Day - 9 : Conjunctions and Bi-implication

Touseef Haider

Study Mathematics In Lean(MIL) Section 3.4

Logical Conjunctions:

The statement `P ∧ Q` is true if and only if both proposition `P` and `Q` are individually true. If either `P` or `Q` or both are false, then `P ∧ Q` is false.

Lean4 Tactics for Conjunction ∧

There are two main situations: proving a conjunction and using a conjunction that's already a hypothesis.

Proving a Conjunction `P ∧ Q`:

To prove that `P ∧ Q` is true, you need to provide two things:

- A proof of `P`.
- A proof of `Q`.

Lean4 has a couple of ways to do this:

- `constructor` tactic: If your goal is `P ∧ Q`, the tactic `constructor` will split your goal into two new subgoals:
  - The first subgoal will be to prove `P`.
  - The second subgoal will be to prove `Q`. You then prove each of these in turn.

```
example (hP : P) (hQ : Q) : P ∧ Q := by
 -- We have hP : P and hQ : Q
-- We need to show P ∧ Q
-- We can use the constructor for ∧
constructor
-- The first goal is to show P
exact hP
-- The second goal is to show Q
exact hQ
```

- Anonymous Constructor `⟨ proof_of_P, proof_of_Q ⟩`: This is often the most concise way if you already have proofs for `P` and `Q`. If `hP : P` and `hQ : P`, you can directly prove `P ∧ Q` by : `exact ⟨ hP, hQ ⟩`

```
example (hP : P) (hQ : Q) : P ∧ Q := by
    exact ⟨ hP, hQ ⟩
```

Or even not in tactic mode : `def my_and_proof (hP : P) (hQ : Q) : P ∧ Q := ⟨ hP, hQ ⟩`.

9.1.2. Using a Conjunction Hypothesis

If you have a hypothesis `h : P ∧ Q`, you know both `P` and `Q` are true. You can extract these individual proofs.

```
example (h_and : P ∧ Q) : P := by
  -- We have h_and : P ∧ Q
  -- We need to show P
  -- We can use the constructor for ∧
  let hP := h_and.left
  exact hP
```

We can use `rcases` to breakdown `h: P ∧ Q` into its constituent parts and name them.

```
example (h_and : P ∧ Q) : Q ∧ P := by
  rcases h_and with ⟨ h_P_proof, h_Q_proof ⟩
  -- Now we have:
  -- h_P_proof : P
  -- h_Q_proof : Q
  -- We need to show P
  exact ⟨ h_Q_proof, h_P_proof ⟩
```

An overview of tactics:

1. `rintro` : A recursive intro, it can introduce multiple hypotheses and, if those hypotheses have a structure (like being a conjunction or an existential), it can deconstruct them at the same time.
2. `constructor`: This tactic applies the first suitable constructor for the current goal.
   - For a goal `P ∧ Q`, the constructor will change the goal into two subgoals : `P` and `Q`.
   - For a goal `P ⟺ Q`, constructor will change the goal into two subgoals : `P → Q` and `Q → P`.
3. Tactic Combinator `<;>` : `tac1 <;> tac2` runs `tac1`, and then applies `tac2` to each subgoal generated by `tac1`.
4. `.assumption` : This tactic checks if the current goal can be directly solved by one of the existing hypotheses.

```
-- Assume P and Q are propositions we have proofs for
variable (P Q : Prop) (hP : P) (hQ : Q)

example : P ∧ Q := by
  -- We have hP : P and hQ : Q
  -- We need to show P ∧ Q
  -- We can use the constructor for ∧
  constructor
  -- The first goal is to show P
  assumption
  -- The second goal is to show Q
  assumption

-- Using the <;> combinator:
example : P ∧ Q := by
  constructor <;> assumption -- Applies constructor, then assumption to each new goal
```

```
variable (P Q : Prop) (hP_implies_Q : P → Q)(hQ_implies_P : Q → P)

example : P ⟺ Q := by
  constructor
  -- We need to show P → Q
  intro hP
  exact hP_implies_Q hP
  -- We need to show Q → P
  intro hQ
  exact hQ_implies_P hQ
```

Using `fun` lambda expressions

Instead of using tactics step-by-step, you can provide a direct proof term using lambda expressions for functions (implications) and constructors for types like ∧ and ⟺ .

- For `P → Q`, the term is `fun (hP : P) ⟹ proof_of_Q_using_hP`.

```

example (p_implies_q : P → Q) (hP : P) : Q := by
  exact p_implies_q hP
  example (p_implies_q : P → Q) (hP : P) : Q :=
    (fun (p_implies_q : P → Q) (hP : P) => p_implies_q hP) p_implies_q hP
```

What is `contrapose!`?

