Day - 6 : Implication and Universal Quantifier

Touseef Haider

Study Mathematics In Lean(MIL) Section 3.1

6.1. Implication $\rightarrow$:

In logic and in Lean 4, an implication is a statement of the form " if P, then Q". It is written as `P → Q`.

- `P` is called the hypothesis.
- `Q` is called the conclusion.

The statement `P → Q` asserts that whenever `P` is true, `Q` must also be true. It does not say anything about whether `P` or `Q` are true on their own. Think of it as a promise: "I promise that *if* you give me a proof of P, I can give you a proof of Q."

How to prove an implication ($P \rightarrow Q$) in Lean4:

The most common way to prove an implication `P → Q` is to :

- Assume `P` is true.
- Using that assumption, show that `Q` must be true.

The Lean4 tactic for this is `intro`. If your goal is `P → Q`, typing `intro hP` (or any other name instead of `hP`) will:

- Add `hP : P` to your list of hypotheses (you are assuming P).
- Change your goal to `Q`.

Example:

Let's say we want to prove `Nat → Nat` (a very simple, somewhat silly tupe, but it illustrates the point). In a Lean file, you might have:

```
example : Nat → Nat := by
  intro n  -- n is now a hypothesis of type Nat
  exact n  -- our goal is Nat, and n is a Nat, so we are done
```

Here `P` is `Nat` (the input type) and `Q` is `Nat` (the output type). `intro n` assumes we have a natural number `n`. The goal becomes to provide a `Nat` and `n` itself fulfills this.

If you already have a hypothesis `h` that states `P → Q`, and you also have a proof of `P` (let's call it `hP_proof : P`), you can use them to get a proof of `Q`.

The Lean4 tactic for this is `apply`. If you have `h : P → Q` and your current goal is `Q`, typing `apply h` will:

- Try to match `Q` with the conclusion of `h`.
- If it matches, it will change your goal to `P` (the premise of `h`). You now need to prove `P`.

If you have `h: P → Q` and also `hP_proof : P`, you can directly get a proof of `Q` by writing `h hP_proof`(standard function application, applying function `h` to argument `hP_proof`). Lean understands function application, and implications are like functions that take a proof of `P` and return a proof of `Q`.

Example

Suppose we have:
`P, Q : Prop` (Propositions, i.e., true/false statements)
`hPQ : P → Q`
`hP : P`

And we want to prove `Q`.

```
variable (P Q : Prop)
variable (hPQ : P → Q)
variable (hP : P)

example : Q := by
  apply hPQ  -- Goal changes from Q to P
  exact hP   -- We have hP : P, so this closes the goal
```

Alternatively:

```
example : Q :=
  hPQ hP -- This directly provides the proof of Q
```

This is a fundamental concept. If `P` implies `Q`, and `P` is true, then `Q` must be true. It's like saying, "If it's raining (`P`), then ground is wet (`Q`). It is raining (`hP`). Therefore, the ground is wet."
A little note about the notion `h hP_proof`. This notion is standard function application. You are applying the "function" `h` to the "argument" `hP_proof`.

- Since `h` is of type `P → Q` (it expects a proof of `P` and yields a proof of `Q`)
- And `hP_proof` is of type `P` (it is a proof of `P`),
- The result of `h hP_proof` is an object of type `Q` (a proof of `Q`).

6.2 Universal Quantifier (∀):

The universal quantifier, written as ∀ is used to make statements about all elements of a certain type.
A statement like `∀ x : T, P x` reads: "For all `x` of type `T`, the property `P x` holds true."

- `x` is a variable.
- `T` is the type of `x`. (e.g. `Nat` for natural numbers, `Real` for real numbers)
- `P x` is a proposition (a statement that can be true or false) that depends on `x`.

How to prove a universal quantification (`∀ x : T, P x`) in Lean4:

To prove that a property `P x` holds for all `x` of type `T`, the standard method is :

- Consider an arbitrary element `x` from type `T`. you make no assumptions about `x` other than it being of type `T`.
- Then, show that `P x` holds for this arbitrary `x`.

Example:

Let's say we want to prove that for any natural number `n`, `n=n`.

```
example : ∀ n : Nat, n = n := by
  intro n  -- n is now an arbitrary hypothesis of type Nat
  rfl      -- The goal is n = n. ‘rfl‘ (reflexivity) proves this.
```

Here, `intro n` picks an arbitrary natural number `n`. Then we need to show `n=n`, which `rfl` does.

How to use universal quantification (`h : ∀ x : T, P x`) in Lean4:

If you have a hypothesis `h` that states `∀ x : T, P x`, it means you know that `P x` is true for any `x` you choose from type `T`. So you can use this hypothesis with a specific value.

