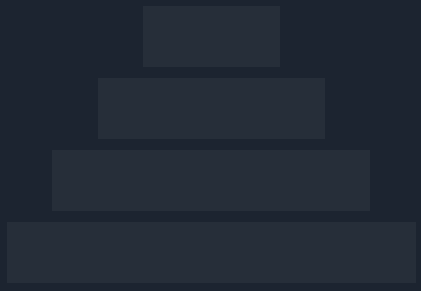
A HANDS-ON COURSE IN

# Verifying Cryptography with Lean 4

From `1+1=2` to a machine-checked proof that real elliptic-curve code is correct.



A curriculum for the curious undergraduate —
no prior formal-verification or Lean experience assumed.

---

Companion to the `*-ed25519-verified` and `pasta-pallas-verified` proof projects.

Every code snippet in this book runs. Every claim it makes about a proof, a proof assistant has checked.

# How to read this book

You are about to learn one of the most powerful ideas in computer science: how to make a computer *prove* that a program is correct — not test it on a few inputs and hope, but establish, with the certainty of mathematics, that it does the right thing on *every* input. We will aim that power at cryptography, where a single overlooked carry bit can quietly compromise every key a system ever generates.

This book assumes you can program a little and remember a little high-school algebra. It assumes **nothing** about formal methods, proof assistants, or Lean. We start from `1 + 1 = 2` and end at a real, published, machine-checked proof that the field arithmetic behind Ed25519 — the signature scheme in your SSH client, your phone, and half the internet — is correct.

- **The colored boxes each mean one thing.** A coral *big idea* box holds the load-bearing concept of a section. A grey *try it* box is an invitation to run something yourself. An amber *pitfall* box is a trap with its warning sign. A green *aha* box is an intuition meant to click. A framed *checkpoint* ends each chapter.
- **Do the exercises.** Reading a proof is like watching someone swim. You learn by getting in the water. Solutions are in the `solutions/` folder, but consult them only after a real attempt.
- **Everything runs.** The `exercises/` folder has Lean files you can open and check. When the book says "Lean accepts this," you can watch it happen.

**⋆ Aha**

The secret this book reveals: a proof is not a wall of Greek symbols meant to intimidate. A proof is a *program* — and a proof assistant is a very strict compiler for it. Once you see proofs as programs, the fear evaporates and the fun begins.

### Working the pen-and-paper material

The notebook-ruled *Pen and paper* boxes are not optional enrichment; they are half the course. Each one performs a computation with the *real* constants of the systems under study — $2^{255} - 19$, radix $2^{51}$, the fold constant 19, the actual inversion chain — because the numbers themselves carry the arguments: a headroom margin of 17 bits, a design constant that fails at 8 and works at 16, a certificate that beats trial division by a factor of $10^{34}$. Copy each one out by hand at least once — transcription is where the steps become yours. Every chapter's exercises are followed immediately by *Solutions and pathways*: the pathway (how a person finds the answer) before the answer, because the

pathway is the transferable part. The honest protocol: attempt, struggle a little, then read — in that order.

### For instructors

The book is engineered for self-study, which makes it easy to teach from: every exercise carries an immediate pathway-then-answer solution, so contact hours can go to the parts that need a human — discussing the discussion exercises (each chapter has one; they are the seminar seeds), pair-debugging the Lean files, and auditing real repositories together (Appendix C is a ready-made lab session). Grading suggestion: collect the pen-and-paper worked examples *reproduced from memory* rather than problem sets — the book's bet is that a student who can re-derive the $16p$ audit or the certificate cost ledger unprompted has the durable skill, and that bet is testable. The Lean solution files compile against the pinned toolchain in the repo; `lake build Solutions` is your answer key's answer key. Prerequisites in practice: one programming course (any language) and comfort with high-school algebra; no number theory, no logic, no Rust. The thirteen-week plan below has been paced so the two hard climbs — Chapter 9 and the Interlude — each get a full week with nothing else competing.

### A thirteen-week plan

For self-study or a seminar, the book paces naturally as a semester:

| Weeks | Material | Deliverable |
|---|---|---|
| 1 | Ch. 1 + toolkit Cards 1–2 | the two Ch. 1 audits, by hand |
| 2–3 | Ch. 2–3 + `Ch02/Ch03.lean` | term-mode proof portfolio |
| 4 | Ch. 4 + `Ch04.lean` | the `zero_add` board trace, from memory |
| 5 | Ch. 5 + `Ch05.lean` | ten goals, right tool each |
| 6 | Ch. 6 + `Ch06.lean` | the Euclid inversion, reproduced |
| 7 | Ch. 7 + `Ch07.lean` | hand-checked certificate for 97 |
| 8 | Ch. 8 + repo reading (App. C) | annotated extract of `gen/` |
| 9 | Ch. 9 + `Ch09.lean` | the miniature bridge, proved |
| 10 | Interlude | the complete by-hand verification |
| 11 | Ch. 10–11 | audit drill on a stranger's repo |
| 12 | Ch. 12 + `Ch12.lean` | graduation: spec–refusal–fix–certificate |
| 13 | project | one open lemma or one solo bridge |

# Contents

# Chapter 1

# Why Verify? The Bug That Testing Cannot Find

## 1.1 A story about one carry bit

In 2014, researchers examining widely deployed elliptic-curve code found arithmetic bugs of a very particular species: the code was correct on *almost every* input. Not most inputs — almost all of them, in a precise sense. One famous example, a carry-propagation flaw in an implementation of curve25519 arithmetic, produced a wrong answer with probability on the order of $2^{-64}$ per random input.

Pause on that number. If you tested this function a billion times per second, around the clock, you should expect to wait *centuries* before a random test happens to catch the bug. Every unit test passes. Every integration test passes. Fuzzers shrug. The code ships.

**△ Pitfall**

"It passed all the tests" means: it worked on the inputs we tried. For a 32-bit function there are four billion inputs and exhaustive testing is feasible. A field element in Ed25519 is 255 bits. The number of input *pairs* to a two-argument field operation is about $10^{153}$ — more than the square of the number of atoms in the observable universe. Testing samples a raindrop from that ocean.

**∠ Pen and paper: feel what $2^{-64}$ means, with the real numbers**

Claims about astronomical improbability deserve to be checked by hand, so check this one. A failure probability of $2^{-64}$ per random input means you expect one hit per $2^{64}$ trials. First, get $2^{64}$ into scientific notation the way you always can: $\log_{10} 2 \approx 0.30103$, so

$$\log_{10} 2^{64} = 64 \times 0.30103 \approx 19.27 \qquad \Longrightarrow \qquad 2^{64} \approx 1.8 \times 10^{19}.$$

At $10^9$ tests per second, the expected waiting time is

$$\frac{1.8 \times 10^{19}}{10^9} = 1.8 \times 10^{10} \text{ seconds.}$$

A year is $\approx 3.15 \times 10^7$ seconds (a number worth memorizing: "$\pi \times 10^7$ seconds per year" is

accidentally almost exact), so

$$\frac{1.8 \times 10^{10}}{3.15 \times 10^{7}} \approx 580 \text{ years.}$$

So: a test farm hammering this function a *billion* times per second, started when Copernicus published, would be expected to see the bug for the first time about now. And this is the *optimistic* case where failing inputs are hit by uniform sampling — for carry bugs they are typically *correlated*, clustered in corners uniform sampling underweights.
Now the input space itself. A single `FieldElement` is 255 bits; a pair is 510 bits, and

$$\log_{10} 2^{510} = 510 \times 0.30103 \approx 153.5 \qquad \Longrightarrow \qquad 2^{510} \approx 10^{153}.$$

For comparison, the number of atoms in the observable universe is around $10^{80}$. Testing all pairs is not "hard"; it is not a thing that happens in this universe.

Why does cryptographic code have bugs of exactly this shape? Because of how it must be written. To be fast and resistant to timing attacks, real implementations represent a 255-bit number in several machine-word *limbs* (we will spend happy hours with limbs in Chapter 9) and postpone expensive carry propagation as long as possible. The rare inputs where a deferred carry finally overflows are precisely the inputs no test generator stumbles on. The bug lives in the gap between "the arithmetic we meant" and "the arithmetic we wrote," and that gap is only visible on a set of inputs of measure nearly zero.

### ∠ Pen and paper: where exactly the danger zone sits — the headroom budget

You can locate the habitat of every delayed-carry bug with one line of arithmetic, using the real Ed25519 parameters. The implementation stores a field element as five limbs, each meant to carry 51 bits of payload, in 64-bit machine words. The slack between payload and word is the *headroom*:

$$64 - 51 = 13 \text{ bits of headroom per limb.}$$

Adding two elements limb-wise adds their limbs, so a freshly reduced limb (value $< 2^{51}$) can absorb additions — but each addition can roughly double the limb, i.e. spend up to one bit of headroom. How many lazy additions before a limb can reach the 64-bit cliff? We need

$$k \cdot (2^{51} - 1) \;<\; 2^{64} \qquad \Longleftrightarrow \qquad k \;\leq\; \frac{2^{64}}{2^{51}} = 2^{13} = 8192.$$

So the code may skip carry propagation for thousands of additions — a huge performance win — *provided someone, somewhere, is counting.* The 2014-species bug is precisely a miscount: a code path where the running total of spent headroom exceeds the budget on inputs shaped just so. Notice what kind of fact the budget is: a *quantified arithmetic invariant* ("for all reachable values, limb $< 2^{51+j}$ after $j$ additions") — exactly the kind of statement a test cannot establish and a proof assistant eats for breakfast. When Chapter 10 makes "bounds clauses" feel bureaucratic, remember this box: the bounds clause *is* the headroom count, and the headroom count is where the bodies were buried.

And in cryptography, "rare wrong answer" does not mean "rare small glitch." Wrong field arithmetic can leak private keys: several published attacks turn a single faulty group operation into full key recovery. The stakes are not a corrupted pixel; they are every signature your machine has ever made.

## 1.2 There is another way

What if, instead of sampling inputs, we could make a statement about *all* of them — and have a machine check that statement with the same rigor a compiler checks syntax?

**⁎ The big idea**

A **formal proof of correctness** is a mathematical argument, written in a language precise enough for a computer to verify, that a program satisfies its specification on *every* input. Not sampled. Not probabilistic. Every input, forever, or the proof does not check.

The tool that checks such arguments is called a *proof assistant*. This book uses **Lean 4**, a modern proof assistant that is also a full-fledged programming language. Others you may have heard of: Rocq (formerly Coq), Isabelle/HOL, Agda. The ideas transfer; the syntax differs.

