P04_TypeTheory

Type Theory

P04 — What is Lean, really?

You've spent four days using Lean. You've seen it as a proof assistant and as a programming language. You know its kernel is written in C++. Today: what does all of that **mean**?

S01	Types, languages, compilation	What a computer does; what types are; static vs dynamic; compilers, bootstrapping
S02	Dependent type theory	Why normal types aren't enough; dependent functions; Curry–Howard; Prop; Sort hierarchy
S03	How Lean implements this	Expr; inductives, structures, classes; Quot; Decidable
S04	Verified computation, axioms, and trust	Verified filter; the 7 axioms; `#print axioms`; de Bruijn criterion

What a computer actually does

No types. No structure. Bit patterns and opcodes.

```
CPU
registers: rax, rbx, rcx, … (16 general-purpose, each 64 bits)
fetch → decode → execute → repeat
```

↓

```
Memory (RAM)
addr 0: 01001010
addr 1: 11000011   just bytes.
addr 2: 00000010
…
```

Every abstraction — integers, strings, lists, functions — is built on top of this by humans.

Assembly — the untyped world

Assembly: human-readable mnemonics for machine instructions. An **assembler** translates them into machine code bytes.

Integer addition (x86)

```
mov  eax, 2        ; put 2 into register eax
add  eax, 3        ; add 3, result 5 in eax
```

The value 2 is just bits: `00000000 00000000 00000000 00000010`.

Assembly — the untyped world

Assembly: human-readable mnemonics for machine instructions. An **assembler** translates them into machine code bytes.

Integer addition (x86)

```
mov  eax, 2        ; put 2 into register eax
add  eax, 3        ; add 3, result 5 in eax
```

The value 2 is just bits: `00000000 00000000 00000000 00000010`.

Reinterpret the same bits as a float

```
movd   xmm0, eax     ; copy bits to float register
addss  xmm0, xmm0    ; float add on those bits
```

No error. No warning. Bits `00…010` = integer 2 = float $\approx 2.8 \times 10^{-45}$.

What is a type?

A type constrains how a piece of data is created, used, and stored:

1. What **values** can it take?
2. What **operations** are valid on it?
3. How is it **represented** in memory?

What is a type?

A type constrains how a piece of data is created, used, and stored:

1. What **values** can it take?
2. What **operations** are valid on it?
3. How is it **represented** in memory?

At the machine level, a type selects which instruction to emit:

`a + b` where `a, b : Int32` → emits **`add`** (integer add)
`a + b` where `a, b : Float64` → emits **`addsd`** (float add)

What is a type?

A type constrains how a piece of data is created, used, and stored:

1. What **values** can it take?
2. What **operations** are valid on it?
3. How is it **represented** in memory?

At the machine level, a type selects which instruction to emit:

`a + b` where `a, b : Int32` → emits **`add`** (integer add)
`a + b` where `a, b : Float64` → emits **`addsd`** (float add)

Beyond the machine: types make programs **safer** (the compiler catches misuse before you run the code) and more **convenient** (you write `+` instead of choosing **`add`** vs **`addsd`**).

Classifying type systems

When are types checked?

- **Static:** before the program runs (compile time). Compiler rejects type mismatches.
- **Dynamic:** while the program runs. Flexible, but errors surface late.
- **Untyped:** no type system at all.

How is code executed?

- **Compiled:** translated to machine code ahead of time.
- **Interpreted:** executed by another program at runtime.
- **JIT:** compiled on the fly during execution.

Classifying type systems

When are types checked?

- **Static:** before the program runs (compile time). Compiler rejects type mismatches.
- **Dynamic:** while the program runs. Flexible, but errors surface late.
- **Untyped:** no type system at all.

How is code executed?

- **Compiled:** translated to machine code ahead of time.
- **Interpreted:** executed by another program at runtime.
- **JIT:** compiled on the fly during execution.

