Proof and Countermodel

An Introduction to Logic with Lean

Flyxion
Independent Researcher

2026

Preface

This book teaches logic through the act of getting Lean to accept or reject a claim. It does not present logic as an abstract preliminary to be endured before the real, practical work of programming begins; the formal statement, the tactic script, and Lean's response together constitute the lesson.

Each unit ends with a small number of experiments: typically one successful proof and one or two countermodels demonstrating why a plausible-looking alternative fails. A reader can learn as much from discovering exactly why a converse refuses to typecheck as from completing a valid proof. This pairing is not a pedagogical decoration; it is the book's organizing method, carried through consistently from the first chapter to the last.

The six chapters that follow form a complete, self-contained introduction: propositions and connectives, natural deduction read directly through Lean's tactics, the constructive/classical boundary made observable rather than historical, quantifiers and their scope, relations and orders, and elementary set theory treated as applied logic. A further, more ambitious volume, *Inference Under Constraint*, extends this method into non-classical logic, modality, and a broader admissibility-theoretic program; that volume assumes this one but is not required to complete it. This book stands on its own.

No prior experience with Lean, type theory, or formal proof is assumed. Some comfort with ordinary mathematical notation is useful but not essential — the notation is introduced as it is needed.

Contents

Part I

Foundations

Chapter 1

Propositions and Connectives

This chapter introduces the basic propositional connectives $\wedge$, $\vee$, $\neg$, $\to$, and $\leftrightarrow$, together with Lean's `Bool`-valued truth-table decidability through the `decide` tactic. It establishes, in miniature, the pairing that organizes the whole book: a claim is proved where it holds, and a plausible-looking converse or strengthening is shown false by an explicit countermodel where it does not.

1.1 The Basic Connectives

[TODO: introduce $\wedge$, $\vee$, $\neg$, $\to$, $\leftrightarrow$ as `Prop`-valued Lean expressions; contrast with their `Bool`-valued truth-table counterparts used for the `decide`-based experiments below.]

1.2 A Worked Pair: Disjunction Introduction

The first genuine result of the book is small on purpose: it should be possible to state, prove, and understand the countermodel to its converse within a single sitting.

Experiment: Disjunction introduction

For any propositions p and q, $p \to p \vee q$.

```
theorem or_intro_left (p q : Prop) (hp : p) : p ∨ q :=
  Or.inl hp
```

The proof is definitional: `Or.inl` is literally the introduction rule for the left disjunct. Nothing about q is used or needed.

Countermodel: The converse of disjunction introduction

The converse, $p \vee q \to p$, is not valid: it fails whenever q holds and p does not.

```
example : ¬ ∀( p q : Bool, (p || q) = true → p = true) := by
  decide
```

Working at the level of `Bool` rather than `Prop` makes the countermodel a finite, decidable check: Lean simply enumerates the four truth-value combinations and exhibits $p =$ false, $q =$ true as a witness where the premise holds and the conclusion fails.

1.3 De Morgan, Contraposition, Distributivity

[TODO: three further experiment/countermodel pairs — De Morgan's laws in both directions, contraposition, and distributivity of $\wedge$ over $\vee$ (and the reverse). Each should follow the pattern above: a `Prop`-level proof via natural deduction, paired where appropriate with a `Bool`-level `decide` countermodel to a plausible but invalid strengthening.]

1.4 Application: Digital Logic Circuits

Propositional logic is not only a tool for evaluating arguments; it is, quite literally, the specification language for digital hardware. An AND gate, an OR gate, and a NOT gate compute $\wedge$, $\vee$, and $\neg$ on electrical signals instead of truth values, and a circuit built from them computes whatever Boolean expression its wiring diagram represents. This gives the truth tables and `decide` calls from earlier in the chapter a second life: two circuits that look completely different can be shown to compute the exact same function by checking that their Boolean expressions agree on every input — which is exactly what `decide` already does.

