Contents

1 Introduction

This serves as a brief starting point for understanding how the Curry-Howard correspondence appears in Lean, as well as being an introduction to the language itslef. Lean is both a **functional programming language** and a **theorem prover**. We'll focus primarily on its role as a theorem prover. But what does this mean, and how can that be achieved?

A programming language defines a **set of rules, semantics, and syntax** for writing programs. To achieve a goal, a programmer must write a program that meets given specifications. There are two primary approaches: **program derivation** and **program verification** ([NPS90] Section 1.1). In **program verification**, the programmer first writes a program and then proves it meets the specifications. This approach checks for errors at **run-time** when the code executes. In **program derivation**, the programmer writes a proof that a program with certain properties exists, then extracts a program from that proof. This approach enables specification checking at **compilation-time**, catching errors before execution. This distinction corresponds to **dynamic** versus **static type systems**. Most programming languages combine both approaches; providing basic types for annotation and compile-time checking, while leaving the remaining checks to be performed at runtime.

Example 1.1 *In a dynamically typed language, like JavaScript, variables can change type after they are created. For example, a variable defined as a number can later be reassigned to a string:*

```
let price = 100;
price = "100";   // valid in JavaScript
```

TypeScript is a statically typed superset of JavaScript. Unlike JavaScript, it performs type checking at compile time. This means we can prevent the previous behavior simply by adding a type annotation:

```
let price: number = 100;
price = "100";  // Error: Type 'string' is not assignable to type
'number'
```

Nonetheless, TypeScript, even though it has a quite sophisticated type system, cannot fully capture complex data structures. In many cases, program specifications can only be enforced at runtime. Lean, by contrast, is fully grounded in program derivation and static type checking, which is precisely what makes

it a powerful **theorem prover**. At its core lies an advanced and flexible type system, capable of expressing and verifying a wide range of mathematical statements. This system is built on on **Type Theory**, a branch of marthematics and logic that aims to provide a foundation for all mathematics and wich is a (abstract) programming language itself.

It's important to note that Type Theory is not a single, unified theory, but rather a family of related theories with various extensions, ongoing developments and rich hiostorical ramifications. Creating a language like Lean requirev careful consideration of which rules and features to include. We shall give a brief overview of the historical development of type theory, and an a introduction on what comes next.

(The following historical intro is insbyured by [Car19] intro) Type theory emerged as a fundamental response to Bertrand Russell's paradox. Considers the set $S = \{x \mid x \notin x\}$ (the set of all sets that do not contain themselves). This is a paradoxical construction, leading to the contradiction $S \in S \iff S \notin S$. Ernst Zermelo and Fraenkel addressed the contradiction by introducing Zermelo-Fraenkel set theory (ZFC), which became the standard in modern mathematics. ZFC provides an untyped but stratified view of the mathematical universe, maintaining classical logical principles while avoiding paradoxes through careful axiomatization. Russell chose a fundamentally different path. He recognized expressions like $A(A)$ or $x \in x$ as ill-typed, introducing his theory of types. Hs's first systematic response was **Ramified Type Theory**, wich turned out to be problematic. In the 1930s, Alonzo Church developed **Lambda Calculus** as a foundation for mathematics, initially pursuing a type-free approach. However, Church's original untyped system suffered from inconsistencies similar to Russell's paradox ([Wad15]). To address these issues, Church introduced the **Simply Typed Lambda Calculus** in 1940 ([Chu40]). This system is a version of **Simple Type Theory**, a framework able to replace set-theory and propositional logic. Lambda calculus influenced the development of many programming languages as being a foundation for functional programming. Per Martin-Löf revolutionized type theory in the 1970s by introducing **dependent types** that can depend on values of other types. Think for instance of a of vectors of length n or a sequnce of n elements. **Dependent Type Theory** extends the expressive power of type systems by allowing the direct representation of predicates and quantifiers (in the sense of Frege), powerful enough to replace set theory and predicate logic." Dependent Type Theory is a derivation of **Martin-Löf Type Theory** (also known as **Intuitionistic Type Theory**). Martin-Löf's system embraced constructive (intuitionistic) principles, requiring that the existence of mathematical objects be demonstrated through explicit construction rather than classical proof by contradiction. Martin-Löf Type Theory also introduced **identity types** to represent equality. In the 1980s, Thierry Coquand and Gérard Huet introduced the **Calculus of Constructions** (CoC), synthesizing insights from Martin-Löf's dependent type theory with higher-order **polymorphism**. The Calculus of Constructions served as the theoretical foundation for the Coq proof assistant, one of the most influential interactive theorem provers. The original CoC was later extended with

inductive types to form the **Calculus of Inductive Constructions** (CIC). Inductive types allow for the definition of data structures like natural numbers, lists, and trees. The Lean theorem prover, developed by Leonardo de Moura and others, is also based on CIC but incorporates several important refinements and differences from Coq's implementation.