	Compiled	JIT	Interpreted
Statically typed	C[W], Rust[S], Haskell[S], **Lean**[S]	Java[S], C#[S], Kotlin[S]	
Dynamically typed	Erlang[S]	JavaScript[W], Julia[S]	Python[S], Ruby[S], Lua[S]
Untyped	Assembly		Bash, Forth

[S] **strongly typed** (type mismatches cause errors) [W] **weakly typed** (silent coercion)

What the compiler does

Stage	Input	Output
Parsing	Source code	Abstract Syntax Tree (AST)
Type checking	AST	Annotated / validated AST
Optimization	Annotated AST	Intermediate Representation
Code generation	IR	Assembly / machine code
Linking	Object files	Executable binary

What the compiler does

Stage	Input	Output
Parsing	Source code	Abstract Syntax Tree (AST)
Type checking	AST	Annotated / validated AST
Optimization	Annotated AST	Intermediate Representation
Code generation	IR	Assembly / machine code
Linking	Object files	Executable binary

Errors are caught at the **type-checking** stage, long before any machine code is generated. A program with a type error is never compiled.

What the compiler does

Stage	Input	Output
Parsing	Source code	Abstract Syntax Tree (AST)
Type checking	AST	Annotated / validated AST
Optimization	Annotated AST	Intermediate Representation
Code generation	IR	Assembly / machine code
Linking	Object files	Executable binary

Errors are caught at the **type-checking** stage, long before any machine code is generated. A program with a type error is never compiled.

Type erasure. After compilation, types are gone — the machine code is untyped. Types cost nothing at runtime in compiled languages (C, Rust, Lean: 0 bytes overhead). Dynamic languages carry types on every object (Python `int`: 28 bytes vs. C `int`: 4 bytes).

The bootstrap problem

A compiler translates source code into machine code. But the compiler itself is a program. What compiled the compiler?

Step	Written in	Compiled by
First assembler	Hand-written machine code	(toggling bytes directly)
B compiler (1969)	Assembly	Assembler
C compiler (1973)	B	B compiler
C compiler v2	C	C compiler v1 ← self-hosting
…	C	previous C compiler

History. B and C were developed by Ken Thompson and Dennis Ritchie at Bell Labs. C became self-hosting in 1973.

The bootstrap problem

A compiler translates source code into machine code. But the compiler itself is a program. What compiled the compiler?

Step	Written in	Compiled by
First assembler	Hand-written machine code	(toggling bytes directly)
B compiler (1969)	Assembly	Assembler
C compiler (1973)	B	B compiler
C compiler v2	C	C compiler v1 ← self-hosting
…	C	previous C compiler

History. B and C were developed by Ken Thompson and Dennis Ritchie at Bell Labs. C became self-hosting in 1973.

> *"You can't trust code that you did not totally create yourself. […] No amount of source-level verification or scrutiny will protect you from using untrusted code."*
>
> — Ken Thompson, *Reflections on Trusting Trust* (1983)

Lean's bootstrap

Lean is **largely self-hosting**: the parser, elaborator, tactics, and compiler are written in Lean. The kernel is C++. The trusted base is small:

1. A **kernel** type-checks every definition and proof.
2. Everything else — parser, elaborator, tactics, compiler — is Lean code, checked by that kernel.
3. When Lean accepts your proof, the check depends only on this small trusted core.

Lean's bootstrap

Lean is **largely self-hosting**: the parser, elaborator, tactics, and compiler are written in Lean. The kernel is C++. The trusted base is small:

1. A **kernel** type-checks every definition and proof.
2. Everything else — parser, elaborator, tactics, compiler — is Lean code, checked by that kernel.
3. When Lean accepts your proof, the check depends only on this small trusted core.

Bootstrap entry point: `stage0/` in Lean's repo contains pre-generated C files. Anyone can compile them with a C compiler and bootstrap Lean from scratch.

The trust chain

Directory	Language	Role
`stage0/src/`	Auto-gen. C	Pre-compiled Lean frontend; bootstrap entry point
`src/kernel/`	C++	Trusted kernel: type checker, inductives, quotients
`src/runtime/`	C++	Memory management, IO, object representation

When you run any program, you implicitly trust:

1. **CPU hardware** — silicon correctly implements the instruction set
2. **Operating system** — loads and runs your binary faithfully
3. **Your compiler** — source you can read and, in principle, verify
4. **The bootstrap chain** — pre-compiled binaries from previous versions, not independently verifiable from source alone

Where this leaves us

1. **Types** are metadata that make programs safer and guide the compiler — from basic (`int` vs `float`) to expressive (Rust's ownership, Haskell's parametric polymorphism).
2. **Compilers are bootstrapped** — the trust chain goes back to hardware, and you always build on prior work.
3. **Type checking** validates your program and provides the information the compiler needs to generate correct machine code. In Lean, this same mechanism checks proofs.

