Contents

1 Classical Logic

Logic is the study of reasoning, branching into various systems. We refer to **classical logic** as the one that underpins much of traditional mathematics. It's the logic of the ancient Greeks (not fair) and truthtables, and it remains used nowadays for pedagogical reasons. We first introduce **propositional logic**, which is the simplest form of classical logic. Later we will extend this to **predicate (or first-order) logic**, which includes quantifiers and predicates. In this setting, a **proposition** is a statement that is either true or false, and a **proof** is a logical argument that establishes the truth of a proposition. Propositions are constructed via **formulas** built from **propositional variables** (also called atomic propositions) combined with logical **connectives** such as "and" ($\land$), "or" ($\lor$), "not" ($\neg$), "implies" ($\Rightarrow$), and "if and only if" ($\Leftrightarrow$). These connectives allow the creation of complex or compound propositions.

Definition 1.1 (Propositional Formula) *([Tho99]) A **propositional formula** is either:*

- *A **propositional variable:** $X_0, X_1, X_2, \ldots$, or*
- *A **compound formula** formed by combining formulas using connectives:*

$$(A \land B), \quad (A \Rightarrow B), \quad (A \lor B), \quad \bot, \quad (A \Leftrightarrow B), \quad (\neg A)$$

  where A and B are formulas themselves.

We are going to describe classical logic though a formal framework called **natural deduction system** developed by Gentzen in the 1930s ([Wad15]). It specifies rules for deriving **conclusions** from **premises** (assumptions from other propositions), called **inference rules**.

Example 1.2 (Deductive style rule) *Here is an hypothetical example of inference rule.*

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_n}{C}$$

Where the $P_1, P_2, \ldots, P_n$, above the line, are hypothetical premises and, the hypothetical conclusion C is below the line.

The inference rules needed are:

- **Introduction rules** specify how to form compound propositions from simpler ones, and
- **Elimination rules** specify how to use compound propositions to derive information about their components.

Let's look at how we can define some connectives.

Conjunction ($\wedge$)

- Introduction

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-Intro}$$

- Elimination

$$\frac{A \wedge B}{A}\ \wedge\text{-Elim}_1 \qquad\qquad \frac{A \wedge B}{B}\ \wedge\text{-Elim}_2$$

Disjunction ($\vee$)

- Introduction

$$\frac{A}{A \vee B}\ \vee\text{-Intro}_1 \qquad\qquad \frac{B}{A \vee B}\ \vee\text{-Intro}_2$$

- Elimination (Proof by cases)

$$\frac{A \vee B \qquad [A] \vdash C \qquad [B] \vdash C}{C}\ \vee\text{-Elim}$$

Implication ($\rightarrow$)

- Introduction

$$\frac{[A] \vdash B}{A \rightarrow B}\ \rightarrow\text{-Intro}$$

- Elimination (Modus Ponens)

$$\frac{A \rightarrow B \qquad A}{B}\ \rightarrow\text{-Elim}$$

Notation 1.3 *We use $A \vdash B$ (called turnstile) to designate a deduction of B from A. It is employed in Gentzen's* ***sequent calculus*** *([GTL89]), whereas in natural deduction the corresponding symbol is*

$$\begin{matrix} A \\ \vdots \\ B \end{matrix}$$

There are some minor differences, in fact, which I don't fully understand. The square brackets around a premise $[A]$ mean that the premise A is meant to be ***discharged*** *at the conclusion. The classical example is the introduction rule for the implication connective. To prove an implication $A \rightarrow B$, we assume A (shown as $[A]$), derive B under this assumption, and then discharge the assumption A to conclude that $A \rightarrow B$ holds without the assumption. The turnstile is predominantly used in judgments and type theory with the meaning of "entails that".*

Lean has its own syntax for connectives and their relative inference rules. For instance $A \wedge B$ can be presented as `And(A, B)` or `A ∧ B`, . Its untroduction rule is constructed by `And.intro _ _` or shortly `⟨_, _⟩` (underscore are placeholder for assumptions or "propositional functions"). The pair $A \wedge B$ can be then consumed using elimination rules `And.left` and `And.right`.