A proof assistant is built around a small, paranoid core called the *kernel*. Everything you will learn in this book — clever tactics, powerful automation, beautiful notation — is scaffolding whose only job is to produce a proof object the kernel accepts. The kernel is a few thousand lines of code that does one thing: check that each step of a proof follows from the previous ones by a fixed set of rules. If the kernel accepts, the theorem holds. If it does not, no amount of confidence, seniority, or good intentions makes the program correct.

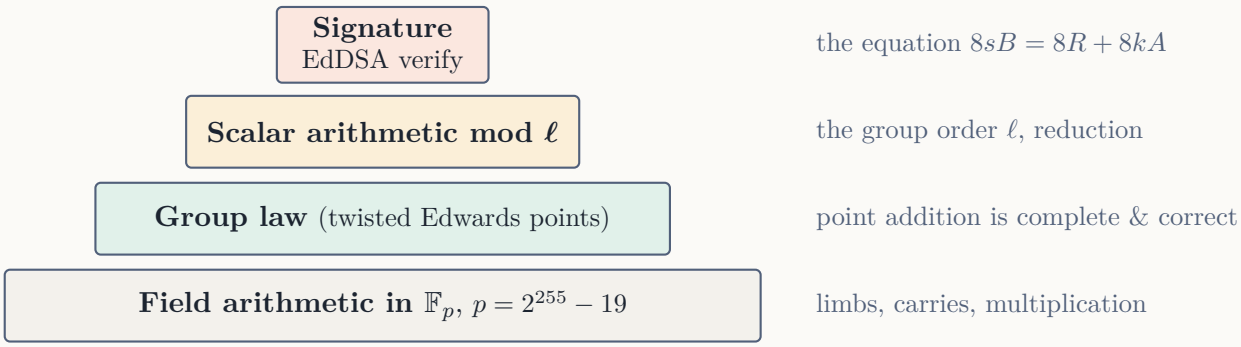
**⋆ Aha**

Here is the emotional core of formal verification, and it is worth internalizing early: **the proof assistant is not your examiner, it is your collaborator**. It never gets tired, never skips a case, never says "obviously." Every hour you spend arguing with it is an hour a bug did not survive. People who love proof assistants love them the way climbers love a good belayer.

## 1.3 What we will actually verify

This book is not a tour of toy examples. It is the curriculum companion to a set of real verification projects in which the arithmetic core of **Ed25519** — the elliptic-curve signature scheme used by SSH, Signal, TLS, and most cryptocurrency systems — was machine-checked in Lean 4, starting from the actual Rust source code of the `curve25519-dalek` library and several of its production forks.

The proofs are organized as a pyramid. Each layer states the correctness of one abstraction level and rests on the layer beneath it:



By the end of this book you will be able to read — and extend — the real proofs at every layer of this pyramid. The journey looks like this:

- **Chapters 2–5** teach Lean itself, from `#eval 1+1` to proofs by induction and the automation that dispatches arithmetic goals.
- **Chapters 6–7** build the mathematics: modular arithmetic, finite fields, and how to convince a paranoid kernel that a 77-digit number is prime.
- **Chapters 8–9** cross the bridge from Rust to Lean: how real code is translated into a form we can reason about, and the single most important idea in the whole enterprise — the *denotation function*.
- **Chapters 10–12** assemble the pyramid: field correctness, the ethics of axioms and honest boundaries, and the layers above.

## 1.4 Proofs versus tests: the honest comparison

Formal verification is not magic, and this book will never pretend otherwise. It is worth being precise, right now, about what a machine-checked proof does and does not give you.

| Testing | Proving |
|---|---|
| Checks sampled inputs | Checks *all* inputs |
| Cheap to start, cheap to run | Expensive to write, cheap to re-check |
| Finds bugs | Establishes their absence (w.r.t. the spec) |
| Trusts nothing | Trusts the spec, the model, the kernel |
| Silent about *why* code is right | The proof *is* the why |

That word *spec* in the right column is the fine print, and it matters enormously. A proof shows that code satisfies a specification. If the specification says the wrong thing — or says nothing, or is accidentally trivial — the proof is worthless no matter how green the checkmark. A recurring theme of this book (it gets its own chapter, Chapter 11) is how to read a verification claim skeptically: What exactly was proven? Against which model of the code? Resting on which axioms?

**⋆ Aha**

The most dangerous artifact in formal methods is not a wrong proof — the kernel prevents those. It is a *correct proof of the wrong statement*. Learning to smell those is as important as learning to write proofs at all.

## 1.5 Why Lean, and why now

Twenty years ago, verifying real cryptographic C or Rust code was a heroic, multi-year effort. Three things changed:

1. **Proof assistants matured.** Lean 4 is fast, pleasant, and comes with *Mathlib*, a library of over a million lines of formalized mathematics — finite fields and elliptic-curve ingredients included, so we do not start from bare axioms.
2. **Translation pipelines appeared.** Tools like *Charon* and *Aeneas* mechanically translate real Rust code into Lean definitions, so the thing we verify is derived from the code that ships, not a hand-transcribed approximation (Chapter 8).
3. **Automation got serious.** Decision procedures like `omega` (linear integer arithmetic) and `decide` discharge the boring 90% of goals, leaving humans the interesting 10%.

None of this made verification *easy*. It made verification *possible for a well-prepared person in finite time* — and preparing you is exactly what this book is for.

**>_ Try it yourself**

You do not need anything installed yet, but if you want to run code from Chapter 2 onward, install Lean now. One command:

```
curl https://elan.lean-lang.org/elan-init.sh -sSf | sh
```

Then open the `exercises/` folder of this repository in VS Code with the *Lean 4* extension. The orange progress bar you will see is the proof checker working through the file — your new collaborator saying hello.

## Exercises

**Exercise 1.1.** A function takes two 255-bit inputs and is buggy on exactly one input pair. Assume you can test $10^9$ random pairs per second. Estimate the expected time to find the bug by random testing, in multiples of the age of the universe ($\approx 4 \times 10^{17}$ seconds). You may approximate freely; the point is the order of magnitude.

**Exercise 1.2.** Give an example, from your own programming experience, of a bug that survived a test suite. What property would a specification have needed to state in order to exclude it?

**Exercise 1.3.** (Discussion) A colleague says: "Our crypto library is audited by three firms every year; formal verification is redundant." Name one class of defect audits are better at than proofs, and one class where proofs are strictly stronger.

**Exercise 1.4.** Redo the headroom budget for a hypothetical radix-26 representation on 32-bit words (ten limbs of 26 bits for a 255-bit value, a real design used on small CPUs). How many bits of headroom per limb? How many lazy additions fit in the budget? Compare with the radix-51/64-bit numbers and state which design must reduce more often.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 1.1.

*Pathway.* The only inputs that reveal the bug form a set of size 1 inside a set of size $2^{510}$, so a uniformly random test hits it with probability $2^{-510}$; expected number of trials is $2^{510}$ (waiting time of a geometric distribution). Then it is the Chapter-1 conversion drill: powers of two → powers of ten → seconds → universes.

*Answer.* Expected trials $2^{510} \approx 10^{153.5}$. At $10^9$ per second: $10^{153.5-9} = 10^{144.5}$ seconds. Divide by the age of the universe, $4 \times 10^{17}$ s:

$$\frac{10^{144.5}}{4 \times 10^{17}} \approx 10^{126.9} \quad \text{— about } 10^{127} \text{ universe-ages.}$$

Any answer within a few orders of magnitude is "correct": the lesson is that no engineering factor (faster farms, smarter fuzzing schedules) dents a number with 127 digits of margin.

### Solution 1.2.

*Pathway.* Pick a bug whose trigger was a *property of the input*, not a coding typo — those are the ones a spec excludes. Then ask: what is the universally quantified sentence that is false in the buggy program?

*Answer.* (Model answer.) A JSON parser that crashed on deeply nested arrays survived a big test suite: no test nested past depth 50. The specification that excludes it must *quantify over all inputs*: "for every input string, the parser terminates and returns either a value or a well-formed error" — termination-for-all is exactly what the tests never said. The general shape to remember: test suites assert $P(x_1), \ldots, P(x_n)$; specifications assert $\forall x,\ P(x)$; bugs live in the gap.

### Solution 1.3.

*Pathway.* Sort defect classes by *whether their badness is expressible as a violated formal property of the code.* Audits see things that are not properties of the code; proofs cover input space no human can.

*Answer.* Audits win at: flaws in the *specification itself* and its surroundings — wrong protocol choice, misuse-prone APIs, side channels outside the model, deployment and key-handling practice. A proof of the wrong spec passes; a good auditor smells that the spec is wrong. Proofs are strictly stronger at: input-space coverage for the stated property — carry bugs, overflow corners, algebraic edge cases on a measure-zero slice. The honest synthesis: audits examine the *claim*, proofs guarantee the *claim's body.* A serious system wants both, aimed at their targets.

### Solution 1.4.

*Pathway.* Same two lines as the worked example, new constants: headroom = word − radix; budget $= 2^{\text{headroom}}$.

*Answer.* Headroom $32 - 26 = 6$ bits, so at most $2^{32}/2^{26} = 2^6 = 64$ lazy additions — against 8192 for radix-51/64-bit, a budget 128 times tighter. The radix-26 design must interleave reductions far more

often, and its correctness argument has 128 times less slack for miscounting — one concrete reason ports of crypto code to small targets are disproportionately bug-prone, and why per-fork verification (Chapter 10) is not paranoia.

**⋈ Checkpoint**

Before moving on, you should be able to explain to a friend: (1) why testing fundamentally cannot establish correctness of a 255-bit arithmetic function; (2) what a proof assistant's kernel is and why its small size matters; (3) what a proof of correctness actually promises — and the role the specification plays in that promise.

# Chapter 2

# Meet Lean: A Language Where Programs and Proofs Live Together

## 2.1 First contact

Lean 4 is two things wearing one syntax: a programming language (fast, functional, compiled) and a proof assistant. You will learn both faces, but we start with the friendlier one. Open a new file `Scratch.lean` and type:

```
#eval 1 + 1           -- 2
#eval 2 ^ 255 - 19    -- a 77-digit number, instantly
#eval "hello".length -- 5
```

`#eval` runs an expression and prints the result right in your editor. Notice the second line: Lean's natural numbers are *arbitrary precision* by default. The number $2^{255} - 19$ — which will follow us through the whole book — is a perfectly ordinary value here, not an overflow.

`#check` asks a different question: not "what is the value?" but "what is the *type*?"

```
#check 1 + 1         -- 1 + 1 : Nat
#check "hello"       -- "hello" : String
#check (1 : Int) - 5 -- Int
```

**∗ The big idea**

In Lean, **every expression has a type**, and the type checker verifies every file before anything runs. This obsession with types is not bureaucracy — in Chapter 3 it will turn out to be the entire mechanism by which proofs work. Learn to read `e : T` as "$e$ is of type $T$" — or, with a squint we will justify later, "$e$ is *evidence* for $T$."