Where this leaves us

1. **Types** are metadata that make programs safer and guide the compiler — from basic (`int` vs `float`) to expressive (Rust's ownership, Haskell's parametric polymorphism).
2. **Compilers are bootstrapped** — the trust chain goes back to hardware, and you always build on prior work.
3. **Type checking** validates your program and provides the information the compiler needs to generate correct machine code. In Lean, this same mechanism checks proofs.

Next: What happens when the return type of a function is allowed to depend on the input value?

Type Theory — Contents

Types and Languages

Dependent Type Theory

Lean's Implementation

Verified Computation and Trust

The limits of ordinary types

In C, Rust, Haskell, Java — every typed language you know — the return type of a function is fixed at definition time. It cannot mention the argument's value.

```
length : List a -> Int
```
return type is always `Int`

```
get : List a -> Int -> Maybe a
```
return type is always `Maybe a`

The limits of ordinary types

In C, Rust, Haskell, Java — every typed language you know — the return type of a function is fixed at definition time. It cannot mention the argument's value.

```
length : List a -> Int
```
return type is always `Int`

```
get : List a -> Int -> Maybe a
```
return type is always `Maybe a`

What you **want** to say but **cannot**:

```
get : (xs : List a) -> ValidIndex xs -> a
```

"Give me a list and a proof that your index is in bounds, and I **guarantee** I return an element — no `Maybe`, no runtime crash."

The limits of ordinary types

In C, Rust, Haskell, Java — every typed language you know — the return type of a function is fixed at definition time. It cannot mention the argument's value.

```
length : List a -> Int
```
return type is always `Int`

```
get : List a -> Int -> Maybe a
```
return type is always `Maybe a`

What you **want** to say but **cannot**:

```
get : (xs : List a) -> ValidIndex xs -> a
```

"Give me a list and a proof that your index is in bounds, and I **guarantee** I return an element — no `Maybe`, no runtime crash."

The return type mentions `xs`. Ordinary type systems cannot do this.

Dependent function types

Allow the return type to depend on the input value:

Ordinary: `Nat -> Bool` (return type fixed)

Dependent: `(n : Nat) -> Fin (n + 1)` (return type mentions `n`)

Dependent function types

Allow the return type to depend on the input value:

Ordinary: `Nat -> Bool` (return type fixed)

Dependent: `(n : Nat) -> Fin (n + 1)` (return type mentions `n`)

Safe indexing

`(xs : List a) -> Fin xs.length -> a` — no out-of-bounds possible

Length-indexed vectors

`Vec a n` — a list that knows its length in the type

Dimension-matched matrices

`mul : Mat m n -> Mat n p -> Mat m p` — compiler rejects mismatches

Format strings

The type of `printf` depends on the format argument

Curry–Howard: propositions are types

Logic	Type theory (mechanism)
Proposition P	Type P
Proof of P	Term $t : P$
$P \to Q$ (implication)	function type (Pi with constant codomain)
$P \wedge Q$	inductive with one constructor, two fields
$P \vee Q$	inductive with two constructors
$\neg P$	$P \to$ `False`
$\forall x,\ P(x)$	dependent function (Pi type)
$\exists x,\ P(x)$	inductive with one constructor: witness + proof

Curry–Howard: propositions are types

Logic	Type theory (mechanism)
Proposition P	Type P
Proof of P	Term $t : P$
$P \to Q$ (implication)	function type (Pi with constant codomain)
$P \wedge Q$	inductive with one constructor, two fields
$P \vee Q$	inductive with two constructors
$\neg P$	$P \to$ `False`
$\forall x,\ P(x)$	dependent function (Pi type)
$\exists x,\ P(x)$	inductive with one constructor: witness + proof

Type-checking is proof-checking. The same kernel that validates your programs validates your proofs. **No separate logic needed.**

The problem with proofs-as-data

In the Curry–Howard picture, a proof is a term — a piece of data. Two different proofs of $2 \leq 4$ are two different terms, just like `3` and `7` are different `Nat`s.

