Day - 8 : Negation and Proof by Contradiction

Touseef Haider

Study Mathematics In Lean(MIL) Section 3.3

Negation ¬ :

Logically ¬ P (read as "not P") means that the proposition P is false. In Lean4, ¬ P is often defined as implication ¬ P := P → false. Where false is a proposition that is inherently false (it has no proofs). So, ¬ P means "if P is true, it would lead to a Contradiction (false)". This definition is very powerful because it allows us to use the tactics we already know for implications (like intro) to prove negations.

Proving ¬ P :

If you want to prove ¬ P, given that ¬ P is P → false, you would typically use the intro tactic:

- Assume P is true (e.g., intro hP where hP : P).
- Your goal becomes False.
- You then need to derive a contradiction (False) from hP and any other available hypotheses.

Using ¬ P (i.e., h : P → False):

If you have a hypothesis h : ¬ P (which means h : P → False), and you also manage to derive a proof of P (say, hP_proof : P), then you can obtain a proof of False by applying h to hP_proof : h hP_proof would be a proof of False.

This P → False definition of negation is quite elegant.

A proof by contradiction works like this:

- To prove a proposition P : Assume ¬ P is true. If this assumption leads to logical impossibility (a contradiction, or False), then the original assumption ¬ P must be false, which means P must be true.
- To prove ¬ P: Assume P is true. If this leads to False, then P must be false, so ¬ P is true.

Lean's core logic is constructive. In constructive logic, proving P by assuming ¬ P and deriving False is how ¬¬ P is proven. To get from ¬¬ P to P (which is what "proof by contraction for P" often implies) typically requires an appeal to classical logic, specifically the Law of Excluded Middle (P ∨ ¬ P). Lean4 allows you to use classical reasoning.

1. The by_contra Tactic : When you want to prove a proposition P using classical proof by contradiction, you can use the by_contra tactic. If your goal is P, and you use by_contra h_np (where h_np is a name you choose for the new hypothesis). This tactic will:
   - Add h_np : ¬ P (i.e., h_np : P → False) to your hypotheses.
   - Change your goal to False.

   You then need to use h_np and other available facts to derive a contradiction.

```
example (P : Prop) -- ... other hypotheses ... : P := by
  by_contra h_not_P -- Assume h_not_P : ¬P, goal becomes False
  -- ... proof steps to derive False ...
  -- For instance, if you can show P from other means (call it h_P_somehow):
  -- exact (h_not_P h_P_somehow) -- This would be h_not_P (P → False) applied to P,
    yielding False
```

2. The `absurd` Tactic: The `absurd` tactic is useful when you have already derived both a proposition `P` and its negation `¬ P` in your context, and you want to conclude `False` (or even close any goal, because from `False`, anything follows by `False.elim`). If you have

   - `hP : P`
   - `hnP : ¬ P` (which is `P → False`

   Then `absurd hP hnP` will prove `False`. Since anything follows from `False` (this is known as the principle of explosion, or `False.elim` in Lean), if your goal is , say , `Q`, and you can show `False` by `absurd hP hnPP`, then `absurd hP hnP` can close the goal `Q`.

```
example (P Q : Prop) (hP : P) (hnP : ¬P) : Q := by
  -- We have P and ¬P, which is a contradiction.
  -- From a contradiction, any goal Q can be proven.
  exact absurd hP hnP -- The type of 'absurd hP hnP' is False.
                      -- 'exact' uses False.elim to make it match goal Q.
-- Or more explicitly:
example (P Q : Prop) (hP : P) (hnP : ¬P) : Q := by
  have h_false : False := absurd hP hnP
  exact False.elim h_false
```

   Often, if your goal is `False` directly, you might use `exact (hnP hP)`. The `absurd` tactic can be a bit more expressive or useful if the goal isn't immediately `False` but you have reached a contradiction.

A note on classical logic:

The `by_contra` tactic effectively assumes the law of exluded middle (`P ∨ ¬ P`) for the proposition `P`. If you are working within purely constructive mathematics, you would avoid `by_contra` for proving positive (non-negated) statements unless `¬¬ P → P` is constructively provable for that specific `P`. However, for much of standard mathematics formalized in Lean, classical logic (and thus `by_contra`) is commonly used.