## 2.2 Definitions and functions

New names are introduced with `def`:

```
def p : Nat := 2 ^ 255 - 19

def double (n : Nat) : Nat := 2 * n

def isEven (n : Nat) : Bool := n % 2 == 0

#eval double 21      -- 42
#eval isEven p       -- false (p is odd, good: p is supposed to be prime!)
```

Three things to absorb from this snippet:

- Function application is written with a space: `double 21`, not `double(21)`. It looks strange for a week and then all other syntax looks noisy forever after.
- Every definition states its types: `double` takes a `Nat` and returns a `Nat`. Lean can often infer types, but in this book we write them — specifications are the whole game, and a type is a tiny specification.
- Definitions are *immutable equations*, not instructions. There is no "assignment." `p` *is* $2^{255} - 19$, forever.

Functions of several arguments simply take them in sequence, and functions are values you can pass around:

```
def addMul (a b c : Nat) : Nat := a + b * c

def twice (f : Nat -> Nat) (x : Nat) : Nat := f (f x)

#eval twice double 10   -- 40
```

The type of `twice` is worth staring at: `(Nat -> Nat) -> Nat -> Nat`. Arrows associate to the right, and a multi-argument function is really a chain of single-argument ones — this is called *currying*. Nothing about it needs memorizing now; it will become muscle memory.

## 2.3 Inductive types: building data from nothing

Where do types like `Nat` and `Bool` come from? They are not built-in magic. They are *inductive types* — data types defined by listing every way to construct a value. Here is `Bool`, exactly as the core library defines it:

```
inductive Bool where
  | false : Bool
  | true  : Bool
```

Read: "a `Bool` is either `false` or `true`, and there is no other way to make one." That closed-world clause is what makes case analysis — and later, proofs by cases — airtight.

Now the star of the show. The natural numbers, following Peano's 1889 idea:

```
inductive Nat where
  | zero : Nat               -- 0 exists
  | succ (n : Nat) : Nat     -- every number has a successor
```

Every natural number is `zero` wrapped in finitely many `succ`s: the number 3 *is* `succ (succ (succ zero))`. (Lean stores big numbers efficiently under the hood, but *reasons* about them through this two-case skeleton.) Functions on inductive types are defined by *pattern matching* — one equation per constructor:

```
def add : Nat -> Nat -> Nat
  | m, Nat.zero   => m
  | m, Nat.succ n => Nat.succ (add m n)
```

### ∠ Pen and paper: running the definition of addition by hand

The two equations of `add` are a complete calculator; let us be the computer once, because every proof in Chapter 4 secretly replays this trace. Abbreviate `Nat.succ` as $s$ and `Nat.zero` as $z$, so $3 = s(s(s(z)))$ and $2 = s(s(z))$. Compute `add` 3 2; at each step exactly one equation applies, decided by the *second* argument's outermost constructor:

$$\begin{aligned} \texttt{add}\ 3\ s(s(z)) &= s(\texttt{add}\ 3\ s(z)) && \text{(second equation, } n = s(z)) \\ &= s(s(\texttt{add}\ 3\ z)) && \text{(second equation, } n = z) \\ &= s(s(3)) && \text{(first equation)} \\ &= s(s(s(s(s(z))))) \ = \ 5. \end{aligned}$$

Three observations to carry forward. (1) The recursion counts down the *second* argument only — the first is inert. That asymmetry is why `n + 0 = n` will hold "by computation" while `0 + n = n` will not (Chapter 3 makes this precise). (2) Each step is justified by *matching a constructor* — there is no arithmetic insight anywhere, just pattern matching, which is why a small, dumb kernel can check it. (3) The number of steps equals the second argument — fine for 2, comedy for $2^{255}$; Lean therefore *stores* numerals in efficient binary and uses this unary skeleton only for *reasoning*. Keep the two roles separate in your head: fast representation for computing, tiny inductive skeleton for proving.

### ⋆ Aha

Look at what just happened. Addition — the operation at the bottom of every cryptosystem in this book — is not an axiom or a CPU instruction here. It is a *two-line recursive program*, and every arithmetic fact we will ever prove unwinds, ultimately, to these two equations. When Lean later claims $a + b = b + a$ for *all* numbers, it will be because the structure of this definition forces it, not because anyone tested it.

**△ Pitfall**

`Nat` subtraction *truncates*: `#eval (3 - 5 : Nat)` prints `0`, not $-2$, because natural numbers have nowhere to go below zero. This single fact causes a large fraction of all beginner proof failures — an identity like $a - b + b = a$ is simply *false* for `Nat`. When subtraction must mean subtraction, use `Int`, or carry a hypothesis $b \le a$. Real verification projects hit this constantly: machine arithmetic wraps, truncates, and overflows, and the proofs must say so honestly.

**∠ Pen and paper: truncation meeting real cryptographic constants**

Do not file the truncation pitfall under "beginner trivia"; run it against the book's own constants. In the Ed25519 field code, subtraction of field elements must compute $a - b$ *as integers would*, but the limbs live in `Nat`-like unsigned words. Take the real situation: a fresh limb $a_0 < 2^{51}$ and a lazily-grown limb $b_0 < 2^{54}$ (Chapter 8 explains how limbs grow). Then in `Nat`:

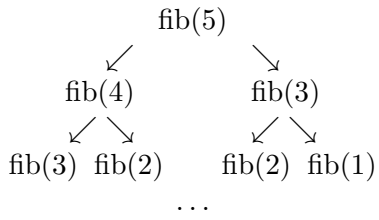
$$a_0 - b_0 \;=\; 0 \quad \text{whenever } b_0 > a_0 \qquad \text{— the true difference is simply gone.}$$

Concretely: $a_0 = 2^{51} - 19$ (the low limb of $p$ itself!) and $b_0 = 2^{53}$: true difference $2^{51} - 19 - 2^{53} = -(3 \cdot 2^{51} + 19)$, truncated result 0. Any downstream carry chain silently absorbs the 0 and produces a well-formed, plausible, *wrong* field element.

The production fix (verified in Chapter 10) is to compute $a - b$ as $a + (16p - b)$: adding the multiple-of-$p$ constant $16p$ lifts every limb difference above zero *before* subtracting, and the extra $16p$ vanishes modulo $p$. Check the crucial inequality yourself with the real numbers: the largest subtrahend limb allowed by the bounds discipline is $< 2^{54}$, and the smallest limb of the $16p$ constant is $16 \cdot (2^{51} - 19) = 2^{55} - 304$, so every limb of $16p - b$ is at least $2^{55} - 304 - (2^{54} - 1) = 2^{54} - 303 > 0$. No truncation, on any input — and note that the argument needed $16p$: the same computation with $8p$ bottoms out at $2^{54} - 152 - (2^{54} - 1) = -151 < 0$, which *can* truncate. One design constant, justified by three lines of pen-and-paper arithmetic you just verified.

**∠ Pen and paper: drawing the cost of a definition — the `fib` call tree**

A definition's *meaning* and its *cost* are different objects, and one drawing separates them forever. Take the naive recurrence $\text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n)$ (exercise 2.2) and draw the calls made by fib(5) as a tree, each node spawning its two recursive children:




Count the visible duplication: fib(3) is computed twice, fib(2) three times, fib(1) five times — the *multiplicities are themselves Fibonacci numbers*, which you can prove by noticing that fib($k$)'s call count obeys the same recurrence as fib itself. Total calls for fib($n$): $C(n) = 2\,F_{n+1} - 1$, exponential in $n$ — the tree is a picture of the exponent. Two lessons travel with the picture. First, the *value* fib(5) = 5 is independent of the route: an efficient two-accumulator loop computes the same function, and "the same function" is exactly the kind of statement Lean

> can *prove* (equal outputs for all inputs) — specs describe meaning, never cost. Second, this meaning/cost split is why Chapter 7 can let a slow-but-simple definition *define* truth while fast code *computes* it, with a theorem pinning them together — the whole verified-crypto architecture in embryo, visible in a doodle of fib(5).

## 2.4 Structures: records with guarantees

The last data-building tool we need bundles several fields together:

```
structure Point where
  x : Int
  y : Int

def origin : Point := { x := 0, y := 0 }

#eval origin.x   -- 0
```

A preview of why this matters to us: the Rust type `struct FieldElement51(pub [u64; 5])` — five 64-bit limbs representing one element of $\mathbb{F}_p$ — will arrive in Lean (Chapter 8) as essentially a structure holding an array of five machine words. The verification question of this entire book is: *do operations on those five words faithfully implement arithmetic in $\mathbb{F}_p$?* Structures are how the data crosses the bridge.

## 2.5 Namespaces, Mathlib, and reading error messages

Real developments organize names in `namespace` blocks (`Nat.add`, `Point.x`) and import the mathematical library:

```
import Mathlib
open Nat

#check Nat.Prime          -- the primality predicate, ready-made
#check ZMod               -- integers mod n -- our Chapter 6 home
```

And a word of comfort about *error messages*. You will see many. Lean's are precise and honest, and the single most useful habit you can develop this week is: **read the expected/actual types in the message, slowly**. A message like

```
type mismatch: argument has type Int but is expected to have type Nat
```

is not the compiler being difficult; it is a specification violation caught at the cheapest possible moment. Verification is this same experience scaled up: the machine holds the line, and the line is exactly where you drew it.

**>_ Try it yourself**

Open `exercises/Ch02.lean`. It contains the definitions from this chapter with a few holes marked `sorry` (a placeholder Lean accepts with a loud warning). Replace each with a working definition and watch the warnings disappear. In particular: define `mul : Nat -> Nat -> Nat` by recursion, using `add` — the same bootstrapping order (add, then mul) the real field proofs follow.

## Exercises

**Exercise 2.1.** Define `pow : Nat -> Nat -> Nat` (by recursion on the exponent) and check with `#eval` that `pow 2 10 = 1024`.

**Exercise 2.2.** Define `fib : Nat -> Nat`. Then evaluate `fib 32`. Notice the pause — naive recursion is exponential. (Lean's `#eval` is fast; your algorithm is slow. The distinction matters when we later care about *what* is being computed versus *how*.)

**Exercise 2.3.** Write a structure `Rational` with fields `num : Int` and `den : Nat`, and a function `Rational.add`. What property of `den` can your type *not* enforce yet? (Keep your answer; Chapter 3 gives you the tool to fix it.)

