Introduction to Lean

Daan van Gent and Jesse Vogel

1 What is Lean?

Lean is an *interactive theorem prover*. Here, 'theorem prover' does not mean that Lean can automatically prove theorems, but rather that it can verify the validity of the proofs that you provide it. The word 'interactive' means that Lean can help you during the construction of a proof, by providing you the current state of the proof, giving you suggestions as to what to do next, and sometimes even automatically proving parts of the theorem for you.

Mathlib is a mathematical library written in Lean, by mathematicians of the Lean community.

There are other theorem provers, such as Coq, Aga, Isabelle, HOL (Light), Mizar, Metamath, and many more smaller projects, each with their own mathematical library.

2 Type theory

The mathematical foundation of Lean is *type theory* (rather than set theory). Type theory deals with *types*, which you can think of much like sets. Furthermore, every object in type theory is a *term* x of a unique type T, written as

$$x : T.$$

In fact, the type T is a term as well, usually of the type '`Type`', which is also denoted as '`Sort 1`'. Subsequently, '`Sort n`' is a term of type '`Sort (n + 1)`'.

Type theory consists of the following axioms.

Axiom Product type or function type. Given types A and B, we can construct the *product type*

$$\prod_{a:A} B$$

or, definitionally equivalent, the *type of functions*

$$A \to B.$$

A term $f : A \to B$ is also called a *function from A to B* (note the familiar notation!). In Lean, this type is denoted either '`Π (a : A), B`' or '`A → B`'.

Axiom Function application. Let A and B be types, and $f : A \to B$ a function from A to B. Then for all terms $a : A$, we obtain by *function application* a term

$$f(a) : B.$$

In Lean, we write simply `f a`, omitting the brackets.

In general, the operator $\to$ is non-associative. However, we will write

$$A \to B \to C := A \to (B \to C).$$

This choice is convenient for *currying*. Set theorist like to write $f_1 : A \times B \to C$ for a map that takes two inputs. Type theorists prefer $f_2 : A \to B \to C$, which is functionally equivalent. Both $f_1(a, b)$ and $f_2(a)(b)$ produce an element of C depending on a and b.

Axiom Function abstraction. Given types A and B, we can construct terms of $A \to B$ using *λ-abstraction.*

Example. For $A = B = \mathbb{N}$, we construct the function $x \mapsto x \cdot x + 3$. It is written in terms of a multiplication function and an addition function on $\mathbb{N}$. In Lean, we write '`λ (x : Nat), x * x + 3`', where '`Nat`' is the type of natural numbers.

Example. Given types A, B and C we can construct a function between $A \to B \to C$ and $B \to A \to C$ that swaps the parameters, by

```
λ (f : A → B → C), λ (b : B), λ (a : A), f a b
```

Axiom Dependent product types or Π-types. The product '`Π (a : A), B`' defined before is somewhat degenerate. Generally, the factors of the product may depend on the index term $a : A$.

Given a type A and a function $t : A \to$ `Type` we define the *dependent product type* or *Π-type*

$$\prod_{a:A} t(a).$$

In Lean, we write '`Π (a : A), t a`'.

The dependent product has analogous application and abstraction rules. Note that it was necessary to first define independent products, as t is given as a term of such a product.

Note that '`1 + 3 : Nat`' and '`2 + 2 : Nat`' denote different terms, but they should 'reduce' to the same term '`4 : Nat`'. For this, we need reduction rules.

Axiom **Conversion rules.** Any two terms which can be obtained from each other through one of the following conversion rules, are assumed to be *equal by definition*.

- α-**conversion**: renaming variables. For example, '`λ (x : Nat), x + 1`' can be converted to '`λ (y : Nat), y + 1`'.
- β-**conversion**: applying functions to their arguments. For example, '`(λ (x : Nat), x + 1) 7`' can be converted to '`7 + 1`'.
- η-**conversion**: notion of extensionality. For example, '`λ (x : Nat), f x`' can be converted to '`f`'.

