `¬ p` is defined as `p → False`. This is why the last line works.
- `False` has no introduction rule. As long as the system is sound, it cannot be constructed except inside a proof by contradiction like this.
- If you write `import Mathlib` in the first line, you can use `by_contra` instead of `intro`.

Tactic: rfl

The tactic `rfl` closes the goal if it is an equality of the form `a = a`.

```
example (n : Nat) : 37 + n = 37 + n := by show_term
  rfl
```

link

- Actually, Lean unfolds the definition of some complicated terms to some extent. Therefore, if the sides of the (desired) equality are *definitionally equal*, then `rfl` can close the goal.

```
example : 2 + 2 = 4 := by show_term
  rfl
```

link

- This works because, *by definition*, `2 = (0 + 1) + 1`, `4 = (((0 + 1) + 1) + 1) + 1)`, while `2 + 2` is defined inductively as `2 + 2 = (2 + 1) + 1`.

Tactic: rw

The tactic `rw` substitutes (*rewrites*) a certain expression in the goal if we have an equality in the hypothesis.

```
example (n : Nat) (hn : n = 3) : 1 + n = 4 := by
show_term
  rw [hn]
```

link

- It is also possible to rewrite an expression in other hypotheses.

```
example (n m : Nat) (hn : n = 3) (hm : m = n + 2) :
    1 + m = 6 := by show_term
  rw [hn] at hm
  rw [hm]
```

link

Tactic: congr

The tactic `congr` reduces the proof of equality `f x = f y` to `x = y`.
Note that `f` can be a function or a predicate.

```
example (n : Nat) (f : Nat → Nat) :
    f ((n + 1) ^ 2) = f (n ^ 2 + 2 * n + 1) := by show_term
  congr
  ring
```

link (`ring` and `gcongr` below requires `import Mathlib`)

- There is a much more powerful `gcongr`.

```
example {a b x c d : ℝ} (h : 0 ≤ x)(h1 : a + 1 ≤ b + 1)
  (h2 : c + 2 ≤ d + 2) : x * a + c ≤ x * b + d := by
  show_term gcongr
  · linarith
  · linarith
```

link

Tactic: calc

The tactic `calc` can prove a goal that can be proved by a sequence of relations satisfying transitivity, by justifying each passage.

```
example (n m :  Nat) (hn :  n = 3) (hm :  m = n + 2) :
    1 + m = 6 := by show_term
  calc
    1 + m = 1 + (n + 2) := by rw [hm]
    _ = 1 + (3 + 2) := by rw [hn]
    _ = 6 := by rfl
```

link

- `calc` block can be used not only for equalities but also for inequalities and inclusion of sets.

Tactic: intro/(specialization)

When the goal is of the form `∀ x, P x`, `intro x` gives a variable `x` and transforms the goal into `P x`.

```
example : ∀ (n : Nat), n + 0 = n := by
  intro n
  rfl
```

link

When a hypothesis is of the form `hP : ∀ x, P x`, for any term `z`, we have `hP z : P z`.

```
example (n : Nat) (hn : ∀ m, n + m = m) : n = 0 := by
  have hzero : n + 0 = 0 := by
    exact hn 0
  exact hzero
```

link

Tactic: use/obtain

- When the goal is of the form `∃ x, P x`, where `x` is a term of the correct type, `use x` transforms the goal into `P x`.
- When a hypothesis is of the form `h : ∃ x, P x`, `obtain` gives an object (say `x`) and a proof of `P x`.

```
import Mathlib
example (A B : Set Nat) (hAB : ∃ n, n ∈ A ∩ B) :
    ∃ (n : Nat), n ∈ A ∧ n ∈ B := by
  obtain ⟨n, hn⟩ := hAB
  use n
  constructor
    · exact hn.left
    · exact hn.right
```

(need `import Mathlib` for `use`) link

Using (already proven) theorems

Theorems are given names and can be used later.

```
theorem MyNat.add_zero (n : Nat) : n + 0 = n := by rfl
variable (m : Nat)
#check MyNat.add_zero
#check MyNat.add_zero 1
#check MyNat.add_zero m
#check MyNat.add_zero (m + 0)
example (n : Nat) : (n + 0) + 0 = n := by
  rw [MyNat.add_zero (n + 0), MyNat.add_zero n]
```

link

- The addition $+$ is defined inductively, and the case 0 is definition, thus `n + 0 = n` is proven by `rfl`. We call it `MyNat.add_zero`.
- Under `#check`, you can see of what proof it is.
- It takes a variable. Every time you give a variable, say `m`, `MyNat.add_zero m` is a proof of an equation `m + 0 = 0`.

Practice

- Let us play Natural number game again.

- In the next lesson, we follow Mathematics in Lean.
- Make sure to install Lean, Git and to `clone` the repository (follow this link) (or use a desktop app).