**Exercise 2.4.** (Paper) Using only the two defining equations of `add`, write out the full reduction trace of `add 2 3` the way the worked example traced `add 3 2`. How many steps does each take? State the general rule for the step count of `add m n`.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 2.1.

*Pathway.* Ask: which argument does the recursion consume? An exponent counts *how many times* to multiply — so recurse on it, exactly as `add` recursed on "how many times to take a successor." The base case is the empty product. If you wrote `pow b 0 = 0`, evaluate `pow 2 0` against your algebra ($b^0 = 1$) and let the mismatch teach the convention.

*Answer.*

```
def pow : Nat → Nat → Nat
  | _, Nat.zero   => 1
  | b, Nat.succ e => pow b e * b
```

`#eval pow 2 10` prints `1024`. Every choice mirrors `add`: recursion on the count, base case the operation's identity element (0 for $+$, 1 for $\times$) — a pattern you will reuse for iterated point addition in Chapter 12.

### Solution 2.2.

*Pathway.* The definition writes itself from the recurrence $F_{n+2} = F_{n+1} + F_n$ with two base cases. The interesting part is the pause, and it is worth quantifying rather than shrugging at: let $C(n)$ count the calls needed for `fib n`. Each call recurses twice, so $C(n) = C(n-1) + C(n-2) + 1$ — the cost obeys (almost) the Fibonacci recurrence itself, hence grows like the golden ratio to the $n$.

*Answer.*

```
def fib : Nat → Nat
  | 0 => 0
  | 1 => 1
  | n + 2 => fib (n + 1) + fib n
```

One can show $C(n) = 2\,F_{n+1} - 1$; for $n = 32$ that is $2 \cdot 3524578 - 1 \approx 7 \times 10^6$ calls to produce the seven-digit answer $F_{32} = 2178309$ — the value is cheap, the *route* is exponential. Hold on to the distinction: in verification we constantly separate *what* a function computes (its spec) from *how* (its cost and structure); this is your first taste of the two coming apart.

### Solution 2.3.

*Pathway.* The formula $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ transcribes directly; the sting is in the question. Try to build a value that should not exist.

*Answer.*

```
def Rational.add (x y : Rational) : Rational :=
  ⟨x.num * y.den + y.num * x.den, x.den * y.den⟩
```

The type cannot enforce `den` $\neq 0$: the value `⟨1, 0⟩` constructs without complaint, and every function must now either handle it or silently misbehave on it. Chapter 3 adds a third field — *a proof* `den` $\neq$ `0` stored inside the value — after which the bad value is not handled but *unrepresentable*. The same move, scaled up, is the bounds-carrying field element of Chapter 8.

### Solution 2.4.

*Pathway.* Constructor of the second argument decides the equation; count how many times you use the second equation before the first fires.

*Answer.* `add` $2\ s(s(s(z))) = s($`add` $2\ s(s(z))) = s(s($`add` $2\ s(z))) = s(s(s($`add` $2\ z))) = s(s(s(2))) = 5$: three uses of the second equation (one per `succ` in the second argument), one of the first — four steps. `add 3 2` took three. In general `add m n` takes $n + 1$ steps: $n$ unrollings plus the base case. The first argument never influences the step count — it is along for the ride, which is precisely the asymmetry that will make `0 + n = n` a theorem needing induction rather than a computation.

**⋈ Checkpoint**

You should now be able to: evaluate and type-check expressions with `#eval`/`#check`; define functions, including recursive ones over `Nat`; explain what an inductive type is and recite the two constructors of `Nat`; and state from memory what `Nat` subtraction does on $3 - 5$, and why that will matter.

# Chapter 3

# Propositions as Types: The Idea That Makes It All Work

## 3.1 A suspicious similarity

Here are two things that look unrelated. First, a function that converts a pair into something else:

```
def swap (pair : A x B) : B x A := (pair.2, pair.1)
```

Second, a fact of logic: *if $A$ and $B$ both hold, then $B$ and $A$ both hold.* To prove it, you would say: "suppose I have evidence for $A$ and evidence for $B$; then I can produce evidence for $B$ and evidence for $A$ — just present the same two pieces in the other order."

That prose proof and that program are the *same object*. The function takes a pair of values and reorders it; the proof takes a pair of pieces of evidence and reorders it. This is not an analogy or a teaching trick. It is a theorem about logic and computation, discovered independently by logicians (Curry, Howard) and now the load-bearing wall of Lean:

**∗ The big idea**

**The Curry–Howard correspondence.** A proposition can be read as a type — the type of its proofs. A proof is then simply a *program* of that type. Checking a proof is type-checking a program. There is no separate "proof checker" bolted onto Lean: the type checker you met in Chapter 2 *is* the proof checker.

| Logic | Programming | In Lean |
|---|---|---|
| proposition $P$ | type | `P : Prop` |
| proof of $P$ | value/program of that type | `h : P` |
| $P \to Q$ (implication) | function type | `P -> Q` |
| $P \wedge Q$ (and) | pair type | `P / Q` |
| $P \vee Q$ (or) | tagged union | `P Q` |
| $\neg P$ (not) | `P -> False` | `Not P` |
| "true" | type with one trivial value | `True` |
| "false" | *empty* type | `False` |

Take a minute with the last row: `False` is a type with *no constructors* — no way to build a value. To prove a false statement you would have to produce an inhabitant of an empty type. That is why the system is sound: lies have no evidence, so lies do not type-check.

## 3.2 Proofs are programs: first proofs

Let us write actual proofs as actual programs. Implication is a function type, so proving "$P$ implies $P$" means writing the identity function:

```
theorem p_implies_p (P : Prop) : P -> P :=
  fun h => h
```

Read `fun h => h` aloud as a proof: "assume $P$ holds — call the evidence $h$; then $P$ holds, by $h$." Every classical proof-writing phrase has a program shape:

| You say in prose | You write in Lean |
|---|---|
| "Assume $P$; call it $h$" | `fun h => ...` |
| "By hypothesis $h$" | `h` |
| "Apply lemma $f$ to fact $h$" | `f h` |
| "Both parts hold: ... and ..." | `And.intro pf1 pf2` |
| "From $h : P \wedge Q$, the first part" | `h.1` |

The pair-swapping example, now as an official theorem:

```
theorem and_swap (P Q : Prop) : P /\ Q -> Q /\ P :=
  fun h => And.intro h.2 h.1
```

And transitivity of implication is exactly function composition:

```
theorem imp_trans (P Q R : Prop) : (P -> Q) -> (Q -> R) -> (P -> R) :=
  fun pq qr => fun p => qr (pq p)
```

**∠ Pen and paper: type-checking a proof by hand — being the kernel once**

When Lean accepts `and_swap`, what does it actually *do*? It type-checks the term — and you can replay that check on paper, which is the single best way to make Curry–Howard stop feeling like a slogan. The term is `fun h => And.intro h.2 h.1`; the claimed type is `P ∧ Q → Q ∧ P`. A type-checker works by walking the term and maintaining a *context* of what each variable is assumed to be — write the context on the left of a $\vdash$, the judgment on the right:

1. Goal: $\vdash$ `fun h` $\Rightarrow \ldots : P \wedge Q \to Q \wedge P$
2. A `fun` against an arrow type: put the argument in the context.
   $h : P \wedge Q \ \vdash$ `And.intro` $h.2\ h.1 \ : \ Q \wedge P$
3. `And.intro` needs a proof of $Q$ and a proof of $P$ (in that order, to build $Q \wedge P$).
4. $h : P \wedge Q \ \vdash \ h.2 : Q$ ✓ (second component of a conjunction)
5. $h : P \wedge Q \ \vdash \ h.1 : P$ ✓ (first component)
6. All leaves check; the term has the claimed type. QED.

Every step is a bookkeeping rule — "`fun` eats an arrow," "application must match the function's argument type," "`.1` projects a pair." No inspiration occurred anywhere, which is the entire point: *checking* a proof is mechanical (a few thousand lines of kernel), while *finding* it is the creative act. Now deliberately break it: try to check `fun h => And.intro h.1 h.2` against the same type. Step 4 demands $h.1 : Q$, but the context gives $h.1 : P$ — mismatch, rejected. A wrong proof does not check; you have just watched soundness happen at the level where it lives.

**⋆ Aha**

If you have ever composed two functions, you have already done everything this proof does. The intimidating part of formal logic — "natural deduction," "inference rules" — turns out to be the part you knew from programming all along. What logicians call *modus ponens*, you call *calling a function*.

## 3.3 Equality and the proof that $1 + 1 = 2$

The proposition $a = b$ is also a type. Its only constructor is reflexivity — `rfl` — which proves `a = a`. How can that ever prove anything interesting? Because Lean *computes* before comparing:

```
theorem one_plus_one : 1 + 1 = 2 := rfl
```

Lean unfolds `1 + 1` using the two-line definition of addition from Chapter 2, arrives at `2`, and sees that both sides are *literally the same value*. The equation holds by computation. This mechanism — *definitional equality* — is the engine that lets proofs lean on programs, and later lets us prove facts about extracted Rust code by, in part, just running it symbolically.

**∠ Pen and paper: watching `rfl` compute — and watching it get stuck**

Both halves of the `rfl` story fit on one sheet of paper, using the `add` equations from Chapter 2 (recursion on the second argument; $s$ for successor). First, the success. To check `1 + 1 = 2`, the kernel normalizes both sides:

$$1+1 \;=\; \texttt{add}\ s(z)\ s(z) \;=\; s(\texttt{add}\ s(z)\ z) \;=\; s(s(z)) \;=\; 2.$$

Two rewrite steps, both forced by the definition, and the two sides become *the same term* — `rfl` applies. Nothing about this argument used the smallness of 1: for `2^255 - 19` the same normalization runs on the efficient binary representation and succeeds just as surely (only your patience is finite, not the principle).
Now the failure, on the same sheet. To check `0 + n = n` for a *variable* $n$, the kernel again tries to normalize the left side:

$$\texttt{add}\ z\ n\ = ?$$

Which defining equation applies? The first needs the second argument to be $z$; the second needs it to be $s(\cdot)$. But $n$ is *opaque* — an arbitrary variable exhibits neither constructor. No equation fires, the term is stuck, the two sides stay different terms, `rfl` fails. Note precisely what failed: not the *truth* of the statement (it is true for every concrete $n$, and each instance — `0 + 5 = 5`, say — normalizes fine), but computation's ability to see it *uniformly*. Whenever a variable blocks the recursion pattern, computation ends and *induction* must take over — now you can predict, before trying, which equalities are `rfl`s. Test yourself: `n + 0 = n`? (Second argument is the concrete $z$ — first equation fires — `rfl`.) `n * 1 = n`? Depends on which argument `mul` recurses on; go read your Chapter 2 solution and decide.