Axiom **Functional extensionality.** The η-conversion rule follows from β-conversion combined with a more general axiom: *functional extensionality.* It states, as in set theory, that any two functions $f, g : A \to B$ are equal if $f(a) = g(a)$ for all $a \in A$. To state this axiom properly, we should first treat logic in type theory.

3 Logic in type theory

Logic can be done in type theory in a very surprising way. The key idea is that

logical propositions are types,
whose terms are proofs of that proposition.

This correspondence between propositions and types is known as the *Curry–Howard correspondence.* If you are still thinking about sets, you can think of a proposition as the set of all proofs of that proposition. In particular,

a proposition is true if and only if it is non-empty.

In fact, this is the definition of truth in type theory.

Now we look at how logical operations, such as $\wedge$, $\vee$, $\Rightarrow$, $\forall$, etc. are encoded in type theory.

Example. Let P and Q be propositions. Then P and Q are non-empty if and only if $P \times Q$ is non-empty. Hence, we define $P \wedge Q$ to be $P \times Q$.

Example. A proof of an implication $P \Rightarrow Q$ can be thought of as a map that sends a proof of P to a proof of Q, so we define $P \Rightarrow Q$ as $P \to Q$.

Example. Let $P(x)$ be a proposition for every $x \in X$ for some set X. An unorthodox way to write $\forall\ (x \in X),\ P(x)$ would be

$$\bigwedge_{x \in X} P(x).$$

Analogous to the binary $\wedge$ from before, this becomes a Π-type in type theory.

Example. We can encode $\top$ (true) and $\bot$ (false) as *the* types with 1 and 0 terms, respectively, which we construct later. Furthermore, it turns out to be most convenient to define $\neg P$ as $P \to \bot$.

The following table describes the correspondence for most of the important definitions.

Logic	Type theory
proposition P	type P
proof of P	term $p : P$
$P \Rightarrow Q$	$P \to Q$
$P \wedge Q$	$P \times Q$
$P \vee Q$	$P + Q$
$\forall$	Π-type
$\exists$	Σ-type
true	unit type
false	empty type
not P	$P \to$ `false`

Sum types '+' and sigma types 'Σ' will be defined in the next section.

Let's consider a definition a mathematician would like to make.

Definition (Monoid)**.** A *monoid* consists of the following data:

1. a type M
2. a map $m : M \to M \to M$
3. a term $e : M$
4. a *proof* (i.e. term) of $\forall\ (a\,b\,c : M),\ m(m(a,b),c) = m(a,m(b,c))$
5. a *proof* (i.e. term) of $\forall\ (a : M),\ m(a,e) = a \wedge m(e,a) = a$

Since propositions are *types*, we need *terms* of those types to construct monoids. This seems to introduce an issue: even if G, m and e remain unchanged, changing any of the *proofs* changes the monoid! This problem is solved in Lean by imposing the following axiom.

Axiom Propositional extensionality. Any two proofs of the same proposition are equal. Formally,

$$\forall\ (P : \texttt{Prop})\ (p\,q : P),\ p = q.$$

In particular, in the previous example, all proofs of associativity are considered equal.

4 Inductive types

By default, Lean comes with the types '`Prop`', '`Sort 1`', '`Sort 2`', '`Sort 3`', etc. and allows for the construction of (dependent) products. Every other type which is not of this form is an *inductive type*.

Inductive types are defined by a list of *constructors*. This list will be an exhaustive list of all the ways a term of this type can be constructed.

Consider the following inductive type `Weekday`.

```
inductive Weekday
| monday    : Weekday
| tuesday   : Weekday
| wednesday : Weekday
| thursday  : Weekday
| friday    : Weekday
| saturday  : Weekday
| sunday    : Weekday
```

It has seven constructors, namely `monday`, `tuesday`, etc. As we will see later, it follows from the *axiom of induction* (or *axiom of recursion*) that all terms of `Weekday` are of this form, and that they are all distinct.

The constructors of an inductive type are allowed to have inputs. For example, the integers are defined in Lean as an inductive type.

```
inductive Int
| of_nat          : Nat → Int
| neg_succ_of_nat : Nat → Int
```

That is, every '`Int`' is either of the form '`of_nat n`' or of the form '`neg_succ_of_nat n`', for some natural number '`n`'. Mathematically, we say every integer equals either n or equals $-(n+1)$, for some natural number n.

