Contents

1 Introduction

This serves as a brief starting point for understanding how mathematical proofs can be formalized in Lean, as well as being an introduction to the language itself. 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 while typing, thus, 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; // Javascript internal system recognize this
variable as a number
price = "100";   // "100" transform the number 100 to a string. This
is 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, while writing our code, simply by adding a type annotation:

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

Now converting a variable with type annotation number to a string results into a compile time error.

Nonetheless, TypeScript, even though it has a sophisticated type system, cannot fully capture complex mathematical properties. As well as for the most programming languages, program specifications can only be enforced at runtime. Lean, by contrast, uses a much more powerful type system that enables it to express and verify mathematical statements with complete rigor fully during compilation time. This makes it particularly suitable as a **theorem prover** for formalizing mathematics.

Lean's type system is based on **dependent type theory**, specifically the **Calculus of Inductive Constructions** (CIC) with various extensions. 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 historical ramifications. Creating a language like Lean requires careful consideration of which rules and features to include.

Type theory emerged as an alternative foundation for mathematics, addressing paradoxes that arose in naive set theory. Consider Russell's famous paradox: let $S = \{x \mid x \notin x\}$ be the set of all sets that do not contain themselves. This construction is paradoxical, leading to the contradiction $S \in S \iff S \notin S$. Type theory resolves such issues by working with **types** as primary objects rather than sets, and by restricting which constructions are well-formed.

Dependent type theory, the framework underlying Lean, extends basic type systems by allowing types to depend on values. For instance, one can define the type of vectors of length n, where n is itself a value. This capability makes dependent type theory particularly expressive for formalizing mathematics.

Various proof assistants have been developed based on different variants of type theory, including Agda, Coq, Idris, and Lean. Each system makes different design choices regarding which rules and features to include. Lean adopts the **Calculus of Inductive Constructions**, which extends the Calculus of Constructions, introduced in Coq, with **inductive types**. Inductive types allow for the definition of structures such as natural numbers, lists, and trees.

A fundamental design feature of Lean is its **universe hierarchy** of types, with **Prop** (the proposition type) as a distinguished universe at the bottom. The **Prop** universe exhibits two special properties: **impredicativity** and **proof irrelevance**. Proof irrelevance means that all proofs of the same proposition are considered equivalent—what matters is whether a proposition can be proven, not which specific proof is given. This separation between propositions (**Prop**) and data types (**Type**) was first introduced in **N.G. de Bruijn's AUTOMATH system** (1967) ([Tho99]). This design choice has significant practical implications for how we write and work with proofs in Lean.

For our purposes, we do not need to delve deeply into the theoretical foundations; instead, we will introduce the relevant concepts as needed while working with Lean. The practical aspects of writing proofs in Lean will be our primary focus, and the theoretical machinery will be explained only insofar as it aids understanding of how to formalize mathematics effectively.

1.1 Lean first steps

In the language of type theory, and by extension in Lean, we write `x : X` to mean that x is a **term** of type X. For example, `2 : ℕ` annotates 2 as a natural number, or more precisely, as a term of the natural number type. Lean has internally defined types such as `Nat` or $\mathbb{N}$ (you can type `\Nat` to get the Unicode symbol). The command `#check` allows us to inspect the type of any term or variable.

Example 1.2

```
#check 2 -- 2 : Nat
#check 2 + 2 -- 2 + 2 : Nat
```

[Try this example in Lean Web Editor] By following the link, you can try out the code in your browser. Lean provides a dedicated ***infoview*** *panel on the right side. Position your cursor after* `#check 2`*, and the infoview will display the output* `2 : Nat`*. This dynamic interaction, where the infoview responds to your cursor position, is what makes Lean an* ***interactive theorem prover****. As you move through your code, the infoview continuously updates, showing computations, type information, and proof states at each location.*

At first glance, one might be tempted to view the colon notation as analogous to the membership symbol $\in$ from set theory, treating types as if they were sets. While this intuition can be helpful initially, type theory offers a fundamentally richer perspective. The crucial insight is the **Curry-Howard correspondence**, also known as the **propositions-as-types** principle. This correspondence establishes a deep connection between mathematical proofs and programs: **propositions correspond to types**, and **proofs correspond to terms** inhabiting those types.

Under this interpretation, a term $x : X$ can be understood in two more ways:

- As a **computational object**: x is a program or data structure of type X
- As a **logical object**: x is a proof of the proposition X

Lean is a concrete realization of the propositions-as-types principle. In Lean, proving a theorem amounts to constructing a term of the appropriate type. When we write `theorem_name : proposition`, we are declaring that `theorem_name` is a proof (term) of `proposition` (type).

For example, consider proving that $2 + 2 = 4$:

Example 1.3

```
theorem two_plus_two_eq_four : 2 + 2 = 4 := rfl
```

Lean's syntax is designed to resemble the language of mathematics. Here, we use the `theorem` *keyword to encapsulate our proof, giving it the name* `two_plus_two_eq_four`*. This allows us to reference and reuse this result later in our code. After the semicolon* `:` *we introduce the statement* `2 + 2 = 4`*. The* `:=` *operator introduces*

the proof term that establishes the theorem's validity. The proof itself consists of a single term: `rfl` *(short for* ***reflexivity****). This is a proof term that works by* ***definitional equality****, Lean's kenerl automatically reduces both sides of the equation to their normal (definitional) form and verifies they are identical. Since* $2+2$ *computes to* 4*, the proof succeeds immediately. We can now use this theorem in subsequent proofs. For instance:*

```
example : 1 + 1 + 1 + 1 = 4 := two_plus_two_eq_four
```

Well, this example is simple enough for Lean to evaluate by itself: `1 + 1 + 1 + 1 = 2 + 2 = 4` *and conclude with* `two_plus_two_eq_four`*. Actually,* `rfl` *would solve the equation similarly, so this is just applying* `rfl` *again (it's a bit of cheating, but just for demonstration purposes). Here, I used* `example`*, which is handy for defining anonymous expressions for demonstration purposes. Before diving into the discussion, here is another keyword,* `def`*, used to introduce definitions and functions.*

```
def addOne (n : Nat) : Nat := n + 1
```

This definition expects a natural number as its parameter, written `(n : Nat)` *and returns a natural numebr. [Run in browser]*

Let's now turn to how logic is handled in Lean and how the Curry-Howard isomorphism is reflected concretely.

References

[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.