**△ Pitfall**

`rfl` proves $2^{255} - 19$-sized computations happily, but it can only prove what computation alone can see. `n + 0 = n` is `rfl` (the definition's first equation matches), yet `0 + n = n` is *not* — recursion is on the *second* argument, and `n` is an opaque variable, so nothing unfolds. The statement is still true; it just needs a real proof (induction — next chapter). The asymmetry feels unfair for about a day. Then it becomes your sharpest mental model of what a computer can and cannot know for free.

## 3.4 Universals, existentials, and dependent types

Cryptographic specifications are universal statements: "*for all* inputs, the output is correct." In Lean, ∀ is a function type whose *result type mentions the argument*:

```
theorem add_self_even : forall n : Nat, isEven (n + n) = true := ...
```

A proof of `forall n, P n` is a function that eats any `n` and returns a proof of `P n` — one uniform recipe covering all the infinitely many cases at once. This is precisely the thing testing could not give us in Chapter 1: testing produces finitely many `P 3`, `P 17`, `P 42`; a proof produces the function.

**∠ Pen and paper: a universal statement as one uniform recipe**

Feel the difference between testing and proving on the smallest interesting universal: *every number of the form $n + n$ is even*, where $\text{even}(m) := \exists k,\ m = 2k$. The testing mind samples: $3 + 3 = 6 = 2 \cdot 3$ ✓, $7 + 7 = 14 = 2 \cdot 7$ ✓, and after a while believes. The proving mind writes *one function*:

$$\text{given any } n, \text{ return the witness } k := n \text{ together with the fact } n + n = 2n.$$

That is the entire proof — a recipe that, *handed any* $n$, produces the evidence for *that* $n$: for $n = 3$ it yields witness 3 and the equation $6 = 6$; for $n = 2^{254}$ it yields witness $2^{254}$ just as cheaply, because the recipe never looks at the digits. In Lean:

```
theorem add_self_even : ∀ n : Nat, ∃ k, n + n = 2 * k :=
  fun n => ⟨n, (Nat.two_mul n).symm⟩
```

Read the term against the table: `fun n =>` consumes the ∀ (a function, as promised); `⟨n, ...⟩` produces the ∃ (a witness paired with evidence). Now the punchline for this book: every correctness certificate in the companion repositories has exactly this shape — `∀ a b, Bnd a → Bnd b → ∃ c, ...` — a function that eats *arbitrary* 255-bit inputs and manufactures the evidence for them. When Chapter 1 said proofs cover $10^{153}$ input pairs at once, this recipe-not-samples mechanism is *how*. No induction was even needed here; when the recipe does need to consult the structure of $n$, induction (Chapter 4) is how a recipe consults structure.

Dually, `exists n, P n` is proved by handing over a concrete witness together with evidence: `Exists.intro 4 pf`. And remember the `Rational` exercise from last chapter — the denominator you could not keep nonzero? Dependent types fix it by letting data carry proofs:

```
structure Rational where
  num    : Int
  den    : Nat
  den_ne_zero : den ≠ 0     -- a PROOF, stored inside the value
```

No value of this type with a zero denominator can ever be constructed, anywhere, by anyone. In the real Ed25519 development this exact pattern appears as a *bounds invariant*: a field element travels together with the proof that its five limbs are small enough not to overflow the next multiplication. The data structure makes the unsafe states unrepresentable.

## 3.5 What about proof by contradiction?

One more resident of the logical zoo. Lean's core logic is *constructive*: a proof of existence builds a witness. Classical reasoning — "it's either true or false, and not false, hence true" — is available the moment you want it (Mathlib imports it as `Classical.choice`, and we will meet it again on the trust ledger in Chapter 11), but it is an *ingredient you can see*, not smuggled seasoning. When a verification result says "this proof uses only `propext`, `Classical.choice`, `Quot.sound`," that is a complete list of the logical beliefs you are being asked to hold. Three. You can audit them over coffee.

**>_ Try it yourself**

Open `exercises/Ch03.lean` and prove, as programs (no tactics yet!): `P -> Q -> P`; `(P / Q) -> (P  Q)`; and modus ponens `P -> (P -> Q) -> Q`. Each is a one-liner. Feel free to be delighted when the pieces click together like typed Lego.

## Exercises

**Exercise 3.1.** Prove `and_assoc :  (P / Q) / R -> P / (Q / R)` as a term-mode program using `h.1`, `h.2`, and `And.intro`.

**Exercise 3.2.** Prove `or_swap :  P  Q -> Q  P`. You will need case analysis on which side holds: `match h with | Or.inl p => ...  | Or.inr q => ...`

**Exercise 3.3.** `Not P` is *defined* as `P -> False`. Using only that, prove `P -> Not (Not P)`. Write down in one sentence what program you just wrote.

**Exercise 3.4.** (Thought) Explain to a skeptical friend why a type with no constructors is the right representation of falsehood — and what would go wrong with the whole edifice if someone added a constructor to it.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 3.1.

*Pathway.* Draw the shapes before writing the term. You *have* a nested pair $((p, q), r)$; you *owe* $(p, (q, r))$. The projections `.1`/`.2` take pairs apart; `And.intro` puts them together; the whole proof is re-parenthesizing a package. Chase each component: where does $p$ live inside the input? At `h.1.1`.

*Answer.*

```
theorem and_assoc' : (P ∧ Q) ∧ R → P ∧ (Q ∧ R) :=
  fun h => And.intro h.1.1 (And.intro h.1.2 h.2)
```

Read it back as prose: "from $((p, q), r)$, produce $(p, (q, r))$." If your version has the components in a different arrangement, type-check it by hand as in the worked example — exactly one arrangement survives step 4.

### Solution 3.2.

*Pathway.* A disjunction is a *tagged* value — you cannot project both sides out of it (only one exists!), so the projection style of 3.1 is unavailable. The only way to consume `P ∨ Q` is case analysis: handle

each tag. In term mode that is a `match` with two arms; each arm holds evidence for one side and must rebuild the output disjunction with the *other* tag.

*Answer.*

```
theorem or_swap : P ∨ Q → Q ∨ P :=
  fun h => match h with
    | Or.inl p => Or.inr p
    | Or.inr q => Or.inl q
```

Note the tag flip: evidence that arrived tagged "left" leaves tagged "right" and vice versa. The compiler checks *exhaustiveness* — delete an arm and the proof is rejected, because a case of reality would be unhandled. Compare with `and_swap`: same theorem shape, entirely different evidence plumbing. The connective dictates the program.

### Solution 3.3.

*Pathway.* Unfold the abbreviation twice. $\neg P = P \to \texttt{False}$, so $\neg\neg P = (P \to \texttt{False}) \to \texttt{False}$. The goal $P \to \neg\neg P$ is therefore $P \to (P \to \texttt{False}) \to \texttt{False}$: two arguments — a proof and a refuter — and you must produce `False`. There is exactly one way to get a `False`: apply the refuter.

*Answer.*

```
theorem not_not_intro : P → ¬¬P :=
  fun p f => f p
```

The one-sentence description: *"given evidence $p$ and any would-be refutation $f$ of $P$, feed $p$ to $f$ — the refutation defeats itself."* It is modus ponens wearing a philosophy costume. (The converse, $\neg\neg P \to P$, is *not* writable in this plain style — you would have to conjure a $p$ out of a function that only consumes them. That is the constructive/classical boundary of the last section, met in the wild.)

### Solution 3.4.

*Pathway.* Recall what a proof of `False` would be — a value of the empty type — and follow the consequences of the type not being empty anymore.

*Answer.* (Model answer.) "Falsehood must be the proposition with *no evidence*: if you could construct a proof of `False`, 'proof' would stop meaning anything. In this system that is not a moral claim but a structural one — `False` is an inductive type with zero constructors, so no closed term of that type exists. Every other proposition's strength derives from this emptiness: $\neg P$ means $P \to \texttt{False}$, and the principle 'from `False`, anything' is safe precisely because its premise is unreachable. Add one constructor `oops : False` and watch the edifice fall: `False.elim oops` now proves *every* proposition — $1 = 2$, 'this code is correct,' everything — and every certificate ever checked becomes worthless simultaneously. The kernel's whole job is to be the thing that never lets `oops` in." Connect this forward: Chapter 11's `#print axioms` audit is, at bottom, checking that nobody smuggled in an `oops` dressed as an axiom.

**⋈ Checkpoint**

You should now be able to: translate each logical connective into its type ($\to$, $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$); write small proofs as terms; explain why `rfl` proves `1 + 1 = 2` but not `0 + n = n`; and articulate the Curry–Howard slogan — *proofs are programs, propositions are types, checking is type-checking* — with a straight face and genuine conviction.

# Chapter 4

# Tactics: Proving as a Dialogue

## 4.1 From programs to conversations

Writing proofs as raw programs, as in Chapter 3, is honest work, but it scales badly: a real correctness proof for field multiplication would be a program the size of a small compiler. Nobody writes those by hand. Instead, Lean offers *tactic mode*: an interactive dialogue where you issue commands and Lean builds the proof program for you, step by step, showing you the remaining work after each move.

You enter the dialogue with the keyword `by`:

```
theorem and_swap (P Q : Prop) : P ∧ Q → Q ∧ P := by
  intro h
  constructor
  · exact h.2
  · exact h.1
```

Place your cursor after `by` in the editor and Lean shows the **goal state** — the exact logical situation at that point:

```
P Q : Prop
⊢ P ∧ Q → Q ∧ P
```

Everything above the turnstile ⊢ is what you *have* (the context); the line after it is what you *owe* (the goal). Every tactic transforms this picture. After `intro h`, the hypothesis moves above the line; after `constructor`, the goal splits in two. Proving becomes a game whose board you can always see.