- Direct Application: If you have `h : ∀ x : T, P x` and a specific value `val : T`, then `h val` is a proof of P val. This is again like function application : `h` is a function that takes any term of type `T` and gives you a proof that `P` holds for that term.
- `specialize` tactic: If you have `h : ∀ x : T, P x` and you want to create a new hypothesis that `P` holds for a specific value `val : T` (perhaps `val` is another hypothesis or a term you have constructed), you can use the `specialize` tactic. Typing `specialize h val` will change the hypothesis `h: ∀ x : T, P x` into `h : P val`.

Example:

Suppose we have :
`is_even : Nat → Prop` (a property of natural numbers)
`h_all_even_implies_something : ∀ n : Nat, is_even n → Q` (for some proposition Q)
`h_two_is_even : is_even 2`

And we want to use `h_all_even_implies_something` specifically for `n=2`.

Using direct application in a term:

```
variable (Q : Prop)
variable (is_even : Nat → Prop)
variable (h_all_even_implies_something : ∀ n : Nat, is_even n → Q)
variable (h_two_is_even : is_even 2)

-- We want to get a proof of (is_even 2 → Q)
def specific_implication : is_even 2 → Q := h_all_even_implies_something 2

-- If we wanted Q itself:
example : Q :=
  (h_all_even_implies_something 2) h_two_is_even
```

Using the `specialize` tactic:

```
example (Q : Prop) (is_even : Nat → Prop)
        (h_all_even_implies_something : ∀ n : Nat, is_even n → Q)
        (h_two_is_even : is_even 2) : Q := by
  -- h_all_even_implies_something : ∀ (n : Nat), is_even n → Q
  specialize h_all_even_implies_something 2
  -- Now, h_all_even_implies_something : is_even 2 → Q
  apply h_all_even_implies_something
  exact h_two_is_even
```

In this tactic example, after `specialize h_all_even_implies_something 2`, the hypothesis `h_all_even_implies_someth` is updated to be `is_even 2 → Q`. Then we can `apply` it and use `h_two_is_even`.

The universal quantifier ∀ and implication → are very closely related. In fact, `P → Q` can ben seen as `∀ (_ : P), Q` where you quantify over proofs of `P`. This is why `intro` works for both.

Example:

If you have a function `f` that takes an element of type α and gives you an element of type β (so, $f : \alpha \to \beta$) and another function `g` that takes an element of type β and gives you an element of type γ (so, $g : \beta \to \gamma$), then you can compose them to create a new function that takes an element of type α and gives you an element of type γ. This new composed function, let's call it `g_comp_f`, would behave such that `g_comp_f a = g (f (a))` for any `a : α`.

In Lean, we can state this as a theorem:
For any types α, β, γ, and any functions `g : β → γ` and `f : α → β`, there exists a function from $\alpha \to \gamma$ (which is `g ∘ f`).

```
-- This is the statement we want to prove
example : ∀ (α β γ : Type) (g : β → γ) (f : α → β), α → γ := by
  -- To prove the ∀ for types, we introduce arbitrary types α, β, γ
  intro α
  intro β
  intro γ
  -- Our goal is now: (g : β → γ) → (f : α → β) → α → γ

  -- To prove the ∀ for functions g and f (which are actually implications in the goal now),
  -- we introduce arbitrary functions g and f with the specified types.
  intro g  -- g : β → γ is now a hypothesis
  intro f  -- f : α → β is now a hypothesis
  -- Our goal is now: α → γ

  -- To prove the implication α → γ, we introduce an arbitrary element 'a' of type α
  intro a  -- a : α is now a hypothesis
  -- Our goal is now: γ

  -- We need to construct an element of type γ.
  -- We have g : β → γ. If we can make something of type β, we can apply g to it.
  -- We have f : α → β and a : α.
  -- So, (f a) has type β.
  let beta_val := f a  -- This step is optional, just for clarity. beta_val has type β.

  -- Now apply g to beta_val (which is f a) to get something of type γ.
  exact (g beta_val) -- or more directly: exact (g (f a))
```

Let's break down the tactics and reasoning:

- `intro α`, `intro β`, `intro γ`: These handle the `∀ (α β γ : Type)`.
- `intro g`, `intro f`: These handle the `∀ ( g: β → γ) (f : α → β)`.
- `intro a`: the goal is now $\alpha \to \gamma$. To prove this implication, we assume the hypothesis `a : α`. Our context adds `a : α`, and the goal becomes γ.
- `let beta_val := f a`: We know `f : α → β` and `a : α`. Applying `f` to `a` gives `f a`, which has type β. We can name this intermediate value `beta_val`. This is using our hypothesis `f`.
- `exact (g beta_val)` or `exact (g (f a))`: We know `g : β → γ` and `beta_val` (which is `f a`) has type β. Applying `g` to `beta_val` gives `g beta_val`, which has type γ. This matches our goal, so `exact` completes the proof for this branch.

This example shows `intro` being used for both universal quantifiers (over types and over specific functions which themselves are implications) and for the final implication (to get `a : α`). Then, it uses the hypotheses (which are functions/implications) by applying them (`f a` and `g ( . . .)`).

Function composition is so common that there's an operator ∘ for it, so you could also define the composed function as `g ∘ f`.