Example 1.4 *Let's look at our first Lean example*

```
example (H_A : A) (H_B : B) : (A ∧ B) := And.intro H_A H_B
```

Lean aims to resemble the language used in mathematics. For instance, when defining a function or expression, one can use keywords such as `theorem` *or* `def`*. Here, I used* `example`*, which is handy for defining anonymous expressions for demonstration purposes. After that comes the statement to be proved:*

```
(H_A : A) (H_B : B) : (A ∧ B)
```

Meaning given a proof of A and a proof of B we can form a proof of $(A \wedge B)$. The operator `:=` *assigns a value (or return an expression) for the statement which "has to be a proof of it".* `And.intro` *is implemented as:*

```
And.intro: p → q → (p ∧ q).
```

It says: if you give me a proof of p and a proof of q, then i return a proof of $p \wedge q$. We therefore conclude the proof by directly giving `And.intro H_A H_B`*. Here another way of writing the same statment.*

```
example (H_p : p) (H_B : B) : And(A, B) := ⟨H_p, H_B⟩
```

This system of inference rules allows us to construct proofs in an algorithmic and systematical way, organized in what is called a **proof tree**. To reduce complexity, we follow a **top-down** approach (see [Tho99] and [NPS90]). This methodology forms the basis of **proof assistants** like Lean, Coq, and Agda, which help verify the correctness of mathematical proofs by checking each step against these rules. We will see later that Lean, in fact, provides an info view of the proof tree which helps us understand and visualize the proof structure. Let's examine a concrete example of a proof.

Example 1.5 (Associativity of Conjunction) *We prove that $(A \wedge B) \wedge C$ implies $A \wedge (B \wedge C)$. First, from the assumption $(A \wedge B) \wedge C$, we can derive A:*

$$\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B}\ \wedge E_1}{A}\ \wedge E_1$$

Second, we can derive $B \wedge C$:

$$\dfrac{\dfrac{\dfrac{(A \wedge B) \wedge C}{A \wedge B}\ \wedge E_1}{B}\ \wedge E_2 \qquad \dfrac{(A \wedge B) \wedge C}{C}\ \wedge E_2}{B \wedge C}\ \wedge I$$

Finally, combining these derivations we obtain $A \wedge (B \wedge C)$:

$$\frac{(A \wedge B) \wedge C \vdash A \qquad (A \wedge B) \wedge C \vdash B \wedge C}{A \wedge (B \wedge C)} \wedge I$$

Example 1.6 (Lean Implementation) *Let us now implement the same proof in Lean.*

```
theorem and_associative : (A ∧ B) ∧ C → A ∧ (B ∧ C) :=
  fun h : (A ∧ B) ∧ C =>
    -- First, from the assumption (A ∧ B) ∧ C, we can derive A:
    have ab : A ∧ B := h.left -- extracts (derive) a proof of (A ∧ B)
    from the assumption
    have a : A := ab.left -- extracts A from (A ∧ B)
    -- Second, we can derive B ∧ C (here we only extract b and c and
    combine them in the next step)
    have c : C := h.right
    have b : B := ab.right
    -- Finally, combining these derivations we obtain A ∧ (B ∧ C)
    show A ∧ (B ∧ C) from ⟨a, ⟨b, c⟩⟩
```

Listing 1: Associativity of Conjunction in Lean

We introduce the `theorem` *with the name* `and_associative`*, which can be referenced in subsequent proofs. The type signature* `(A ∧ B) ∧ C → A ∧ (B ∧ C)` *represents our logical implication. The* `:=` *operator introduces the proof term that establishes the theorem's validity. In the previous code example this proof was directly given. Here, we construct it using a function with the* `fun` *keyword. Why a function? We have already encountered the Curry-Howard correspondence in Lean previously, though without explicitly stating it. According to this correspondence, a proof of an implication can be understood as a function that takes a hypothesis as input and produces the desired conclusion as output. We will revisit this concept in more detail later. The* `have` *keyword introduces local lemmas within our proof scope, allowing us to break down complex reasoning into manageable intermediate steps, mirroring our natural deduction proof of before. Finally, the* `show` *keyword presents our final result.*