**∗ The big idea**

A tactic proof is a **recorded conversation with the goal state**. The skill of proving is not memorizing tactic names — it is learning to *read the goal state* and recognize which of a handful of moves makes it simpler. Below is the core vocabulary; it covers the vast majority of every proof in the real Ed25519 development.

| Tactic | When the goal looks like… | Effect |
|---|---|---|
| `intro h` | `P → Q`, `∀ x, P x` | assume it; name the evidence |
| `exact e` | anything | finish: `e` is a proof of the goal |
| `apply f` | `Q`, given `f : P → Q` | reduce the goal to `P` |
| `constructor` | `P ∧ Q`, `P ↔ Q`, … | split into pieces |
| `cases h` | have `h : P ∨ Q` (or `∧`, `∃`) | case analysis on `h` |
| `rw [eq]` | contains a rewritable subterm | replace using equation `eq` |
| `simp` | simplifiable clutter | rewrite with a lemma database |
| `induction n` | `∀ n : Nat, …` | base case + inductive step |
| `rfl` | `a = a` after computation | close by computation |

## 4.2 Rewriting: equality as a tool

The workhorse tactic of equational reasoning is `rw` (rewrite). Given a proven equation, it replaces one side by the other inside your goal:

```
example (a b : Nat) (h : a = b) : a + a = b + b := by
  rw [h]      -- goal becomes: b + b = b + b, closed by rfl automatically
```

Chains of rewrites read like the two-column proofs of school geometry, except a machine checks every line. Here is commutativity-and-associativity shuffling, Mathlib lemmas by name:

```
example (a b c : Nat) : a + b + c = c + b + a := by
  rw [Nat.add_comm a b]        -- b + a + c = c + b + a
  rw [Nat.add_assoc]           -- b + (a + c) = c + b + a
  rw [Nat.add_comm a c]        -- b + (c + a) = c + b + a
  rw [← Nat.add_assoc]          -- b + c + a = c + b + a
  rw [Nat.add_comm b c]        -- done
```

The arrow ← rewrites right-to-left. Nobody enjoys writing five-line shuffles like this, which is exactly why Chapter 5 introduces `ring` — but you must *once* feel the manual version to understand what the automation is doing on your behalf.

## 4.3 Induction: the tactic that conquers infinity

Remember the embarrassment of Chapter 3: `0 + n = n` does not hold by computation. Now we can prove it — by induction, the proof technique that inductive types were born for:

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero      => rfl                -- 0 + 0 = 0: computes
  | succ k ih => rw [Nat.add_succ, ih]
```

The `induction` tactic converts a statement about *all* naturals into two finite obligations: the statement

for `zero`, and the statement for `succ k` *assuming it for* `k` (the induction hypothesis `ih`). Because every natural number is built from those two constructors, the two cases cover infinity.

### ∠ Pen and paper: the full board of `zero_add` — every goal state, by hand

Reading a finished tactic proof is like reading a chess score without a board. Here is the same proof *with* the board: every goal state written out, exactly as the editor would show it. Reproduce this trace on paper until the transitions feel inevitable — it is the core skill of this chapter. We prove `0 + n = n`, where `+` recurses on its second argument (Chapter 2).

After `induction n with`, two boards appear. *Case* `zero`:

```
⊢ 0 + 0 = 0
```

The second argument is the literal `0`, so the first defining equation fires: `0 + 0` *computes* to `0`. Both sides are now the same term; `rfl` closes the board. (Compare the worked example in Chapter 3: this is the "computation sees it" half.)

*Case* `succ`: the board now carries an assumption — the induction hypothesis:

```
k : Nat
ih : 0 + k = k
⊢ 0 + (k + 1) = k + 1
```

The left side is `add 0 (succ k)`; the *second* defining equation fires and rewrites it to `succ (0 + k)`. That step is the tactic `rw [Nat.add_succ]` (or one layer of `simp only [Nat.add]`), and the board becomes:

```
k : Nat
ih : 0 + k = k
⊢ (0 + k) + 1 = k + 1
```

Now the subterm `0 + k` is *exactly* the left side of `ih`. Rewriting with `rw [ih]` replaces it:

```
k : Nat
ih : 0 + k = k
⊢ k + 1 = k + 1
```

Identical sides — `rw` auto-closes with `rfl`. Done.

Two habits to take from this trace. First, at every step ask "which defining equation *can* fire, and on which subterm?" — that question, not tactic names, is what drives the proof. Second, notice where the induction hypothesis earned its keep: computation alone got stuck at `0 + k` (opaque variable — Chapter 3's blocked reduction), and `ih` is precisely the permission to cross that gap. An induction proof is computation, plus permission slips at the stuck points.

**⋆ Aha**

Induction is not a new axiom to swallow — it falls out of the inductive definition of `Nat` itself. "Every `Nat` is `zero` or a `succ`" *is* the license to do case analysis; recursion on the structure *is* the induction. Data and proof principle are two views of the same declaration. This is Curry–Howard paying rent again.

**∠ Pen and paper: a debugging session, reconstructed honestly**

Here is a stuck proof, exactly as it happens to everyone, worked through with the discipline this chapter preaches. Goal: every number is at most its double — `∀ n : Nat, n ≤ 2 * n`. Attempt one:

```
theorem le_double (n : Nat) : n ≤ 2 * n := by
  induction n with
  | zero => rfl                -- fine: 0 ≤ 0 checks
  | succ k ih => rw [ih]       -- ERROR
```

The error says `rw` failed: `ih` is `k ≤ 2 * k` — an *inequality*, and `rw` rewrites with *equations* only. First lesson banked: the tactic wasn't wrong, the *move category* was — inequalities compose by transitivity and monotonicity, not substitution. Attempt two, reading the goal state first this time:

```
k : Nat
ih : k ≤ 2 * k
⊢ k + 1 ≤ 2 * (k + 1)
```

Plan on paper before touching the keyboard: $2(k{+}1) = 2k{+}2$, and from `ih`, $k{+}1 \le 2k{+}1 \le 2k{+}2$ ✓ — a two-step transitivity chain. That *plan* is provable many ways (`calc`, `Nat.succ_le_succ` plus lemmas)… at which point the honest practitioner pauses: hypotheses and goal are all *linear arithmetic*. The entire induction was unnecessary:

```
theorem le_double (n : Nat) : n ≤ 2 * n := by omega
```

Third lesson, the big one: getting stuck is often the discovery that you are proving something in the wrong *fragment* — and the next chapter is precisely the field guide to fragments. Keep the session's order of operations, though, because it generalizes: (1) read what the error says about the move category; (2) write the goal state and plan on paper; (3) only then ask whether a decision procedure owns the whole goal. Ten seconds of (3) before an hour of (2) is the professional's cheat — but (2) is what makes you able to survive when (3) says no.

**△ Pitfall**

When a proof gets stuck, resist the urge to try random tactics — the formal-methods equivalent of mashing buttons. The goal state is telling you something. Three honest questions unstick most situations: (1) Is the statement actually true as written — check a small example with `#eval`! (2) Am I missing a hypothesis — is there an unstated bound or nonzero condition?

(3) Is my induction on the right variable? In the real projects behind this book, "the proof is stuck" was, more often than not, the *statement* being subtly wrong — a truncated subtraction, a missing bound. The proof assistant was the messenger.

## 4.4 Structuring real proofs: `have` and `calc`

Big proofs are not flat lists of tactics; they are structured arguments with named intermediate results. The `have` tactic states and proves a stepping stone; `calc` lays out a chain of equalities or inequalities the way you would on a whiteboard:

```
example (a b : Nat) (h : a = 2 * b) : a + a = 4 * b := by
  have h2 : a + a = 2 * a := by rw [Nat.two_mul]
  calc a + a = 2 * a       := h2
       _     = 2 * (2 * b) := by rw [h]
       _     = 4 * b       := by rw [← Nat.mul_assoc]
```

### ∠ Pen and paper: planning a `calc` on paper — backwards from the target

Professionals do not type a `calc` top-to-bottom and hope; they plan it on paper, usually *backwards*. Watch the method on the real reduction identity from the Ed25519 code (Chapter 6 proves the ingredients): given a 256-bit value split as $x = x_{\text{lo}} + 2^{255}x_{\text{hi}}$, the code returns $x_{\text{lo}} + 19x_{\text{hi}}$, and we owe

$$x_{\text{lo}} + 19\,x_{\text{hi}} \;\equiv\; x \pmod{p}.$$

Plan backwards: the target's right side is $x$, which we only know through its *definition*, so the last line of the chain will be "$\ldots = x_{\text{lo}} + 2^{255}x_{\text{hi}} = x$ by the split." What could precede it? Something that turns 19 into $2^{255}$: the fact $19 \equiv 2^{255}$, i.e. the constant identity $2^{255} - 19 = p \equiv 0$. So the middle step is a congruence rewrite, and the paper plan reads, bottom to top:

| | | |
|---|---|---|
| (3) | $x_{\text{lo}} + 2^{255}x_{\text{hi}} = x$ | definition of the split |
| (2) | $x_{\text{lo}} + 19\,x_{\text{hi}} \equiv x_{\text{lo}} + 2^{255}x_{\text{hi}}$ | since $19 \equiv 2^{255} \pmod{p}$ |
| (1) | start: $x_{\text{lo}} + 19\,x_{\text{hi}}$ | |

Reverse it, and the `calc` types itself — each line's justification was decided *before* any Lean was written. The habit scales: the real multiplication proof in `dalek-ed25519-verified` was planned exactly this way, as a page of paper algebra whose lines became `have`s. When you cannot plan the chain on paper, you are not ready to type it; the proof assistant checks reasoning, it does not supply it.

This style is not cosmetic. In the verified field arithmetic you will read later, a single multiplication correctness proof is a `calc` chain tracking limb products through carries — dozens of steps, each trivial, whose *composition* is the theorem. `have` and `calc` are how proofs stay readable at that scale; they are also how they stay *maintainable*, because a broken step localizes the damage to one line.

There is one more structuring fact worth knowing early, because it saved the real project from a crash-course (literally — see Chapter 11): breaking a proof into small named `have` steps also controls the proof assistant's *memory appetite*. A monolithic "figure it all out at once" tactic call over a huge

context can consume gigabytes; ten targeted steps, pennies each, prove the same thing. Structure is not just style — it is engineering.

**>_ Try it yourself**

Open `exercises/Ch04.lean`. It sets up each theorem with the goal state drawn in a comment, then asks you to: prove `and_swap` in tactic mode; prove `zero_add` *without* peeking above; and repair a broken `calc` chain in which exactly one step is wrong. The third exercise is secretly the most realistic job training in this book.

## Exercises

**Exercise 4.1.** Prove by induction: `∀ n : Nat, n + 0 = n` and `∀ n m : Nat, n + succ m = succ (n + m)`. (These are the mirror images of the definitional equations — the ones computation gives you for free — and together they yield commutativity.)

**Exercise 4.2.** Using the previous exercise, prove `∀ n m : Nat, n + m = m + n` by induction on `m`. Write out, in one prose sentence per case, what each branch of your proof says.

**Exercise 4.3.** Prove `∀ n : Nat, 2 * n = n + n` twice: once with `induction`, once with a single `rw` using a Mathlib lemma you find yourself (search hint: `exact?` asks Lean to search for you).

**Exercise 4.4.** (Reading) In the goal state `h : a < 2^51 ⊢ a * 19 < 2^56`, no induction is needed — this is pure arithmetic. Which tactic from the table would you *guess* handles it? (Answer next chapter; your guess is the point.)

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 4.1.

*Pathway.* Before proving anything, predict which facts are *free*: addition recurses on its second argument, so any statement whose second argument exhibits a constructor should compute. Both requested statements — `n + 0 = n` and `n + succ m = succ (n + m)` — have constructor-shaped second arguments (`0` and `succ m`). The honest content of this exercise is noticing that.

*Answer.* `theorem a : ∀ n : Nat, n + 0 = n := fun n => rfl` — the first defining equation. `theorem b : ∀ n m, n + succ m = succ (n+m) := fun n m => rfl` — the second. Neither needs induction; both are the definitional mirror images of the two facts that *do* (`0 + n = n`, `succ n + m = succ (n + m)`), which is exactly the asymmetry the worked example traced. If you reached for `induction` here, nothing is wrong — it succeeds — but recognizing a free fact saves you thirty seconds a hundred times a day.

### Solution 4.2.

*Pathway.* Induct on `m` (the variable the recursion consumes on the *right* of `n + m`), so the definitional equations fire on one side and 4.1's lemmas patch the other. In each case, before touching tactics, write the goal and ask: which side steps by definition, which needs a lemma?

*Answer.*

