1 Type Theory of Lean

Lean 4 is based on a variant of the *Calculus of Inductive Constructions* (CIC), a dependent type theory that serves as both a programming language and a logical foundation for mathematics. This chapter introduces the key concepts of Lean's type system.

1.1 The Universe Hierarchy

Lean uses a hierarchy of *type universes* to avoid Russell's paradox. The fundamental universes are:

- `Prop`: The universe of propositions (also written as `Sort 0`)
- `Type u`: The universe of types at level u (equivalent to `Sort (u+1)`)
- `Sort u`: The general universe at level u

Example 1.1

In Lean:

- `Nat : Type 0`
- `Type 0 : Type 1`
- `Type 1 : Type 2`
- `2 + 2 = 4 : Prop`

Unlike some type theories, Lean's universes are *non-cumulative*: a term of type `Type u` is not automatically a term of type `Type (u+1)`.

Remark 1.1

The separation between `Prop` and `Type` is significant: `Prop` is *proof-irrelevant*, meaning two proofs of the same proposition are considered equal. This enables computational optimizations where proof terms can be erased.

1.2 Propositions as Types

Lean follows the *Curry-Howard correspondence*, which identifies:

- Propositions with types
- Proofs with terms
- Implication with function types

Definition 1.1 (Curry-Howard Isomorphism)

Under the propositions-as-types paradigm:

Logic	**Type Theory**
proposition P	type P
proof of P	term $t : P$
$P \implies Q$	function type $P \to Q$
$P \wedge Q$	product type $P \times Q$
$P \vee Q$	sum type $P \oplus Q$
$\forall x : A, P(x)$	dependent function type $\Pi(x : A), P(x)$
$\exists x : A, P(x)$	dependent pair type $\Sigma(x : A), P(x)$
$\top$ (true)	unit type `Unit`
$\perp$ (false)	empty type `Empty`

Example 1.2

The proposition "if n is even, then n^2 is even" becomes:

`Even n → Even (n^2)`

A proof is a function that transforms a proof of `Even n` into a proof of `Even (n^2)`.

1.3 Function Types and Dependent Types

1.3.1 Simple Function Types

A function type $\alpha \to \beta$ represents functions from type α to type β.

Example 1.3

- `Nat → Nat`: functions from natural numbers to natural numbers
- `Nat → Nat → Nat`: curried binary functions
- `List Nat → Nat`: functions from lists to numbers

1.3.2 Dependent Function Types (Pi Types)

A *dependent function type* allows the output type to depend on the input *value*.

Notation 1.1

The dependent function type is written:

`Π (x : α), β(x)` or `∀ (x : α), β(x)`

where β is a type family indexed by $x : \alpha$.

Example 1.4

- `∀ (n : Nat), Vec` α `n`: functions that take a natural number n and return a vector of length n
- `∀ (`α `: Type),` $\alpha \to \alpha$: the polymorphic identity function
- `∀ (n : Nat), Fin n` $\to \alpha$: functions from n-element finite sets

When the output type does *not* depend on the input value, the Pi type reduces to an ordinary function type: Π `(x :` α`),` $\beta \equiv \alpha \to \beta$.

1.4 Inductive Types

Inductive types are the primary mechanism for introducing new types in Lean. An inductive type is specified by its *constructors*.

1.4.1 Natural Numbers

Definition 1.2 (Natural Numbers)
The type `Nat` is defined inductively:

- `zero : Nat`
- `succ : Nat → Nat`

Every natural number is either `zero` or `succ n` for some `n : Nat`.

Example 1.5
In Lean notation:

- 0 is represented as `.zero`
- 1 is `.succ .zero`
- 2 is `.succ (.succ .zero)`

(Lean provides decimal notation as syntactic sugar.)

Functions on inductive types are defined by *pattern matching* and *recursion*.

Example 1.6 (Addition on Natural Numbers)
```
def add : Nat → Nat → Nat
  | m, .zero   => m
  | m, .succ n => .succ (add m n)
```

1.4.2 Product Types

Definition 1.3 (Product Type)
The product $\alpha \times \beta$ (or `Prod` α β) has one constructor:

$$\texttt{mk} : \alpha \to \beta \to \alpha \times \beta$$

We write `(a, b)` for `mk a b`.