Experiment: Reading a circuit off a truth table

Given a truth table, a Boolean expression that matches it can always be built directly: for each row where the output is true, form the conjunction of each input (negated if that input is false in that row), then join all such conjunctions with $\vee$. For the table

P	Q	R	S
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

the three true rows give $S = (P \wedge Q \wedge R) \vee (P \wedge \neg Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R)$. Rather than trust that construction by eye, state the table directly as one function and the built expression as another, then let `decide` confirm they agree everywhere:

```
def sFromTable (p q r : Bool) : Bool :=
  match p, q, r with
  | true,  true,  true  => true
  | true,  true,  false => false
  | true,  false, true  => true
  | true,  false, false => true
  | false, true,  true  => false
  | false, true,  false => false
  | false, false, true  => false
  | false, false, false => false

def sFromDNF (p q r : Bool) : Bool :=
  (p && q && r) || (p && !q && r) || (p && !q && !r)

example : ∀ p q r : Bool, sFromTable p q r = sFromDNF p q r := by
  decide
```

Experiment: Two circuits, one function

Circuits with visibly different wiring can compute identical functions — the point of a later stage in circuit design is finding the simpler one. A circuit built from an OR of two ANDs need be no more powerful than a single wire:

```
def circuitA (p q : Bool) : Bool := (p && q) || (p && !q)
def circuitB (p q : Bool) : Bool := p

example : ∀ p q : Bool, circuitA p q = circuitB p q := by decide
```

`circuitA` and `circuitB` disagree on how many gates they use and agree on every output — the same distinction between syntactic form and semantic content that separates a Lean proof term from the proposition it proves.

1.5 Exercises

Exercise 1.1. Prove $q \to p \vee q$ (disjunction introduction, right).

Exercise 1.2. State and prove, or refute by countermodel, the claim $(p \to q) \to (\neg q \to \neg p)$.

Exercise 1.3. Using `decide` on `Bool`, determine whether $(p \wedge q) \vee r$ and $(p \vee r) \wedge (q \vee r)$ always agree.

Chapter 2

Natural Deduction as Tactics

This chapter reads Lean's core tactics directly as introduction and elimination rules, rather than as arbitrary programming instructions to be memorized separately from the logic they express. By the end of the chapter, the tactic script and the natural-deduction proof should be visibly the same object viewed two ways.

2.1 Introduction Rules: `intro` and `constructor`

[TODO: `intro` as $\to$-introduction and $\forall$-introduction; `constructor` as $\wedge$-introduction and $\exists$-introduction. Worked experiment: a short conjunction proof built with `constructor`, shown side by side with its natural-deduction tree.]

2.2 Elimination Rules: `apply`, `exact`, `cases`

[TODO: `apply`/`exact` as $\to$-elimination (modus ponens); `cases` as $\vee$-elimination and $\exists$-elimination. Worked experiment: a disjunction-elimination proof via `cases`.]

Experiment: Modus ponens as `apply`

[TODO — state and prove a short chain of implications using `apply` at each step, annotating each tactic call with the elimination rule it corresponds to.]

2.3 Exercises

Exercise 2.1. Prove $(p \wedge q) \to (q \wedge p)$ using `constructor` and `exact`.

Exercise 2.2. Prove $(p \vee q) \to (q \vee p)$ using `cases`.

Chapter 3

Classical and Constructive Logic

This chapter makes the constructive/classical boundary a concrete, observable formal fact rather than historical or terminological background: one direction of double negation type-checks without qualification, and the other genuinely requires a classical principle to be invoked by name.

3.1 Double Negation Introduction

Experiment: $p \to \neg\neg p$

```
theorem dn_intro (p : Prop) (hp : p) : ¬¬p :=
  fun hnp => hnp hp
```

This direction is constructively valid: it requires nothing beyond the definition of negation as $p \to$ False.

3.2 Double Negation Elimination

Countermodel: $\neg\neg p \to p$ is not constructively provable

[TODO: show that a direct constructive proof attempt of $\neg\neg p \to p$ has no closing tactic, then prove it using `Classical.byContradiction` or `Classical.em`, naming explicitly which classical principle was invoked and why none of the purely constructive tactics from Chapter 2 suffice.]

3.3 Excluded Middle

[TODO: `Classical.em`; the equivalence, in the presence of classical logic, between excluded middle and double-negation elimination; a short discussion of why constructive logic rejects excluded middle as a primitive rather than as a derived theorem.]