This construction can be generalized even further: inductive types are allowed to take arguments themselves as well. Given two types, we construct their disjoint union, which we call the *sum*.

```
inductive Sum (A B : Type)
| inl : A → Sum A B
| inr : B → Sum A B
```

For example, the terms of '`Sum Nat Int`' are either of the form '`inl n`' for a natural number '`n`', or '`inr i`' for an integer '`i`'.

Of course, constructors can have multiple inputs. Dual to the (binary) sum we have the (binary) product

```
inductive Product (A B : Type)
| mk : A → B → Product A B
```

whose terms are of the form '`mk a b`', where '`a : A`' and '`b : B`'.

Recall the Π-type, which given a type `A` and a map '`t : A → Type`' produces the type '`Π (a : A), t a`'. It also has a dual, the Σ-type.

```
inductive Sigma (A : Type) (t : A → Type)
| mk : Π (a : A), (t a → Sigma A t)
```

Here '`mk`' is a family of constructors indexed by the type A. Terms of '`Sigma A t`' are of the form '`mk a b`' for some '`a : A`' and '`b : t a`'.

Even though we talk about *inductive* types, so far we have not seen any induction. This comes into play when the constructor has an input of the type that we are currently defining. The natural numbers are defined as an inductive type in this way.

```
inductive Nat
| zero : Nat
| succ : Nat → Nat
```

This example may be confusing, but essentially it says that any natural number is either ‘`zero`’, or the successor ‘`succ n`’ of a natural number ‘`n`’.

One way to state the axiom of induction for the natural numbers is as follows:

Let $P(n)$ be a proposition for every natural number n. If $P(0)$ holds, and for all $n \in \mathbb{N}$ we have $P(n) \Rightarrow P(\mathrm{succ}(n))$, then $P(n)$ holds for all $n \in \mathbb{N}$.

On the other hand, we have recursive definitions of functions:

Let X be a set. Choose $f(0) \in X$, and for all $n \in \mathbb{N}$ define $f(\mathrm{succ}(n)) \in X$ in terms of n and $f(n)$. Then this determines a function $f : \mathbb{N} \to X$.

Recall that, in type theory, propositions and sets have a common generalization, namely a type. Hence, induction and recursion can be seen as special cases of the same thing.

Axiom Recursion. Every inductive type admits recursion. For the natural numbers, this comes down to the following theorem:

```
Nat.rec :
  Π (t : Nat → Sort*), t(0) → (Π (n : Nat), t(n) → t(succ(n)) →
  Π (n : Nat), t(n)
```

(For now ignore the ‘`Sort*`’, it is a fancy way to write ‘every possible type’.) To obtain *induction* from this, take ‘`t`’ to be the family of propositions ‘`P : Nat → Prop`’. This axiom of recursion then simplifies to

```
P(0) → (∀ (n : Nat), P(n) → P(succ(n)) → ∀ (n : Nat), P(n)
```

The first argument is a proof of $P(0)$, and the second a sequence of proofs of the implications $P(n) \Rightarrow P(\mathrm{succ}(n))$ for all n. The result is a proof of $\forall\ (n : \mathbb{N}),\ P(n)$.

To obtain *recursion*, take ‘`t`’ equal to the constant function ‘`λ (n : Nat), X`’, where X is the codomain of f. Then obtain the more simple

```
X → (Π (n : Nat), X → X) → Nat → X
```

The first argument will be the image of 0, and the second argument is a sequence of maps $g_n : X \to X$. The result is a map $f : \mathbb{N} \to X$ that is defined by $f(\mathrm{succ}(n)) = g_n(f(n))$.

Some final observations. Recall that we have defined the logical operator $\wedge$ as a binary product, and that we have defined the binary product as an inductive type. Axioms from logic such as $X, Y \vdash X \wedge Y$ and $X \wedge Y \vdash X$ are not axioms in type theory, but proved. The first statement is just the constructor of $\wedge$, while the second follows from recursion. Even equality '=' is an inductive type!