Example 1.7
- `(3, "hello") : Nat` $\times$ `String`
- `fst` : $\alpha \times \beta \to \alpha$ extracts the first component
- `snd` : $\alpha \times \beta \to \beta$ extracts the second component

1.4.3 Sum Types

Definition 1.4 (Sum Type)
The sum $\alpha \oplus \beta$ (or `Sum` α β) has two constructors:

- `inl` : $\alpha \to \alpha \oplus \beta$ (left injection)
- `inr` : $\beta \to \alpha \oplus \beta$ (right injection)

Example 1.8
To define a function `f` : $\alpha \oplus \beta \to \gamma$, pattern match:

```
def f : α ⊕ β → γ
  | .inl a => ... -- handle case when input is from α
  | .inr b => ... -- handle case when input is from β
```

1.4.4 Lists

Definition 1.5 (List Type)
The type `List` α is defined inductively:

- `nil : List` α (empty list)
- `cons` : $\alpha \to$ `List` $\alpha \to$ `List` α (prepend element)

We write `[]` for `nil` and `h :: t` for `cons h t`.

Example 1.9
The list $[1, 2, 3]$ is encoded as:

`1 :: 2 :: 3 :: []`

1.5 The Three Kinds of Types

Every type in Lean falls into one of three categories:

1. **Function types**: $\alpha \to \beta$ or dependent Π `(x : α), β(x)`
2. **Inductive types**: Defined by constructors
   - Built-in: `Nat`, `Prop` propositions (`True`, `False`, `And`, `Or`)
   - User-defined: Custom types via `inductive` keyword
3. **Quotient types**: `Quotient R` for equivalence relation `R : α → α → Prop`
   - Models sets with equivalence (e.g., integers as equivalence classes of pairs of naturals)
   - Provides `Quot.mk : α → Quotient R` and `Quot.sound : R a b → mk a = mk b`

> **Remark 1.2**
> Quotient types are unique to Lean among proof assistants: they allow direct encoding of mathematical structures defined up to equivalence, without using setoids.

1.6 Type Class Inference

Lean uses *type classes* for ad-hoc polymorphism. A type class is a structure type marked with the `class` keyword.

Example 1.10
The type class `Add α` specifies that type α supports addition:

```
class Add (α : Type u) where
  add : α → α → α
```

Instances are registered with `instance`, and Lean's elaborator automatically finds instances during type checking.

Example 1.11
- `[Add Nat]`: instance for natural number addition
- `[Add Int]`: instance for integer addition
- `a + b` desugars to `Add.add a b`, with the instance inferred

1.7 Definitional vs. Propositional Equality

Lean distinguishes two notions of equality:

Definition 1.6 (Definitional Equality)
Two terms are *definitionally equal* (written $\equiv$) if they reduce to the same normal form by computation (beta-reduction, delta-unfolding, iota-reduction).

Definition 1.7 (Propositional Equality)
Two terms are *propositionally equal* (written `a = b : Prop`) if there exists a proof term `h : a = b`.

Example 1.12
- `2 + 2` and `4` are definitionally equal
- `n + 0` and `n` are propositionally but not definitionally equal (requires induction)

Definitional equality is checked automatically by Lean's kernel; propositional equality requires explicit proof.

1.8 Summary

The type theory of Lean provides:

- A universe hierarchy with `Prop` and `Type u`
- Propositions-as-types interpretation (Curry-Howard)
- Dependent function types (Pi types) for universal quantification
- Inductive types for data structures and propositions
- Quotient types for equivalence classes
- Type classes for overloading and inference
- Two notions of equality: definitional and propositional

This foundation enables Lean to serve simultaneously as a programming language, a theorem prover, and a formalization system for mathematics.

2 Lecture 2

$\alpha :$ Type u, $R : \alpha \to \alpha \to Prop$ is an equivalence relation on α.

$$\alpha / R : Type u$$