3.4 Exercises

Exercise 3.1. Prove Peirce's law, $((p \to q) \to p) \to p$, and identify exactly where a classical principle is required.

Chapter 4

Quantifiers

This chapter introduces universal and existential quantification, with particular attention to scope and quantifier order — the point at which a beginner's intuitions about "for all" and "there exists" most often mislead them.

4.1 Scope and Binding

[TODO: ∀ and ∃ as Lean binders; free versus bound occurrences; a short experiment showing that renaming a bound variable does not change a statement's meaning while renaming a free one does.]

4.2 Quantifier Order

Experiment: ∃-∀ implies ∀-∃

For any two-place predicate P on a type α, $(\exists x, \forall y, P\,x\,y) \to (\forall y, \exists x, P\,x\,y)$.

```
theorem ex_all_to_all_ex {alpha : Type} (P : alpha → alpha → Prop)
    (h : ∃ x, ∀ y, P x y) : ∀ y, ∃ x, P x y := by
  obtain ⟨x, ⟩hx := h
  intro y
  exact ⟨x, hx ⟩y
```

Countermodel: The converse fails

Take $\alpha = \mathrm{Fin}\ 2 = \{0, 1\}$ and $P\,x\,y := (x = y)$. Every y has some x equal to it (namely itself), but no single x is equal to *both* 0 and 1 at once.

```
example : ∀ y : Fin 2, ∃ x : Fin 2, x = y := by decide

example : ¬ ∃( x : Fin 2, ∀ y : Fin 2, x = y) := by decide
```

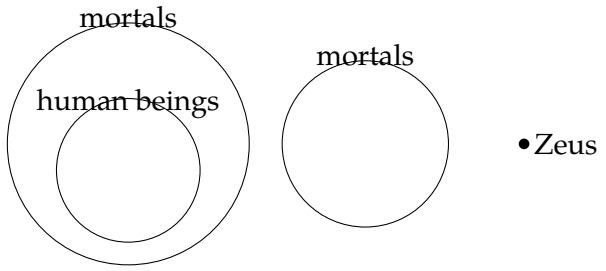
Both facts are decidable finite checks over a two-element type — no classical reasoning is needed to see the asymmetry.

4.3 Euler Diagrams: A Visual Companion to Formal Proof

Before this chapter's quantified statements get any more abstract, it is worth pausing on a much older and entirely visual method for the same job: representing a categorical statement as a disk, and a particular object as a dot, then reading validity directly off the resulting picture. The method predates modern quantifier notation by centuries, but it checks the same thing a Lean proof checks — that the conclusion is forced by the premises in *every* way the diagrams could be filled in — and it is a useful sanity check to run by eye before (or instead of) running it through a tactic script.

4.3.1 A Valid Syllogism

Consider the argument: all human beings are mortal; Zeus is not mortal; therefore Zeus is not a human being. The major premise says the "human beings" disk sits entirely inside the "mortals" disk; the minor premise places a dot for Zeus outside the "mortals" disk entirely.



There is only one way to overlay these two pictures consistently: Zeus's dot must fall outside the human-beings disk too, since that disk is entirely contained inside the mortals disk he is already outside of. The conclusion is forced — no admissible way of drawing the diagrams makes the premises true and the conclusion false.

Experiment: The syllogism formalized

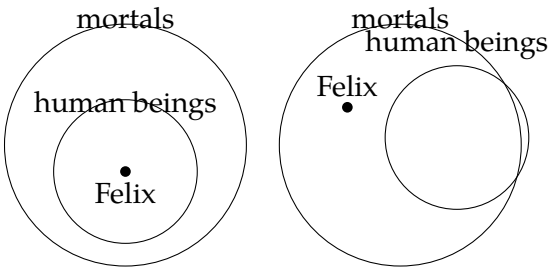
```
theorem zeus_syllogism {alpha : Type} (mortal human : alpha → Prop)
    (zeus : alpha)
    (major : ∀ x, human x → mortal x)
    (minor : ¬ mortal zeus) :
    ¬ human zeus := by
  intro hz
  exact minor (major zeus hz)
```

The Lean proof is exactly the diagram's reasoning stated symbolically: `major zeus hz` produces a proof that Zeus is mortal from the assumption that he is human, which `minor` directly contradicts.