To capture more complex mathematical ideas, we extend our system from propositional logic to **predicate logic**. A **predicate** is a statement or proposition that depends on a variable. In propositional logic we represent a proposition simply by P. In predicate logic, this is generalized: a predicate is written as $P(a)$, where a is a variable. Notice that a predicate is just a function. This extension allows us to introduce **quantifiers**: $\forall$ ("for all") and $\exists$ ("there exists"). These quantifiers express that a given formula holds either for every object or for at least one object, respectively. In Lean if α is any type, we can represent a predicate `P` on α as an object of type α → `Prop`. `Prop` stands for proposition, and it is an essential component of Lean's type system. For now, we can think of it as a special type; somewhat paradoxically, a type of types. Thus given an `x : α` (an element with type α) `P(x) : Prop` would be representative of a proposition holding for `x`.

When introducing variables into a formal language we must keep in mind that the specific choice of a variable name can be substituted without changing the meaning of the predicate or statement. This should feel familiar from mathematics, where the meaning of an expression does not depend on the names we assign to variables. Some variables are **bound** (constrained), while others remain **free** (arbitrary, in programming often called "dummy" variables). When substituting variables, it is important to ensure that this distinction is preserved. This phenomenon, called **variable capture**, parallels familiar mathematical practice: if $f(x) \coloneqq \int_1^2 \frac{dt}{x-t}$, then $f(t)$ equals $\int_1^2 \frac{ds}{t-s}$, not the ill-defined $\int_1^2 \frac{dt}{t-t}$. The same principle applies to predicate logic. For example, consider

$$\exists y.\,(y > x).$$

This states that for a given x there exists a y such that $y > x$. If we naively substitute $y + 1$ for x, we would obtain

$$\exists y.\,(y > y + 1),$$

where the y in $y + 1$ has been **captured** by the quantifier $\exists y$. This transforms the original statement from "there exists some y greater than the free variable x" into the always-false statement "there exists some y greater than itself plus one." To avoid the probelm, in the above example, we would first rename the bound variable to something fresh say z, obtaining $\exists z.\,(z > x)$, and then safely substitute to get $\exists z.\,(z > y + 1)$. We use the notation $\phi[t/x]$ for **substitution**, meaning all occurrences of the free variable x in formulab (or expression) ϕ are replaced by term t. We can now present the inference rules for quantifiers.

Universal Quantification ($\forall$)

- Introduction

$$\frac{A}{\forall x.\,A}\;\forall I$$

  The variable x must be arbitrary in the derivation of A. This rule captures statements like $\forall x \in \mathbb{N}$, x has a successor, but would not apply to $\forall x \in \mathbb{N}$, x is prime (since we cannot derive this for an arbitrary natural number).

- Elimination

$$\frac{\forall x.\,A}{A[t/x]}\;\forall E$$

  The conclusion $A[t/x]$ represents the substitution of term t for variable x in formula A. From a proof of $\forall x.\,A(x)$ we can infer $A(t)$ for any term t.

Existential Quantification ($\exists$)

- Introduction

$$\frac{A[t/x]}{\exists x.\, A}\,\exists I$$

  The substitution premise means that if we can find a specific term t for which $A(t)$ holds, then we can introduce the existential quantifier. The introduction rule requires a witness t for which the predicate holds.

- Elimination

$$\frac{\exists x.\, A \qquad [A] \vdash B}{B}\,\exists E$$

  To eliminate an existential quantifier, we assume A holds for some witness and derive B without making any assumptions about the specific witness.

We can give an informal reading of the quantifiers as infinite logical operations:

$$\forall x.\, A(x) \equiv A(a) \wedge A(b) \wedge A(c) \wedge \dots$$
$$\exists x.\, A(x) \equiv A(a) \vee A(b) \vee A(c) \vee \dots$$

The expression $\forall x.\, P(x)$ can be understood as generalized form of implication. If P is any proposition, then $\forall x.\, P$ expresses that P holds regardless of the choice of x. When p depends on x, this captures the idea that we can derive p from any assumption about x. Morover, there is a duality between universal and existential quantification. We shall develop all this dicussions further after exploring their computational (type theoretical) meaning.