The problem with proofs-as-data

In the Curry–Howard picture, a proof is a term — a piece of data. Two different proofs of $2 \leq 4$ are two different terms, just like `3` and `7` are different `Nat`s.

This creates two problems:

1. **Programs could branch on proofs.** A function could inspect *which* proof you supplied and return different results. Mathematically absurd.
2. **Proofs cannot be erased.** If programs can observe proofs, the compiler must keep them at runtime.

The problem with proofs-as-data

In the Curry–Howard picture, a proof is a term — a piece of data. Two different proofs of $2 \leq 4$ are two different terms, just like `3` and `7` are different `Nat`s.

This creates two problems:

1. **Programs could branch on proofs.** A function could inspect *which* proof you supplied and return different results. Mathematically absurd.
2. **Proofs cannot be erased.** If programs can observe proofs, the compiler must keep them at runtime.

> We need a universe where all proofs of the same statement are **interchangeable** — definitionally equal. That universe is `Prop`.

Prop: proof irrelevance

`Prop` is a special universe where any two proofs of the same proposition are **definitionally equal**. The kernel does not even compare them.

Prop: proof irrelevance

`Prop` is a special universe where any two proofs of the same proposition are **definitionally equal**. The kernel does not even compare them.

1. **No branching on proofs.** The kernel forbids eliminating a multi-constructor `Prop` type (like `Or`) into `Type`. If it allowed this, a function could produce different outputs for proofs that are supposed to be equal — inconsistency.
2. **Erasure follows.** Because no computation can distinguish proofs, the compiler safely deletes them. Zero runtime cost.

Prop: proof irrelevance

`Prop` is a special universe where any two proofs of the same proposition are **definitionally equal**. The kernel does not even compare them.

1. **No branching on proofs.** The kernel forbids eliminating a multi-constructor `Prop` type (like `Or`) into `Type`. If it allowed this, a function could produce different outputs for proofs that are supposed to be equal — inconsistency.
2. **Erasure follows.** Because no computation can distinguish proofs, the compiler safely deletes them. Zero runtime cost.

> Proof irrelevance is a **type-theoretic** rule, not an optimization. Erasure is a consequence, not the cause. Even without a compiler, the kernel would reject proof-dependent branching.

Sort hierarchy: avoiding paradox

If `Type` were its own type (`Type : Type`), you could encode Girard's paradox — the type-theoretic analogue of Burali-Forti's paradox (the "ordinal of all ordinals").

Sort hierarchy: avoiding paradox

If `Type` were its own type (`Type : Type`), you could encode Girard's paradox — the type-theoretic analogue of Burali-Forti's paradox (the "ordinal of all ordinals").

Solution: **stratified universes**. Each universe lives in the next one.

Sort	Notation	Contains
`Sort 0`	`Prop`	proof-irrelevant propositions
`Sort 1`	`Type`	ordinary data types (`Nat`, `List`, …)
`Sort 2`	`Type 1`	the type that contains `Type`
…	…	…

Sort hierarchy: avoiding paradox

If `Type` were its own type (`Type : Type`), you could encode Girard's paradox — the type-theoretic analogue of Burali-Forti's paradox (the "ordinal of all ordinals").

Solution: **stratified universes**. Each universe lives in the next one.

Sort	Notation	Contains
`Sort 0`	`Prop`	proof-irrelevant propositions
`Sort 1`	`Type`	ordinary data types (`Nat`, `List`, …)
`Sort 2`	`Type 1`	the type that contains `Type`
…	…	…

`Prop` is **impredicative**: $\forall$ `α : Type, α -> α` stays in `Prop` when the conclusion is a proposition. Higher universes are predicative: quantifying over `Type` pushes you to `Type 1`.

Where this leaves us (S02)

1. **Dependent function types** let return types mention input values — this is what makes the type system expressive enough for specifications and proofs.
2. **Curry–Howard** means propositions are types and proofs are terms. No separate logic — the same type-checker handles both.
3. **Prop** makes proofs interchangeable and erasable. **Sort hierarchy** prevents paradox.

