Day - 10 : Disjunction and Proof by Cases

Touseef Haider

Study Mathematics In Lean(MIL) Section 3.5

Logical Meaning :

The statement `P ∨ Q` is true if at least one of the proposition `P` or `Q` is true.

- If `P` is true, then `P ∨ Q` is true.
- If `Q` is true, then `P ∨ Q` is true.
- If both `P` and `Q` are true, then `P ∨ Q` is still true.

`Or` in Lean4

In Lean, `Or`, written as `P ∨ Q`, is an inductive type, it has two constructors:

- `Or.inl`(for Or-introduction-left): This constructor is used when you know `P` is true. It takes a proof of `P` and returns a proof of `P ∨ Q`.
- `Or.inr`(for Or-introduction-right): This constructor is used when you know `Q` is true. It takes a proof of `Q` and returns a proof of `P ∨ Q`.

```

variable {x y : ℝ}

example : x < |y| → x < y ∨ x < -y := by
  -- The core strategy here is to split the proof into two cases.
  -- based on whether y is non=negative or negative.
  -- le_or_gt 0 y is a standard lemma giving 0 ≤ y ∨∨ 0 > y
  -- rcases .. with h | h' takes this or statement and creates two proof goals:
  -- 1. The first goal assumes h: 0 ≤ y
  -- 2. The second goal assume h: y < 0
  rcases le_or_gt 0 y with h | h

  -- Case 1 : Assume y is non-negative h : 0 ≤ y
    -- If y ≥ 0, we know |y|=y. The abs_of_nonneg lemma states this.
    -- We use rw to replace |y| with y in our goal.

  · rw [abs_of_nonneg h]
    -- The goal x < |y| → x < y ∨ x < -y becomes x < y → x < y ∨ x < -y
    -- We use intro h to assume the premise
    -- The left tactic tells Lean we are proving the left part.
    -- The goal of the left part is x < y.
    intro h; left; exact h
  · rw [abs_of_neg h]
    intro h; right; exact h
```

Proving the same example using `cases`:

```

variable {x y : ℝ}

example : x < |y| → x < y ∨ x < -y := by
  -- We start with the le_or_gt y lemma, which gives us 0 ≤ y ∨ 0 > y.
```

```
  -- The cases tactic takes a proof of an Or
  -- and splits the proof into one goal for each constructor.
  -- For P ∨ Q, it creates two goals: one assuming P, one assuming Q.
  cases le_or_gt 0 y

  -- The case tactic allows us to explicitly select and work on one of these goals.
  -- inl refers to the left constructor of Or, which is Or.inl
  -- This case corresponds to 0 ≤ y. We name this hypothesis h.
  case inl h =>
    -- Inside this case we have h : 0 ≤ y
    -- rw [abs_of_nonneg h] rewrite |y| to y.
    rw [abs_of_nonneg h]
    intro h; left; exact h

  -- inr refers to the right constructor of Or which is Or.inr
  -- This case corresponds to 0 > y we name this hypothesis h.
  case inr h =>
    rw [abs_of_neg h]
    intro h; right; exact h
```

Proving the same example using `next`:

```

variable {x y : ℝ}

example : x < |y| → x < y ∨ x < -y := by
  -- We start with the le_or_gt y lemma, which gives us 0 ≤ y ∨ 0 > y.
  -- The cases tactic takes a proof of an Or
  -- and splits the proof into one goal for each constructor.
  -- For P ∨ Q, it creates two goals: one assuming P, one assuming Q.
  cases le_or_gt 0 y

  -- The next tactic tells Lean: Focus on the first goal in the current list
  -- and introduce its main new hypothesis naming it h.
  -- Since le_or_gt creates the 0 ≤ y case first, this focuses on that case.
  next h =>
    rw [abs_of_nonneg h]
    intro h; left; exact h
  -- After the first next block is finished, Lean automatically moves to the next goal in
    the list.

  -- This next tactic focuses on that second goal (which is the 0 > y) case.
  -- It introduces its main new hypothesis, naming it h.
  next h =>
    rw [abs_of_neg h]
    intro h; right; exact h
```

Proving the same example using `match`:

```

variable {x y : ℝ}

example : x < |y| → x < y ∨ x < -y := by
  -- We start with le_or_gt 0 y, which gives 0 ≤ y ∨ 0 > y
  -- match ... with tells Lean to inspect the structure of le_or_gt 0 y
  match le_or_gt 0 y with
    -- The | introduces a pattern to match against.
    -- Or.inl h : This pattern says ‘‘If le_or_gt o y" was constructed using Or.inl (meaning
     the left side , 0 ≤ y, holds), then bind the proof of 0 ≤ y to the name h
    -- => separates the pattern from the tactics to run for that case.
    | Or.inl h =>
    -- We are now in the case where h : 0 ≤ y
      rw [abs_of_nonneg h]
      intro h; left; exact h
    | Or.inr h =>
    -- We are now case where h : 0 > y
      rw [abs_of_neg h]
      intro h; right; exact h
```

Divisibility Example:

```
example {m n k : ℕ} (h : m ∣ n ∨ m ∣ k) : m ∣ n * k := by
  -- h is our hypothesis which states either m divides n or m divides k.
  -- The racases tactic is perfect for handling Or statement. It splits the proof into two
    cases. It also smart enough to handle the definition of divisibility at the same time.
  rcases h with ⟨ a, rfl ⟩ | ⟨ b, rfl ⟩
    -- It says deconstruct h
    -- In the first case (m ∣ n) name the witness a.
    -- Use rfl to substitute n with m*a everywhere.
    -- Same things for b.
    -- Case 1 : We assume m ∣ n.
    -- Because rcases ... ⟨ a, rfl ⟩ we know a : ℕ and every n has been replaced by m*a.
    -- OUr goal m ∣ n*k becomes m∣ m*a *k

  . rw [mul_assoc]
    -- rw [mul_assoc] changes the goal to m ∣ m * (a * k)
    apply dvd_mul_right
    -- apply dvd_mul_right uses the lemma that a number m always divides m times any other
    number (m ∣ m* x)
  . rw [mul_comm, mul_assoc]
    apply dvd_mul_right
```