4.3.2 An Invalid Syllogism, and the Converse Error

Now change the minor premise: all human beings are mortal; Felix is mortal; therefore Felix is a human being. The major premise diagram is unchanged, but the minor premise only

says Felix's dot lies *somewhere* inside the mortals disk — nothing pins it down relative to the human-beings disk nested inside.



Both pictures satisfy both premises, and they disagree about the conclusion: in the first, Felix is a human being; in the second, he is a mortal non-human — a cat, say. Because the premises do not force a unique picture, the argument is invalid.

Fallacy: Converse error

This argument would be valid if the major premise were replaced by its converse ("all mortals are human beings"), but a universal conditional is not logically equivalent to its converse, so no such replacement is available.

Formal	**Informal**
$\forall x,\ P(x) \to Q(x).$	If x makes $P(x)$ true, then x makes $Q(x)$ true.
$Q(a),$ for a particular $a.$	a makes $Q(x)$ true.
$\therefore P(a).$ $\quad \leftarrow$ invalid	$\therefore a$ makes $P(x)$ true. $\quad \leftarrow$ invalid

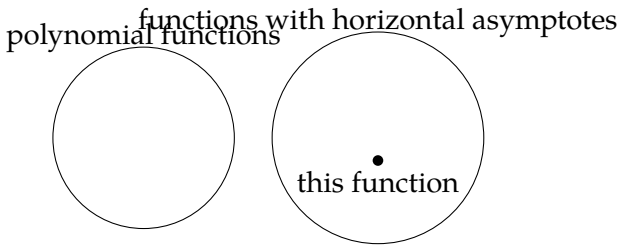
Fallacy: Inverse error

A closely related pattern: this would be valid if a conditional were logically equivalent to its *inverse*, but it is not.

Formal	**Informal**
$\forall x,\ P(x) \to Q(x).$	If x makes $P(x)$ true, then x makes $Q(x)$ true.
$\neg P(a),$ for a particular $a.$	a does not make $P(x)$ true.
$\therefore \neg Q(a).$ $\quad \leftarrow$ invalid	$\therefore a$ does not make $Q(x)$ true. $\quad \leftarrow$ invalid

4.3.3 An Argument with "No"

A universal *negative* statement — "no polynomial functions have horizontal asymptotes" — is pictured as two disks that do not overlap at all, rather than one nested inside the other.



The argument "no polynomial functions have horizontal asymptotes; this function has a horizontal asymptote; therefore this function is not a polynomial function" is valid: the dot's disk and the polynomial-functions disk share no space at all, so the dot cannot lie in both.

Experiment: Universal modus tollens

Transforming "no P are Q" into "$\forall x$, if $P(x)$ then $\neg Q(x)$" turns this into an instance of a pattern already familiar from Chapter 2.

```
theorem no_p_are_q {alpha : Type} (P Q : alpha → Prop) (a : alpha)
    (major : ∀ x, P x → ¬ Q x)
    (minor : Q a) :
    ¬ P a := by
  intro ha
  exact (major a ha) minor
```

Comparative Note: Disks and ontologies

The disk-in-disk picture used throughout this section — a category represented as a region, membership represented as containment, individuals represented as dots — is not unique to syllogistic logic. Ontology frameworks such as BFO (Basic Formal Ontology) and the OBO Foundry organize is-a hierarchies the same way: a subsuming category contains its subcategories, and reasoning about class membership reduces to the same containment question these Euler diagrams are built to answer. The notation differs, but the underlying structure — subsumption as containment — is the same object viewed twice.

4.4 Exercises

Exercise 4.1. State and prove De Morgan's laws for quantifiers: $\neg\forall x, P\,x \leftrightarrow \exists x, \neg P\,x$ (classically) and identify which direction survives constructively.

Exercise 4.2. Draw the Euler diagrams for, and construct an explicit countermodel to, the inverse-error argument form given above.