Where this leaves us (S02)

1. **Dependent function types** let return types mention input values — this is what makes the type system expressive enough for specifications and proofs.
2. **Curry–Howard** means propositions are types and proofs are terms. No separate logic — the same type-checker handles both.
3. **Prop** makes proofs interchangeable and erasable. **Sort hierarchy** prevents paradox.

Next: How does Lean actually implement all of this? What data structures and mechanisms make dependent type theory concrete?

Let’s look at this in Lean.

Type Theory — Contents

Types and Languages

Dependent Type Theory

Lean’s Implementation

Verified Computation and Trust

Expr: Lean's term language

Everything you write — definitions, theorems, tactic proofs — is elaborated into a single data type called `Expr`. The kernel type-checks `Expr` trees and nothing else.

#	Constructor	Role
1	`bvar`	bound variable (de Bruijn index)
2	`fvar`	free variable
3	`mvar`	metavariable (hole for elaboration)
4	`sort`	universes (`Prop`, `Type`, `Type 1`, …)
5	`const`	reference to a named declaration
6	`app`	function application
7	`lam`	lambda abstraction
8	`forallE`	**dependent function type (Pi)**
9	`letE`	let-binding
10	`lit`	literal shortcut (Nat/String)
11	`mdata`	metadata (kernel ignores)
12	`proj`	structure projection shortcut

Source: `src/kernel/expr.h` (C++) and `src/Lean/Expr.lean` (Lean).

Definitional equality: how the kernel decides sameness

The kernel must decide whether two `Expr` trees represent the same term. It does this by **reducing** both sides and comparing the results.

Rule	What it does
β	apply lambda: `(fun x => b) a` reduces to `b[x := a]`
δ	unfold a `def` (replace name with its body)
ι	reduce a pattern match on a known constructor
ζ	inline a `let`-binding
η	`fun x => f x` equals `f`
proof irrel.	any two proofs of the same `Prop` are equal

Definitional equality: how the kernel decides sameness

The kernel must decide whether two `Expr` trees represent the same term. It does this by **reducing** both sides and comparing the results.

Rule	What it does
β	apply lambda: `(fun x => b) a` reduces to `b[x := a]`
δ	unfold a `def` (replace name with its body)
ι	reduce a pattern match on a known constructor
ζ	inline a `let`-binding
η	`fun x => f x` equals `f`
proof irrel.	any two proofs of the same `Prop` are equal

`theorem` bodies are checked but **never unfolded** — they are opaque to definitional equality. This is what distinguishes `theorem` from `def` in the kernel.

Inductive types: how Lean builds types

The kernel's environment maps names to declarations. An **inductive declaration** specifies constructors; the kernel validates them and generates a **recursor** (the primitive elimination principle).

Inductive types: how Lean builds types

The kernel's environment maps names to declarations. An **inductive declaration** specifies constructors; the kernel validates them and generates a **recursor** (the primitive elimination principle).

1. Almost every type you've seen is an inductive: `Nat`, `Bool`, `List`, `Eq`, `And`, `Or`, `True`, `False`, ...
2. A **structure** is an inductive with exactly one constructor. Lean generates named projections (`.x`, `.y`).
3. A **class** is a structure where `[...]` binders trigger automatic instance search. Same mechanism, one extra feature.

Inductive types: how Lean builds types

The kernel's environment maps names to declarations. An **inductive declaration** specifies constructors; the kernel validates them and generates a **recursor** (the primitive elimination principle).

1. Almost every type you've seen is an inductive: `Nat`, `Bool`, `List`, `Eq`, `And`, `Or`, `True`, `False`, ...
2. A **structure** is an inductive with exactly one constructor. Lean generates named projections (`.x`, `.y`).
3. A **class** is a structure where `[...]` binders trigger automatic instance search. Same mechanism, one extra feature.

One mechanism, many faces. Data types, logical connectives, algebraic structures — all inductives with generated recursors.

Quot: the one exception

`Quot` is the only type that is **not** an inductive. It is a kernel primitive with four constants and one axiom.