A central insight in type theory is the **Curry-Howard correspondence**, which establishes a profound connection between logic and computation. Also known as the **propositions-as-types** principle, this correspondence represents one of the most elegant discoveries in the foundations of mathematics and computer science. It serves also well as a good introduction to type theory, and will be used in this discussion. Nontheless it continuelsy shows new applications and interpretations in modern type theories. The Curry-Howard correspondence was independently discovered by multiple researchers. **Haskell Curry** (1934) first observed the connection between combinatory logic and Hilbert-style proof systems. **William Alvin Howard** (1969) significantly extended the correspondence to natural deduction and the simply typed lambda calculus in his seminal work "The Formulae-as-Types Notion of Construction." The correspondence was further developed through **N.G. de Bruijn's AUTOMATH system** (1967), which was the first working proof checker and demonstrated the practical viability of mechanical proof verification. Amongst its technical innovations are a discussion of the irrelevance of proofs when working in a classical context, which is one of the reasons advanced by de Bruijn for the separation between the notions of type and prop in the system [Tho99]. Lean also adopts this separation . **Per Martin-Löf's type theory** extended the correspondence to dependent types, allowing for the representation of quantifiers and identity types. Modern proof assistants like Coq, Lean, Agda, and Idris all leverage variants of the Curry-Howard correspondence to enable formal verification of mathematical theorems and software correctness properties.

2 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 **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" ($\wedge$), "or" ($\vee$), "not" ($\neg$), "implies" ($\Rightarrow$), and "if and only if" ($\Leftrightarrow$). These connectives allow the creation of complex or compound propositions.

Definition 2.1 (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:*

$$(A \wedge B), \quad (A \Rightarrow B), \quad (A \vee B), \quad \bot, \quad (A \Leftrightarrow B), \quad (\neg A)$$

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 that specifies rules for deriving **conclusions** from **premises** (assumptions from other propositions), called **inference rules**.

Example 2.2 (Deductive style rule) *Here is an hypothetical example.*

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

where the premises $P_1, P_2, \ldots, P_n$ and the conclusion C.

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 2.3 *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*

$$\begin{array}{c} A \\ \vdots \\ B \end{array}$$

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 $\langle$`_, _`$\rangle$ (underscore are placeholder for assumptions "functions"). The pair $A \wedge B$ can be then consumed using elimination rules `And.left` and `And.right`.

Example 2.4 *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 2.5 (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 $A \wedge (B \wedge C)$:

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

Example 2.6 (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 the 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. 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 fuicntion. 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 $\alpha \to$ `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 transfomred to a proposition.

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 logic. Variable capture occurs when a free variable becomes bound by a quantifier during substitution, fundamentally changing the meaning of the expression. 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 th an the free variable x" into the always-false statement "there exists some y greater than itself plus one."

To avoid capture, we must use **capture-avoiding substitution**. 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 formula 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:

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

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 2.7 *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. Roughly speaking, since we know there is an x satisfying $p\, x$, we *name* it w. If q does not mention w, then showing that q follows from $p\, w$ is tantamount to showing q follows from the existence of any such x. The existential rules can be interpreted as an infinite disjunction, so that existential elimination naturally corresponds to a *proof by cases*. In Lean, this reasoning is carried out using the `match` construct. This mechanism is known as *pattern matching*, a fundamental concept in functional programming. For example:

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

Conversely, 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
```

We will see later that functions are primitive objects in type theory. Interestingly enough a relation is expressed as ...

References

[Car19] Mario Carneiro. *The Type Theory of Lean*. Manuscript. Accessed: September 5, 2025. Apr. 2019. URL: https://github.com/digama0/lean-typetheory/.

[Chu40] Alonzo Church. *A Formulation of the Simple Theory of Types*. Vol. 5. 2. Association for Symbolic Logic, 1940, pp. 56–68.

[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.