Example 1.7 *Lean espresses quantifiers as follow.*

```
∀ (x : X), P x
forall (x : X), P x -- another notation
```

Listing 2: For All

```
∃ (x : X), P x
exist (x : X), P x -- another notation
```

Listing 3: Exists

Where `x` is a varible with a type `X`, and `P x` is a proposition, or predicate, holding for `x`.

When introducing an **existential** proof, we need a **pair** consisting of a witness and a proof that this witness satisfies the statement.

```
example (x : Nat) (h : x > 0) : ∃ y, y < x :=
  Exists.intro 0 h -- or shortly ⟨0, h⟩
```

The **existential elimination rule** (`Exists.elim`) performs the opposite operation. It allows us to prove a proposition q from $\exists x : \alpha, p\, x$ by showing that q follows from $p\, w$ for an **arbitrary** value w. The existential rules can be interpreted as an infinite disjunction, so that existential elimination naturally corresponds to a **proof by cases** (with only one single case). In Lean, this reasoning is carried out using **pattern matching**, a known mechanism in functional programming for dealing with cases, with `let` or `match`, as well as by using `cases` or `rcases` construct. For example

```
example (h : ∃ n : Nat, n > 0) : ∃ n : Nat, n > 0 :=
  match h with
  | ⟨witness, proof⟩ => ⟨witness, proof⟩
```

The **universal quantifier** may be regarded as a generalized function. Accordingly, In Lean, universal elimination is simply function application. For example:

```
example : ∀ n : Nat, n ≥ 0 :=
  fun n => Nat.zero_le n
```

Functions are primitive objects in type theory. For instance, the type of the identity function is written as $\alpha \to \alpha$, where α is a type variable. This indicates that the function is polymorphic: it has the same type for both its domain and codomain. In practice, this means it can be applied to booleans (returning a boolean), numbers (returning a number), functions (returning a function), and so on. In the same spirit, we can define a transitivity proof of a polymorphic relation as follows:

```
def Transitive (α : Type) (R : α → α → Prop) : Prop :=
  ∀ x y z, R x y → R y z → R x z
```

To use `Transitive`, we must provide both the type α and the relation itself. For example, here is a proof of transitivity for the less-than relation on $\mathbb{N}$:

```
theorem lt_trans_proof : Transitive Nat (· ≤ · : Nat → Nat → Prop) :=
  fun x y z h1 h2 => Nat.lt_trans h1 h2
```

Looking at this code, we immediately notice that explicitly passing the type argument `Nat` is somewhat repetitive. Lean allows us to omit it by letting the type inference mechanism fill it in automatically. This is achieved by using *implicit arguments* with curly brackets:

```
def Transitive {α : Type} (R : α → α → Prop) : Prop :=
  ∀ x y z, R x y → R y z → R x z
```

With this change, the same example becomes more concise:

```
theorem lt_trans_proof : Transitive (· ≤ · : Nat → Nat → Prop) :=
  fun x y z h1 h2 => Nat.lt_trans h1 h2
```

Lean's type inference system is quite powerful: in many cases, types can be completely inferred without explicit annotations. For instance, in earlier examples, Lean automatically inferred that the types of A, B, and C were `Prop`. Let us now revisit the transitivity proof, but this time for the less-than relation on the real numbers instead of the natural numbers:

```
theorem real_lt_trans_proof : Transitive (· ≤ · : ℝ → ℝ → Prop) :=
  fun x y z h1 h2 => le_trans h1 h2
```

Here, $\mathbb{R}$ denotes the real numbers in Lean , and `lt_trans` is the general transitivity theorem for the less-than relation.

References

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Vol. 7. Cambridge Tracts in Theoretical Computer Science. Translated and adapted from the French book "Le Point Aveugle". Cambridge University Press, 1989.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Now available from `https://www.cse.chalmers.se/research/group/logic/book/book.pdf`. Oxford, UK: Oxford University Press, 1990. URL: `https://www.cse.chalmers.se/research/group/logic/book/book.pdf`.

[Tho99] Simon Thompson. *Type Theory & Functional Programming*. March 1999. Computing Laboratory, University of Kent, 1999.

[Wad15] Philip Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407. URL: `https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf`.