The tactic `contrapose!` is based on the logical equivalence between an implication and its contrapositive: `P → Q` is logically equivalent to `¬ Q → ¬ P`.

- `contrapose`: When your goal is `P → Q`, applying `contrapose` changes the goal to `¬ Q → ¬ PP`.
- `contrapose!` (with the exclamation mark): It is more powerful version that does `contrapose` and then applies `intro` to the new hypothesis (`¬ Q`). So if your goal is `P → Q`, `contrapose! h_nq` will add `h_nq : ¬ Q` to your hypothese and change your goal to `¬ P`.

This tactic is not in core Lean, you usually need to import from `mathlib`.

Example 1:

Proving `(¬ Q → ¬ P ) → (P → Q)`

```
import Mathlib.Tactic.Contrapose

variable (P Q : Prop)

example (h_contrapositive : ¬ Q → ¬ P) : P → Q := by
  -- Our goal is P → Q.
  -- This seems like a perfect place to try contraposition.
  -- We want to assume ¬ Q and prove ¬ P.
  contrapose!
  -- Lean applies contrapose and intro.
  -- Our context now includes:
  -- h_contrapositive : ¬ Q → ¬ P
  exact h_contrapositive hnQ
```

Example 2:

If you are proving `P ⟺ Q`, you will have two goals (after `constructor`): `P → Q` and `Q → P`. You might find that one or both these implications are easier to prove via their contrapositive, so you would use `contrapose!`.

```

example (h_nq_np : ¬ Q → ¬ P) (h_q_p : Q → P) : P ⟺ Q := by
  constructor
  -- Goal 1 : P → Q
  -- Let's try to prove this one by contraposition.
  contrapose!
  exact h_nq_np

  -- Goal 2 : Q → P
  -- We have h_q_p : Q → P
  exact h_q_p
example (h_nq_np : ¬ Q → ¬ P) (h_q_p : Q → P) : P ⟺ Q := by
  constructor
```

```
  -- Goal 1 : P → Q
  -- Let's try to prove this one by contraposition.
  contrapose!
  exact h_nq_np

  -- Goal 2 : Q → P
  -- We have h_q_p : Q → P
  exact h_q_p
```

What is Monotonicity?

In mathematics, a function f between two ordered sets is called *monotone* if it respects the order. That is, if `x ≤ y`, then `f x ≤ f y`.

There is also *antitone*, meaning if `x ≤ y`, then `f y ≤ f x`.

In Lean's `mathlib`, you will typically find definitions like `Monotone f` and `Antitone f`. These definitions usually look something like this

`def Monotone [Preorder α] [Preorder β] ( f : α → β) : Prop := ∀ { x y : α} , x ≤ y → f x ≤ f y`

Using Monotonicity in Lean Proofs

You usually do not use a single mono tactic as often as you apply specific monotonicity lemmas or the definition itself.

1. **If you have `h : Monotone f`:** This `h` is a function. It is a proof that `∀ x y, x ≤ y → f x ≤ f y`. So if you have `h_xy : x ≤ y` , you can get a proof of `f x ≤ f y` by applying `h h_hxy`.
2. **Proving `Monotone f`:** You need to prove `∀ x y, x ≤ y → f x ≤ f y`. You would start your proof with `intros x y h_xy` and then prove `f x ≤ f y`.
3. **Congruence/Order tactics:** Sometimes, tactics like `gcongr` (generalized congruence) or `rel_tac` can help apply order-related lemmas, but applying the `Monotone` hypothesis directly is very common.

Example : Conjunction with Monotonicity

Let's prove that if a function `f` is monotone, and we have `x ≤ y` and `a ≤ b`, then we can conclude `f x ≤ f y` and `f a ≤ f b`.

```

variable (α β : Type) [Preorder α] [Preorder β] (f : α → β ) (x y a b : α )

example (h_mono : Monotone f) : (x ≤ y ∧ a ≤ b) → (f x ≤ f y ∧ f a ≤ f b) := by
  -- Use rintro to introduce the hypothesis and deconstruct it
  rintro ⟨ h_xy, h_ab ⟩
  -- Now we have h_xy : x ≤ y and h_ab : a ≤ b
  -- Goal : f x ≤ f y ∧ f a ≤ f b
  -- Use constructor <;> to split the goal and apply a tactic to each.
  constructor <;> {
  -- Using { ---- } for a tactic block applied to each goal.
    apply h_mono;
  -- Now for the first goal, the subgoal is x ≤ y and the second goal is a ≤ b. Assumption
    will do this job.
    assumption;
  }
```