```
theorem add_comm' (n m : Nat) : n + m = m + n := by
  induction m with
  | zero      => rw [Nat.add_zero, Nat.zero_add]
  | succ k ih => rw [Nat.add_succ, Nat.succ_add, ih]
```

Prose, one sentence per case, as requested. *Zero case:* "$n + 0$ is $n$ by definition, and $0 + n$ is $n$ by the mirror lemma, so both sides are $n$." *Successor case:* "both sides compute to a successor — the left by the defining equation, the right by the mirror lemma — and under the `succ` the two sides are the induction hypothesis." Note the architecture: two definitional equations, two mirror lemmas, one induction — commutativity is not one fact but a small *ecosystem*, and you have now built all of it from bare constructors.

### Solution 4.3.

*Pathway.* For the induction route, the `succ` case needs $2(k+1) = (k+1) + (k+1)$ reorganized into the induction hypothesis's shape plus successors — expect two rewrites. For the library route, the skill is *search*: place the cursor on the goal and run `exact?`.

*Answer.* Induction:

```
theorem two_mul' (n : Nat) : 2 * n = n + n := by
  induction n with
  | zero      => rfl
  | succ k ih => rw [Nat.mul_succ, ih, Nat.succ_add, Nat.add_succ]
```

Library: `exact?` finds `Nat.two_mul`, so `theorem two_mul" (n : Nat) : 2 * n = n + n := Nat.two_mul n`. Both are legitimate craft: the first when you are building the ecosystem, the second when you are using it. Mathlib has over 200,000 lemmas — searching *is* a proof technique, and `exact?` is its tactic.

### Solution 4.4.

*Pathway.* The goal mixes a hypothesis bound, multiplication by a *constant*, and powers of two — inspect the table: nothing fits perfectly, which is the setup's point. What you want is a decision procedure for linear arithmetic.

*Answer.* The intended guess is `simp` or `rw`-with-lemmas — and the intended discovery is that neither is pleasant. The actual answer, one chapter ahead: `omega`, which decides such goals instantly because $a \cdot 19$ is *linear* in $a$ (19 is a constant). If you guessed `omega` from the table's absence, better still: you have started classifying goals by *logical fragment*, which is precisely the professional habit Chapter 5 installs.

**⋈ Checkpoint**

You should now be able to: read a goal state (context, turnstile, goal); drive the core tactics `intro`, `exact`, `apply`, `cases`, `rw`, `induction`; structure a multi-step argument with `have` and `calc`; and — most importantly — when stuck, interrogate the *statement* before blaming the proof.

# Chapter 5

# Numbers and Automation: Making the Machine Do the Boring Parts

## 5.1 The 90/10 rule of verification

Here is a trade secret: most of a real verification effort is not clever. Opening the correctness proof of Ed25519 field multiplication, you will find that the overwhelming majority of proof obligations are statements like

$$a < 2^{51} \ \wedge \ b < 2^{51} \implies a + b < 2^{52}, \qquad\qquad (a + b) \cdot c = a \cdot c + b \cdot c,$$

— bookkeeping a patient undergraduate could verify by hand in a minute each. There are *thousands* of them. The craft of modern verification is to hand exactly this 90% to decision procedures — tactics that implement a complete algorithm for a well-defined logical fragment — and save the human for the 10% that needs insight. This chapter is your tour of the arsenal.

## 5.2 `omega`: linear arithmetic, decided

The tactic you will use more than any other in this book's domain is `omega`. It completely decides *linear arithmetic* over integers and naturals: any goal built from variables, constants, $+$, $-$, multiplication *by constants*, $=$, $<$, $\leq$, $\neg$, $\wedge$, $\vee$, including the hypotheses in context.

```
example (a b : Nat) (h1 : a < 2^51) (h2 : b < 2^51) :
    a + b < 2^52 := by omega

example (a b : Nat) (h : a ≤ b) : a + (b - a) = b := by omega
  -- note: truncated Nat subtraction handled CORRECTLY -- omega knows
```

That second example deserves a salute: `omega` understands `Nat` truncation natively, defusing the Chapter 2 pitfall by algorithm rather than by vigilance. When the goal is false, `omega` *fails* — it is a decision procedure, so failure on a linear goal means the goal (with the hypotheses in view) is simply not true. That property turns `omega` into a *statement-debugging* tool: if it refuses your "obviously

true" bound, go find the counterexample; there is one.

### ∠ Pen and paper: when omega says no — hunting the boundary by hand

Train the counterexample reflex on a real-shaped bound. Claim: "if $a, b < 2^{51}$ then $a+b < 2^{52}-2$." It *sounds* safely true — each term is below half of $2^{52}$, surely the sum clears the bar with room? `omega` refuses. Instead of doubting the tool, hunt at the *boundary*, because linear claims fail at corners of the allowed region: take both variables at their maximum, $a = b = 2^{51} - 1$. Then

$$a + b \;=\; 2^{52} - 2 \;\not<\; 2^{52} - 2.$$

The claim fails by exactly one at exactly one corner — the true theorem is $a + b \le 2^{52} - 2$, or strictly $a + b < 2^{52} - 1$. Now recall Chapter 1: the carry bugs that motivated this book are precisely off-by-a-hair failures at corners of the input region that sampling never visits. The refusal of a decision procedure is the same boundary being found for you, before the code ships instead of after. Standing rule: **when `omega` refuses a linear goal, the statement is wrong; evaluate the goal at the extreme values of every hypothesis, and one of those corners is your counterexample.**

### ∠ Pen and paper: the bound-then-omega pattern at real field scale

Here is the exact arithmetic that stands between the Ed25519 multiplication routine and a machine-word overflow — the central safety computation of the whole field layer, fully pen-and-paper. The bounds discipline lets limbs grow to $a_i, b_j < 2^{54}$ (Chapter 8). Multiplication forms *column sums* of limb products (Chapter 9); the widest column is

$$c_4 \;=\; a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 \qquad \text{(five products)}.$$

The products land in 128-bit words. Does $c_4$ fit? Work the exponents:

$$a_ib_j \;<\; 2^{54} \cdot 2^{54} \;=\; 2^{108}, \qquad c_4 \;<\; 5 \cdot 2^{108} \;<\; 2^3 \cdot 2^{108} \;=\; 2^{111} \;<\; 2^{128}. \checkmark$$

Margin: $128 - 111 = 17$ bits to spare — room for the carry-in each column also absorbs. Every number in this computation is real: the 54 comes from the invariant, the 5 from the limb count, the 128 from the hardware. Change any one — say a hypothetical radix-58 variant: $2 \cdot 58 = 116$, $5 \cdot 2^{116} < 2^{119}$, still fine; radix-62: $5 \cdot 2^{124} < 2^{127}$, one bit from the cliff — and you can re-run the audit in your head. *This* is what the automation divides between itself: `a_i * b_j < 2^108` is the nonlinear atom you must name (one monotonicity lemma), and everything after it — $5 \cdot 2^{108} < 2^{128}$ with the atom opaque — is linear, which is to say, `omega` food. The pattern in the pitfall below is this computation, industrialized.

### △ Pitfall

`omega` does not touch multiplication of two *variables* ($a \cdot b$ is not linear), division in general, or bit-shifts by variables. For a goal mixing $a \cdot b$ with bounds, you often first name the product — `have hab : a * b ≤ 2^102 := ...` using a multiplication monotonicity lemma — and then let `omega` finish with `hab` as an opaque atom. This two-step, *bound the nonlinear part, then release the linear solver*, is the single most-used proof pattern in verified field arithmetic. You will write it dozens of times, and by Chapter 10 it will feel like breathing.

## 5.3 `decide`: when truth is a computation

Some propositions can be checked by running an algorithm to completion: "97 is prime," "these two sorted lists are equal," "$x^3 = x$ for all $x$ in $\mathbb{Z}/6\mathbb{Z}$" (six cases — check them all). For any such *decidable* proposition, the `decide` tactic runs the decision algorithm inside Lean's kernel and turns the answer into a proof:

```
example : Nat.Prime 97 := by decide
example : ∀ x : ZMod 6, x^3 = x^3 := by decide   -- finite: try all six
```

The magic and the limitation are the same fact: the *kernel itself* re-executes the computation. That makes `decide` unimpeachable — and completely hopeless for our 77-digit prime $2^{255} - 19$, where trial division would outlast the universe. There is a variant, `native_decide`, that compiles the check to native code first — fast enough! — but it makes the compiler part of your trusted base, an IOU we will scrutinize hard in Chapters 7 and 11. For now, the rule of the house: **`decide` yes, `native_decide` never in a final certificate.**

## 5.4 `ring` and `norm_num`: algebra on tap

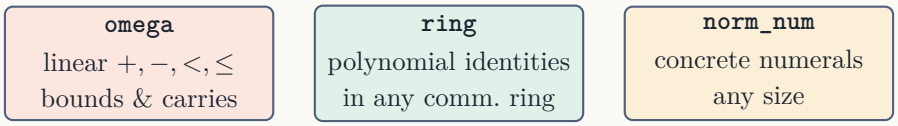
The five-line commutativity shuffle from Chapter 4? Here is the grown-up version:

