Day - 7 : Existential Quantifier

Touseef Haider

Study Mathematics In Lean(MIL) Section 3.2

7.1. Existential Quantifier ∃ :

The existential quantifier, written as ∃, is used to assert that *there exists at least one* element of a certain type that satisfies a given property.

A statement like ∃ `x : T, P x` reads:
"There exists an `x` of type `T` such that the property `P x` holds true."

The key difference from the universal quantifier ∀ is that ∀ `x : T, P x` claims `P x` is true for *all* `x` in `T`, while ∃ `x : T, P x` only claims that it is true for *at least one* `x` in `T`. It does not have to be true for all of them, and it doesnot specify which one(s) satisfy the property, just that such an element exists.

For example:

- ∃ `n : Nat, n > 5` means "There exists a natural number `n` such that `n` is greater than 5."
- ∃ `s : String, s.length` > 10 means "There exists a string `s` such that its length is greater than 10."
- ∃ `x : Nat, x + 1 = 0` means "There exists a natural number `x` such that `x+1=0`. (This is false for natural numbers).

There are two main scenarios: proving an existential statement and using an existential statement that is already a hypothesis.

Proving an Existential Statement :

To prove ∃ `x : T, P x`, you need to do two things:

The Lean4 tactic for this is `exists`. If your goal is ∃ `x : T, P x`, you can type: `exists x_val`. This will:

- Substitute `x_val` for `x` in the property `P x`.
- Change your goal to proving `P x_val`.

You then proceed to prove this new goal.

Example

Let's prove ∃ `n : Nat, n` > 5 .

```
example : ∃ n : Nat, n > 5 := by
  exists 6  -- We provide the witness '6'
```

Here `exists 6` tells Lean, "I claim that 6 is the natural number that satisfies the property.". The goal then becomes `6 > 5`, which is true.

7.2. Using an Existential Hypothesis

When you have `h : ∃ x : T, P x`, you know some `x` exists satisfying `P x`. You need to "open it up" to access that `x` and the proof of `P x`.

Using `rcases`

The `rcases` tactic (recursive cases) is very convenient here. If you have `h : ∃ x : T, P x`, you can write: `rcases h with ⟨x_name, h_property⟩`.

This will:

- Introduce a new variable `x_name : T` (the witness).
- Introduce a new hypothesis `h_property : P x_name` (the proof that the witness satisfies the property).
- The original hypothesis `h` is typically cleared.

Using `cases`

The syntax, `cases h_exists with | intro n h_n_gt_5 => ...proof...`, is a way to use `cases`. The `intro` here is used to name the components destructured from the existential. A more direct way to write this specific pattern for `Exists` (whose constructor is `Exists.mk`) would often be: `cases h_exists with | mk n h_n_gt_5 => ...proof...` or using the anonymous constructor notation: `cases h_exists with | ⟨n, h_n_gt_5⟩ => ...proof...`

Syntax `| intro n h_n_gt_5 =>` effectively does the same: it introduces `n` as the witness and `h_n_gt_5` as the property for that witness within the scope of that case.

Example:

Suppose we know `h_exists : ∃ n : Nat, n > 5` and `h_universal : ∀ k : Nat, k > 5 → k^2 > 25`. We want to prove `∃ m : Nat, m^2 > 25`.

Using `rcases`:

```
example (h_exists : ∃ n : Nat, n > 5)
        (h_universal : ∀ k : Nat, k > 5 → k^2 > 25) :
        ∃ m : Nat, m^2 > 25 := by
  -- Use the existential hypothesis h_exists
  rcases h_exists with ⟨ n, h_n_gt_5 ⟩
  -- Now we have in our context:
  -- n : Nat
  -- h_n_gt_5 : n > 5

  -- We want to prove ∃ m : Nat, m^2 > 25.
  -- Let's use 'n' (our destructured witness) as our witness for 'm'.
  exists n
  -- New goal: n^2 > 25

  -- We can use h_universal.
  exact h_universal n h_n_gt_5
```

Using `cases` :

```
example (h_exists : ∃ n : Nat, n > 5)
        (h_universal : ∀ k : Nat, k > 5 → k^2 > 25) :
        ∃ m : Nat, m^2 > 25 := by
  cases h_exists with
  | intro n h_n_gt_5 =>
    -- Context within this case:
    -- n : Nat
    -- h_n_gt_5 : n > 5

    -- We want to prove ∃ m : Nat, m^2 > 25.
    -- Let's use 'n' as our witness for 'm'.
    exists n
    -- New goal: n^2 > 25

    -- Use h_universal
    let h_prop_for_n := h_universal n -- h_prop_for_n : n > 5 → n^2 > 25
    exact h_prop_for_n h_n_gt_5
```

The expression `h_prop_for_n h_n_gt_5` is function application. The implication `h_prop_for_n` acts like a function that takes a proof of its premise (`n > 5`) and returns a proof of its conclusion (n^2 `> 25`).