Part II

Structure

Chapter 5

Relations and Orders

This chapter covers equivalence relations, partial orders, transitivity, monotonicity, and a light introduction to well-foundedness. It is also the first chapter that recasts definitions already used informally elsewhere — extension and constraint on a history of witnessed facts — as instances of ordinary relational and order-theoretic structure, rather than leaving them as isolated metaphors.

5.1 Equivalence Relations

[TODO: reflexivity, symmetry, transitivity as a bundled structure; worked experiment proving a concrete relation is an equivalence, paired with a countermodel relation that is reflexive and symmetric but not transitive.]

5.2 Partial Orders and Monotonicity

Experiment: Extension as a partial order

A history's set of witnessed facts only grows: extension is reflexive and transitive.

```
structure History (alpha : Type) where
  witnessed : Set alpha

def Extends {alpha : Type} (earlier later : History alpha) : Prop :=
  earlier.witnessed ⊆ later.witnessed

theorem extends_refl {alpha : Type} (h : History alpha) :
    Extends h h := fun x hx => hx

theorem extends_trans {alpha : Type} {h1 h2 h3 : History alpha}
    (h12 : Extends h1 h2) (h23 : Extends h2 h3) :
    Extends h1 h3 := fun x hx => h23 (h12 hx)
```

Reflexivity and transitivity together make `Extends` a preorder on histories; [TODO: discuss why antisymmetry is the one partial-order axiom deliberately not claimed here, and what it would mean for two distinct histories to extend each other].

5.3 Well-Foundedness

[TODO: a short, non-technical introduction to well-founded relations and why they matter for induction and for guaranteeing that a constraint process terminates.]

5.4 Exercises

Exercise 5.1. Prove that intersecting an admissible set with a constraint set never enlarges it: $\text{Constrain}(A, C) \subseteq A$.

Chapter 6

Elementary Set Theory as Applied Logic

This closing chapter treats membership, subset relations, comprehension over typed domains, unions, intersections, and complements as applied instances of the logic already developed, rather than as a separate subject. Russell's paradox is used to draw a sharp distinction between an inconsistent specification and a merely unproved proposition.

6.1 Membership and Comprehension

[TODO: `Set` as predicates in Lean; comprehension notation; worked experiment proving a basic subset relation via `intro`/`exact`, directly reusing the tactics from Chapter 2.]

6.2 Unions, Intersections, Complements

[TODO: De Morgan's laws restated at the level of sets, paired explicitly with their propositional-logic counterparts from Chapter 1 to make the analogy — not merely stated, but visible in the shape of the two proofs side by side.]

6.3 Russell's Paradox: An Inconsistency, Not an Open Question

Experiment: Why unrestricted comprehension fails

[TODO: attempt to state "the set of all sets that do not contain themselves" inside Lean's type theory and show precisely where and why the attempt is rejected before it can be treated as an ordinary object — Lean's stratification blocks the paradox structurally, rather than leaving it as a proposition that happens to be false.]

6.4 Closing the Introductory Sequence

This is the last chapter of the introductory sequence. *Inference Under Constraint* continues from here into non-classical logic, modality, and a broader admissibility-theoretic program; nothing in this book depends on that continuation, and a reader who stops here has a complete, self-contained introduction to logic through Lean.

6.5 Exercises

Exercise 6.1. Prove $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ for sets over a common type.

Lean Setup and Conventions

[TODO: minimal `lakefile.toml`/`lean-toolchain` setup for readers following along; a short note on the `Prop` versus `Bool` distinction used throughout the book, since Chapter 1 introduces both registers and later chapters rely on the reader having internalized the difference.]

Bibliography

[1] J. Avigad, L. de Moura, S. Kong, and S. Ebner, *Theorem Proving in Lean 4*, online textbook, 2021. Available online: https://leanprover.github.io/theorem_proving_in_lean4/.

[2] D. van Dalen, *Logic and Structure*, 5th ed., Universitext, Springer, 2013.

[3] G. Priest, *An Introduction to Non-Classical Logic: From If to Is*, 2nd ed., Cambridge University Press, 2008.