Example - 1

A common classical result is that if the contrapositive `(¬ Q → ¬ P)` holds, then `(P → Q)` also holds. Remembering that `¬ X` is `X → False`, the statement is :
`((Q → False) → (P → False)) → (P → Q)`.
We will prove this using `by_contra`.
Theorem : `∀ (P Q : Prop), ((Q → False) → (P → False)) → (P → Q)`.

```
example (P Q : Prop) (h_contrapositive : (Q → False) → (P → False)) : P → Q := by
  -- Our goal is P → Q.
  -- To prove an implication, we introduce the hypothesis.
  intro hP
  -- Now, hP : P is a hypothesis.
  -- Our goal is Q.

  -- At this point, proving Q directly might be tricky.
  -- Let's try to prove Q by contradiction.
  -- We assume ¬ Q (not Q) and aim to derive False.
  by_contra hnQ
  -- Now, hnQ : ¬Q (which means hnQ : Q → False) is a hypothesis.
  -- Our goal has changed to False.

  -- We have h_contrapositive : (Q → False) → (P → False).
  -- We also have hnQ : Q → False.
  -- We can apply h_contrapositive to hnQ.
  let hnP := h_contrapositive hnQ
  -- Now, hnP : P → False (which means hnP : ¬P).

  -- We have hP : P (from the 'intro hP' earlier).
  -- And we just derived hnP : ¬P.
  -- So, we have P and ¬P, which is a contradiction.
  -- Applying hnP (which is P → False) to hP (which is P) gives False.
```

```
  exact (hnP hP)
  -- This proves False, which was our goal after 'by_contra'.
```

1. `intro hP`:
   - Our initial goal is `P → Q`. The standard way to prove this is to assume `P` and then prove `Q`.
   - `intro hP` adds `hP : P` to our hypotheses, and the goal becomes `Q`.
2. `by_contra hnQ`:
   - We are now trying to prove `Q`. We decide to use proof by contradiction. `by_contra hnQ` does two things. First, it adds `hnQ : ¬ Q` (which is `Q → False`) to our hypotheses. We are assuming `Q` is flase. Second, it changes our goal to `False`. The staretgy is now to show that assuming ¬`Q` (along with `P`) leads to a contradiction.
3. `let hnP := h_contrapositive hnQ`:
   - We have our main hypothesis `h_contrapositive : (Q → False) → ( P → False)`.
   - We also have `hnQ : Q → False` from our `by_contra` step.
   - We can apply `h_contrapositive` to `hnQ`. Since `h_contrapositive` is a function expecting an argument of type `(Q → False)`, and `(P → False)`.
   - We name this `hnP`. So, `hnP : P → False` (which means `hnP :¬ P`).
4. `exact (hnP hP)`:
   - At this point, our goal is `False`.
   - We have `hP : P`.
   - We have `hnP : P → False`.
   - Applying `hnP` to `hP` gives us a proof of `False`.
   - `exact` uses this proof of `False` to satisfy the goal.

This successfully shows that if `¬ Q` leads to `False`, then our initial setup implies a contradiction, thus proving `Q`.

Example - 2

In the example above, the last step was `exact (hbP hP)`. Since `hnP` is `¬ P` and `hP` is `P`, this directly yields `False`. We could also have written:

```
-- ..
-- let hnP := h_contrapositive hnQ -- hnP : ¬ P
-- We have hP : P
exact (absurd hP hnP)
```

Here, `absurd hP hnP` recognizes that `hP` and `hnP` are contradictory and itself evaluates to a proof of `False`. So `exact (absurd hP hnP)` also closes the goal. The `absurd` tactic can be more explicit when you want to highlight that you are using a direct contradiction between two existing hypotheses.