Both `rcases` and this `cases` syntax achieve the same goal of deconstructing the existential. `rcases` is often favored for its brevity when dealing with nested existentials or conjunctions, as it can deconstruct them in one go.

7.3. Simple Lean4 Examples Demonstrating Existential Quantification:

Theorem to Prove:

If we have two properties, `P` and `Q`, defined over some type α, and we know that for any element `x` of α, if `P x` is true then `Q x` is true. Then, if there exists an element for which `P` is true, there must also exist an element for which `Q` is true.

```
import Std.Tactic -- rcases is often in Std, but good to note imports

example (α : Type) (P Q : α → Prop)
        (h_all_P_implies_Q : ∀ x : α, P x → Q x)
        (h_exists_Px : ∃ x : α, P x) :
        ∃ x : α, Q x := by
  -- We have h_exists_Px : ∃ x, P x.
  -- This tells us there is some element that satisfies P. Let's get it.
  -- The `rcases` tactic will extract the witness (let's call it 'w')
  -- and the proof that w satisfies P (let's call it h_Pw).
  rcases h_exists_Px with ⟨ w, h_Pw ⟩
  -- Now in our context we have:
  -- w : α
  -- h_Pw : P w

  -- Our goal is to prove ∃ x : α, Q x.
  -- We need to provide a witness for x. A good candidate is 'w'.
  -- Let's claim that 'w' is the element that satisfies Q.
  exists w
  -- The new goal is now to prove: Q w

  -- Now we need to show Q w.
  -- We have h_all_P_implies_Q : ∀ x : α, P x → Q x.
  -- This means for any x, if P x holds, then Q x holds.
  -- Let's apply this general rule to our specific witness 'w'.
  -- (h_all_P_implies_Q w) gives us a proof of (P w → Q w).
  let h_Pw_implies_Qw := h_all_P_implies_Q w
  -- So, h_Pw_implies_Qw : P w → Q w

  -- We also have h_Pw : P w.
  -- We know that if P w implies Q w, and P w is true, then Q w must be true.
  -- We can use 'exact' with the application of h_Pw_implies_Qw to h_Pw.
  exact (h_Pw_implies_Qw h_Pw)
```

Let's trace the logic and tactics:

1. `rcases h_exists_Px with ⟨ w, h_Pw ⟩`:
   - We start with `h_exists_Px : ∃ x : α, P x`. This hypothesis asserts the existence of an element satisfying `P`.
   - `rcases` deconstructs this. It introduces:
     - `w : α` : This is the specific witness whose existence was asserted by `h_exists_Px`.
     - `h_Pw : P w`: This is the proof that our witness `w` indeed satisfies the property `P`.
2. `exists w`:
   - Our goal is `∃ x : α , Q x`. We need to provide a witness for `x` such that `Q x` holds.
   - The most natural candidate for this witness is the `w` we just obtained from `h_exists_Px`.
   - The `exists w` tactic states our claim: " Yes, such an `x` exists, and it is `w`."

- This changes the goal to proving `Q w`. We now need to show that our chosen witness `w` actually satisfies `Q`.

3. `let h_Pw_implies_Qw := h_all_P_implies_Q w`:
   - We have the universal hypothesis `h_all_P_implies_Q : ∀ x : α, P x → Q x.`
   - Since this holds for *all* `x`, it must hold for our specific `w`.
   - Applying `h_all_P_implies_Q` to `w` (i.e. `h_all_P_implies_Q w`) gives us specific implication for `w`: `P w → Q w`.
   - We name this specific implication `h_Pw_implies_Qw`.
4. `exact (h_Pw_implies_Qw h_Pw)`:
   - Our goal is `Q w`.
   - We have `h_Pw_implies_Qw : P w → Q w`.
   - We also have `h_Pw : P w`.
   - This is exactly the setup for function application. Applying the function `h_Pw_implies_Qw` to the argument `h_Pw` yields a proof of `Q w`.
   - `exact` takes this proof of `Q w` and closes the goal.

This example demonstrates the workflow:

- Using an existential (`rcases`).
- Proving an existential (`exists`).
- Leaveraging universal quantifiers and implications along the way.