Quot: the one exception

`Quot` is the only type that is **not** an inductive. It is a kernel primitive with four constants and one axiom.

What it does: given a type α and a relation `r : α -> α -> Prop`, `Quot r` identifies elements related by `r` (e.g., $\mathbb{Z}$ as $\mathbb{N} \times \mathbb{N}$ modulo equivalence).

Quot: the one exception

`Quot` is the only type that is **not** an inductive. It is a kernel primitive with four constants and one axiom.

What it does: given a type α and a relation `r : α -> α -> Prop`, `Quot r` identifies elements related by `r` (e.g., $\mathbb{Z}$ as $\mathbb{N} \times \mathbb{N}$ modulo equivalence).

Why it can't be an inductive:

1. `Quot.lift f h (Quot.mk r a)` must reduce to `f a` — a custom reduction rule hardcoded in the kernel. User-defined inductives cannot add reduction rules.
2. `Quot.sound` says related elements become equal: this violates constructor injectivity, which all ordinary inductives have.

`Quot.sound` is one of Lean's three mathematical axioms (more in S04).

Decidable: bridging Prop and Bool

`Bool` lives in `Type` — runtime data, two values, decidable checks.
`Prop` lives in `Sort 0` — erased, can express undecidable statements.

`true`/`false` are `Bool` constructors.

`True`/`False` are `Prop`-valued inductives. Completely unrelated.

Decidable: bridging Prop and Bool

`Bool` lives in `Type` — runtime data, two values, decidable checks.
`Prop` lives in `Sort 0` — erased, can express undecidable statements.

`true`/`false` are `Bool` constructors.

`True`/`False` are `Prop`-valued inductives. Completely unrelated.

`Decidable p` bridges them: for a proposition `p`, it provides a runtime tag (`isTrue`/`isFalse`) that `if` can branch on. Each instance is an algorithm; the proof is erased.

```
class inductive Decidable (p : Prop)
  | isFalse (h : ¬p)
  | isTrue  (h : p)
```

Decidable: bridging Prop and Bool

`Bool` lives in `Type` — runtime data, two values, decidable checks.
`Prop` lives in `Sort 0` — erased, can express undecidable statements.

`true`/`false` are `Bool` constructors.

`True`/`False` are `Prop`-valued inductives. Completely unrelated.

`Decidable p` bridges them: for a proposition `p`, it provides a runtime tag (`isTrue`/`isFalse`) that `if` can branch on. Each instance is an algorithm; the proof is erased.

```
class inductive Decidable (p : Prop)
  | isFalse (h : ¬p)
  | isTrue  (h : p)
```

`[DecidablePred p]` lets you write `if p x then ... else ...` with a `Prop`-valued predicate. The tag is kept; the proof is erased.

Large elimination: when Prop targets Type

Proof irrelevance means all proofs of the same `Prop` are equal. So a recursor from a `Prop` type normally can only produce `Prop` results — otherwise “equal” proofs could yield different data.

Large elimination: when Prop targets Type

Proof irrelevance means all proofs of the same `Prop` are equal. So a recursor from a `Prop` type normally can only produce `Prop` results — otherwise "equal" proofs could yield different data.

Exception: if a `Prop`-valued inductive has exactly one constructor and every field is in `Prop`, there is at most one proof. The recursor can safely return `Type`-level results — no axiom needed.

Large elimination: when Prop targets Type

Proof irrelevance means all proofs of the same `Prop` are equal. So a recursor from a `Prop` type normally can only produce `Prop` results — otherwise "equal" proofs could yield different data.

Exception: if a `Prop`-valued inductive has exactly one constructor and every field is in `Prop`, there is at most one proof. The recursor can safely return `Type`-level results — no axiom needed.

1. `Eq` lives in `Prop`, has one constructor (`refl`). So `Eq.rec` can produce data: given `h : a = b` and `F a`, it produces `F b`. This is what `rw` does.
2. `Or` has two constructors — a recursor into `Type` could tell which branch was used. So `Or.rec` can only target `Prop`.

Decidable bridges Prop and Type by living in `Type`. Large elimination bridges from the other side: certain `Prop` types directly produce `Type` results.