Asymmetry of Less Than `a < b → ¬ b < a`

```
variable (a b : ℝ)

example (h : a < b) : ¬ b < a := by
  -- ¬ P mean P → False. So we assume b < a and try to prove False.
  intro h' -- Assume h' : b < a
  -- We have h : a < b and h' : b < a
  -- lt_trans is the transitivity rule: x < y → y < z → x < z.
  -- If we apply it to h and h' we get a < a
```

```
  have : a < a := lt_trans h h'
  -- Now we have a < a
  -- lt_irrefl a is the irreflexivity rule : a < a → False.
  apply lt_irrefl a this
  -- This gives us 'False', which was our goal after intro h'.
```

Standard Negation Lemmas for Inequalities

These #check commands show the type of important lemmas from mathlib. They are often based on the law of trichotomy (for any a, b, exactly one of a<b , a=b or a >b holds) and require classical logic.

```
#check (not_le_of_gt : a > b → ¬ a ≤ b)
#check (not_lt_of_ge : a ≥ b → ¬ a < b)
#check (lt_of_not_ge : ¬ a ≥ b → a < b)
#check (le_of_not_gt : ¬ a > b → a ≤ b)
```

Negating a Universal Statement about Monotonicity

```
-- Goal : Prove that it is not true that every monotone function f has the property that f a
     ≤ f b → a ≤ b.
example : ¬ ∀ {f : ℝ → ℝ}, Monotone f → ∀ {a b}, f a ≤ f b → a ≤ b := by
  -- We need to prove ¬ P. So we assume P and aim for False.
  intro h -- Assume : ∀ {f : ℝ → ℝ }, Monotone f → ∀ {a b : ℝ }, f a ≤ f b → a ≤ b
  -- To show a universal statement is false, we need a counterexample.
  -- Our counterexample is the constant function f(x)=0
  let f := fun x : ℝ ↦ (0 : ℝ)

-- First, we need to prove our counterexample function IS monotone
  have monof : Monotone f := by
  -- To prove Monotone f, we need to show x ≤ y → f x ≤ f y
    intros x y h__xy -- Assume x ≤ y . Goal is f x ≤ f y.
    -- Since f x = 0 and f y = 0, the goal is 0 ≤≤ 0
    -- le_refl 0 proves 0 ≤ 0
    rfl

-- Now let's pick a specific a and b to make the implication fail.
-- We want f a ≤ f b to be true, but a ≤ b to be false.
-- Let a =1 and b=0. Then a ≤ b is 1 ≤ 0 , which is false.
-- f 1 ≤ f 0 is 0 ≤ 0 which is true.
  have h' : f 1 ≤ f 0 := le_refl _

  -- Now, let's use our assumption monof
  -- It claims that For our f (since it is monotone), f a ≤ f b → a ≤ b must hold for all
    a and b
  -- Let's apply it to our f and monof
  have : (1 : ℝ) ≤ 0 := h monof h'

  linarith
```

Proving x ≤ 0 using le_of_not_gt

```
-- Goal : If x is less than every positive ϵ , then x ≤ 0.
example (x : ℝ) (h : ∀ ϵ > 0, x < ϵ) : x ≤ 0 := by
  -- We want to use le_of_not_gt. This lemma states : ¬ x > 0 → x ≤ 0.
  -- So if we can prove ¬ x > 0, we are done.
  apply le_of_not_gt
  -- Our new goal is ¬ x > 0, which mean x > 0 → False.
  intro hx -- Assume x > 0. Goal is False.
  -- Take ϵ = x, which is > 0 since hx : x > 0
  specialize h x hx
  -- But then x < x, which is impossible
  exact lt_irrefl x h
```

Principle of Explosion

```
variable (a : ℕ)

-- Goal : If 0 < 0 (a false statement), then a > 37 (an arbitrary statement)
example (h : 0 < 0) : a > 37 := by
  -- From a false statement, anything follows.
  -- exfalso changes the current goal to False.
  -- We are now saying I can prove False, therefore I can prove anything.
  exfalso
  -- Our goal is now False.
  -- We can prove False because we know 0 < 0 is impossible.
  -- lt_irrefl 0 is 0 < 0 → False.
  apply lt_irrefl 0 h
```