Where this leaves us (S03)

1. **Expr** (12 constructors) is the kernel's only term language. `forallE` makes it dependent.
2. **Inductives** define almost every type. The kernel generates recursors. Structures and classes are special cases.
3. **Quot** is the one kernel primitive that is not an inductive — needed for quotient types and `funext`.
4. **Decidable** bridges `Prop` and `Bool`. **Large elimination** provides a constructive escape for subsingleton `Prop` types.

Where this leaves us (S03)

1. **Expr** (12 constructors) is the kernel's only term language. `forallE` makes it dependent.
2. **Inductives** define almost every type. The kernel generates recursors. Structures and classes are special cases.
3. **Quot** is the one kernel primitive that is not an inductive — needed for quotient types and `funext`.
4. **Decidable** bridges `Prop` and `Bool`. **Large elimination** provides a constructive escape for subsingleton `Prop` types.

Next: Putting it all together — verified computation, Lean's axioms, and what you actually trust.

Let’s look at this in Lean.

Type Theory — Contents

Types and Languages

Dependent Type Theory

Lean's Implementation

Verified Computation and Trust

Subtype, Nonempty, Inhabited

Three types that show the `Prop`/`Type` boundary in practice:

Type	Universe	Witness	At runtime
Subtype `{x : α // p x}`	`Type`	value + proof	proof erased
`Nonempty α`	`Prop`	trapped	erased entirely
`Inhabited α`	`Type`	accessible	kept

Subtype, Nonempty, Inhabited

Three types that show the `Prop`/`Type` boundary in practice:

Type	Universe	Witness	At runtime
Subtype `{x : α // p x}`	`Type`	value + proof	proof erased
`Nonempty` α	`Prop`	trapped	erased entirely
`Inhabited` α	`Type`	accessible	kept

1. `Subtype` bundles data with a proof. Proof erased at runtime — zero overhead.
2. `Nonempty` α: "α has an element" but won't give you one. Witness in `Prop` — trapped.
3. `Nonempty` to `Inhabited` requires `Classical.choice` — the only Prop-to-Type axiom.

Verified computation: the pattern

1. **Algorithm:** write a function with a plain type signature.
   `propFilter (p : α -> Prop) [DecidablePred p] : List α -> List α`
2. **Proof:** prove properties as separate theorems.
   `propFilter_sound : ∀ x ∈ propFilter p xs, p x`
3. **Bundle:** return type carries the guarantee via `Subtype`.
   `verifiedFilter : { ys : List α // ∀ x ∈ ys, p x }`

Verified computation: the pattern

1. **Algorithm:** write a function with a plain type signature.
   `propFilter (p : α -> Prop) [DecidablePred p] : List α -> List α`
2. **Proof:** prove properties as separate theorems.
   `propFilter_sound : ∀ x ∈ propFilter p xs, p x`
3. **Bundle:** return type carries the guarantee via `Subtype`.
   `verifiedFilter : { ys : List α // ∀ x ∈ ys, p x }`

Same runtime code. Proofs are erased — zero overhead. The type signature is the specification, and the kernel has verified it.

Let’s look at this in Lean.

What is an axiom?

A `theorem` has a proof term — the kernel checks it.
An `axiom` has no proof term — the kernel records it unchecked.

What is an axiom?

A `theorem` has a proof term — the kernel checks it.
An `axiom` has no proof term — the kernel records it unchecked.

```
axiom myAxiom : ∀ (P : Prop), P
```

From this, everything is provable: `myAxiom False : False`. An inconsistent axiom destroys the entire system.

What is an axiom?

A `theorem` has a proof term — the kernel checks it.
An `axiom` has no proof term — the kernel records it unchecked.

```
axiom myAxiom : ∀ (P : Prop), P
```

From this, everything is provable: `myAxiom False : False`. An inconsistent axiom destroys the entire system.

> `sorry` introduces the axiom `sorryAx` — Lean warns you. Use it as a placeholder, never in finished work.

Lean's three mathematical axioms

Axiom	What it says
`propext`	Logically equivalent propositions are equal: $(a \leftrightarrow b) \to a = b$
`Quot.sound`	Related elements have equal equivalence classes
`Classical.choice`	Extract a witness from `Nonempty` α (the only Prop-to-Type axiom)

Lean's three mathematical axioms

Axiom	What it says
`propext`	Logically equivalent propositions are equal: $(a \leftrightarrow b) \to a = b$
`Quot.sound`	Related elements have equal equivalence classes
`Classical.choice`	Extract a witness from `Nonempty` α (the only Prop-to-Type axiom)

Derived (not axioms themselves):

- `funext`: from `Quot.sound` alone
- `Classical.em` (excluded middle): from all three

Lean's three mathematical axioms

Axiom	What it says
`propext`	Logically equivalent propositions are equal: $(a \leftrightarrow b) \to a = b$
`Quot.sound`	Related elements have equal equivalence classes
`Classical.choice`	Extract a witness from `Nonempty` α (the only Prop-to-Type axiom)

Derived (not axioms themselves):

- `funext`: from `Quot.sound` alone
- `Classical.em` (excluded middle): from all three

`#print axioms myTheorem` shows which axioms any theorem depends on. A theorem using none is fully constructive.

The full axiom landscape

Lean's `Init` declares seven `axiomInfo` entries:

Axiom	Purpose	Category
`propext`	proposition equality	mathematical
`Quot.sound`	quotient soundness	mathematical
`Classical.choice`	witness extraction	mathematical
`sorryAx`	proof holes	debugging
`Lean.ofReduceBool`	trust compiled Bool reduction	compilation
`Lean.ofReduceNat`	trust compiled Nat reduction	compilation
`Lean.trustCompiler`	compiler correctness	compilation

False.elim: a constructive consequence

`False` is a type with **zero constructors**. There is no way to build a term of type `False`.

False.elim: a constructive consequence

`False` is a type with **zero constructors**. There is no way to build a term of type `False`.

If you have a proof of `False`, you can derive anything:

`False.elim : False -> C` for any type `C`

False.elim: a constructive consequence

`False` is a type with **zero constructors**. There is no way to build a term of type `False`.

If you have a proof of `False`, you can derive anything:

`False.elim : False -> C` for any type `C`

1. This is **not** an axiom. It follows mechanically from the recursor: to handle all cases of `False`, you handle... nothing. So you get a function to any type for free.
2. Negation ¬`P` is defined as `P -> False` — a function that, if you could ever call it, would give you anything.
3. `#print axioms False.elim` shows no axioms — fully constructive.

Trust: the de Bruijn criterion

Layer	What	Trust?
C++ kernel	≈ 8,000 lines: type-checks every term	**trusted**
Axioms	3 mathematical + 4 compilation	**trusted**
Hardware + OS	executes the checker	**trusted**
Elaborator	parses, infers, produces `Expr`	untrusted
Tactics	`simp`, `omega`, `ring`, …	untrusted
Mathlib	every theorem	untrusted

Untrusted = the kernel re-checks everything they produce.

Trust: the de Bruijn criterion

Layer	What	Trust?
C++ kernel	≈ 8,000 lines: type-checks every term	**trusted**
Axioms	3 mathematical + 4 compilation	**trusted**
Hardware + OS	executes the checker	**trusted**
Elaborator	parses, infers, produces `Expr`	untrusted
Tactics	`simp`, `omega`, `ring`, …	untrusted
Mathlib	every theorem	untrusted

Untrusted = the kernel re-checks everything they produce.

External re-checkers (`lean4checker`, `lean4lean`) validate exported proof terms against an independent implementation — without trusting Lean's own elaborator.

Trust: the de Bruijn criterion

Layer	What	Trust?
C++ kernel	≈ 8,000 lines: type-checks every term	**trusted**
Axioms	3 mathematical + 4 compilation	**trusted**
Hardware + OS	executes the checker	**trusted**
Elaborator	parses, infers, produces `Expr`	untrusted
Tactics	`simp`, `omega`, `ring`, …	untrusted
Mathlib	every theorem	untrusted

Untrusted = the kernel re-checks everything they produce.

External re-checkers (`lean4checker`, `lean4lean`) validate exported proof terms against an independent implementation — without trusting Lean's own elaborator.

> You do not trust Lean. You do not trust Mathlib. You do not trust tactics. You trust a small, inspectable kernel and three axioms.