```
example (a b c : Nat) : a + b + c = c + b + a := by ring

example (a b : ZMod p) : (a + b)^2 = a^2 + 2*a*b + b^2 := by ring

example : (2:Int)^255 - 19 > 2^254 := by norm_num
```

`ring` proves any identity that holds in every commutative ring — polynomial rearrangements, binomial expansions, distributivity avalanches — by normalizing both sides to a canonical polynomial form and comparing. `norm_num` evaluates concrete numeric facts, comfortable with numbers of any size. Between `omega`, `ring`, and `norm_num` you now hold the three keys that open most arithmetic doors:




the three keys of verified arithmetic — learn what each fragment *excludes* and you will always know which door you are standing in front of

## 5.5 `simp`: the rewriting engine, and how to hold it

`simp` rewrites the goal to exhaustion using a curated database of thousands of "simplification" lemmas ($x + 0 \rightsquigarrow x$, `List.length (a :: l)` $\rightsquigarrow$ `l.length + 1`, ...). It is the most powerful tactic in Lean and the easiest to misuse. Used well, it clears brush so the real argument stands out. Used lazily — `simp`

`[*]` with every hypothesis thrown in, in a context of sixty accumulated facts — it becomes a search over an enormous rewrite space: slow, fragile under library updates, and occasionally a memory monster.

This is not hypothetical. During the development this book accompanies, a single over-broad `simp`-style discharge in a fat context consumed twelve gigabytes of RAM and took down the machine. The postmortem produced house rules worth adopting from day one:

- Prefer `simp only [lemma1, lemma2]` — an explicit lemma list — in anything you intend to keep.
- Let `simp?` tell you the list: run it once interactively, then paste the `simp only [...]` it suggests into the file.
- Keep contexts lean (pun intended): a proof with sixty hypotheses in scope wants to be five `have`-steps with twelve each.

*** The big idea**

Automation is a *contract*, not a slot machine. Each tactic decides a known fragment: `omega` linear arithmetic, `ring` ring identities, `decide` finite computation, `simp only` a rewrite system you chose. The professional habit is to know *which* contract you are invoking — then failure is information ("this goal is not linear"; "this identity needs the modulus"), never mystery.

**>_ Try it yourself**

Open `exercises/Ch05.lean`: ten arithmetic goals, each solvable by exactly one of `omega` / `ring` / `norm_num` / `decide`. Your task is not just to close them but to close each with the *right* tool — the file rejects overkill by design. Goal number ten is the Chapter 4 cliffhanger: `a < 2^51 → a * 19 < 2^56`. (It is not linear — 19 is a constant, so it is! Think, then fire.)

## Exercises

**Exercise 5.1.** For each, name the tactic and predict success before running: (a) `(a+b)*(a-b) = a*a - b*b` over `Int`; (b) `a < 100 → b < 100 → a*b < 10000` over `Nat`; (c) `Nat.Prime 65537`; (d) `2^51 + 2^51 = 2^52`.

**Exercise 5.2.** Goal (b) above is nonlinear, yet `omega` alone fails while the two-step pattern (bound the product with `Nat.mul_lt_mul` machinery, then `omega`) succeeds. Carry it out. Time yourself; the pattern should take under five minutes by the second attempt.

**Exercise 5.3.** Find a true statement about `Nat` that *no* tactic in this chapter proves in one shot, and sketch in prose how you would decompose it. (Anything genuinely inductive works — automation here decides arithmetic fragments, not all of mathematics.)

**Exercise 5.4.** (Paper) Re-run the worked column-sum audit for the *scalar* arithmetic of the companion projects, which uses radix-52 limbs bounded by $2^{52}$ multiplied into 128-bit words with *nine*-term columns at the widest (schoolbook $5 \times 5$, column $k = 4$ has five terms, but the Montgomery pass adds four more products). Does $9 \cdot 2^{104}$ fit in $2^{128}$? With how many bits of margin?

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 5.1.

*Pathway.* For each goal, name the *fragment* it lives in before naming a tactic: linear with constants? polynomial identity? concrete numerals? finite check? Then the table from this chapter is a lookup, not a guess.

*Answer.* (a) $(a+b)(a-b) = a\,a - b\,b$ over `Int`: polynomial identity in a commutative ring → `ring`. Succeeds. (Over `Nat` it would be trickier — truncated subtraction breaks ring reasoning; part of why the exercise says `Int`.) (b) $a < 100 \to b < 100 \to ab < 10000$: *nonlinear* (product of two variables) → no single tool from the table; needs the two-step pattern (5.2). (c) `Nat.Prime 65537`: decidable, and small enough → `decide` works (with a noticeable pause); `norm_num` is faster — Chapter 7 explains the difference as certificate versus grind. (d) $2^{51} + 2^{51} = 2^{52}$: concrete numerals → `norm_num` (or `decide`; but build the habit of matching tool to fragment, not firing the biggest gun).

### Solution 5.2.

*Pathway.* The product $ab$ is one opaque quantity as far as linear reasoning is concerned. So: (i) bound the atom with a monotonicity lemma — both factors grow, so the product grows; (ii) hand the bounded atom to `omega`. Finding the lemma name is part of the exercise: `exact?` on the goal `a * b < 100 * 100`, or search Mathlib for "mul_lt_mul".

*Answer.*

```
example (a b : Nat) (ha : a < 100) (hb : b < 100) : a * b < 10000 := by
  have hab : a * b < 100 * 100 := Nat.mul_lt_mul'' ha hb
  omega
```

Two lines, and the division of labor is visible: the `have` names the one nonlinear fact (with the lemma doing the monotonicity), then `omega` finishes since `100 * 100` is a numeral and `a * b` is now an opaque atom below it. Under five minutes by the second attempt is a realistic bar — this exact two-line shape appears *dozens* of times in the field-layer proofs, with $2^{54}$ in place of 100.

### Solution 5.3.

*Pathway.* Look for statements whose truth genuinely needs induction — i.e. not expressible in a decided fragment. Anything universally quantified over *all* `n` about a recursively defined function qualifies.

*Answer.* (Model answer.) $\forall n$, $\texttt{fib}(n) < 2^n$ (with `fib` from Chapter 2). Not linear (contains `fib` and $2^n$), not a ring identity, not concrete, not finite — every tool in this chapter shrugs. Decomposition sketch: induction on $n$ with *two* base cases ($n = 0, 1$) and a step using $\texttt{fib}(n{+}2) = \texttt{fib}(n{+}1) + \texttt{fib}(n) < 2^{n+1} + 2^n < 2^{n+2}$ — where the *final* inequality, with the induction hypotheses as opaque atoms, IS an `omega` goal. That is the grown-up division of labor: induction supplies the skeleton, decision procedures clear each rib.

**Solution 5.4.**

*Pathway.* Same three lines as the worked example; only the constants move.

*Answer.* Products: $2^{52} \cdot 2^{52} = 2^{104}$. Nine terms: $9 \cdot 2^{104} < 2^4 \cdot 2^{104} = 2^{108} < 2^{128}$ — fits, with $128 - 108 = 20$ bits of margin (23 if you count $9 < 2^{3.17}$ more tightly). The scalar layer breathes easier than the field layer's 17 bits, which matches how the two codebases feel to verify. When you meet the scalar Montgomery proofs in the companion repos, this margin computation is the first `have` of every multiplication lemma.

**⋈ Checkpoint**

You should now be able to: match a goal to its decision procedure by the shape of its operators; execute the bound-then-omega pattern for nonlinear bounds; explain why `decide` is trustworthy and where it hits its computational wall; and state the `simp` discipline — and the story of why this book is unusually sincere about it.

# Chapter 6

# Modular Arithmetic: The Number System Cryptography Lives In

## 6.1 Clock arithmetic, taken seriously

You already compute modulo twelve every day: four hours after ten o'clock is two o'clock. Wrap-around arithmetic — add, overflow the dial, keep the remainder — is the entire idea of *modular arithmetic*. Cryptography's only twist is the size of the clock: Ed25519's dial has

$$p = 2^{255} - 19$$

hours on it — a 77-digit prime. Everything else — the wrap-around, the remainder-taking, the algebra — is the twelve-hour clock you already know.

Formally, we write $a \equiv b \pmod{n}$ when $n$ divides $a - b$, and we collect all integers with the same remainder into one object: the world $\mathbb{Z}/n\mathbb{Z}$ has exactly $n$ elements, $\{0, 1, \ldots, n-1\}$, with addition and multiplication that wrap. In Lean/Mathlib this world is a first-class type, and computation in it is exactly what you would hope:

```
#eval (7 + 8 : ZMod 12)    -- 3     (fifteen o'clock is three o'clock)
#eval (5 * 9 : ZMod 12)    -- 9
#eval (3 - 7 : ZMod 12)    -- 8     (subtraction wraps -- no truncation!)

def p : Nat := 2^255 - 19
#eval (2^255 : ZMod p)     -- 19    (of course: 2^255 = p + 19)
```

Note the third line with relief: unlike `Nat`, subtraction in `ZMod n` is a total, well-behaved inverse of addition. Every element has a negative. In algebra terms, $\mathbb{Z}/n\mathbb{Z}$ is a *commutative ring* — and Lean knows it, so the `ring` tactic from Chapter 5 works there natively.

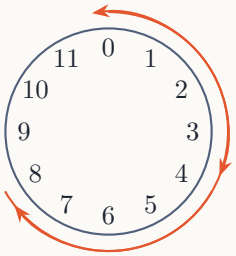
## 6.2 Division: where primes enter

Rings give us $+$, $-$, $\times$. Cryptography also needs $\div$ — and division is where the modulus stops being a size parameter and starts being a design decision. Dividing by $a$ means multiplying by some $a^{-1}$

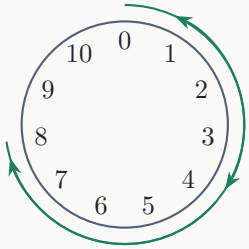
with $a \cdot a^{-1} = 1$. Does such an inverse exist? On the twelve-hour clock, try to invert 4: the multiples of 4 are $4, 8, 0, 4, 8, 0, \ldots$ — the value 1 never appears. 4 has no inverse mod 12, because 4 and 12 share the factor 4.

**∗ The big idea**

In $\mathbb{Z}/n\mathbb{Z}$, the element $a$ is invertible exactly when $\gcd(a, n) = 1$. So if $n = p$ is **prime**, *every* nonzero element is invertible: you can divide freely, and $\mathbb{Z}/p\mathbb{Z}$ earns the title of **field**, written $\mathbb{F}_p$. Fields are the arithmetic paradise where linear algebra, polynomial factoring, and elliptic-curve geometry all work. This — and only this — is why cryptographic moduli are prime: primality is the entry ticket to division.




$\mathbb{Z}/12\mathbb{Z}$: stepping by 4 visits only $\{0, 4, 8\}$ — never 1




$\mathbb{Z}/11\mathbb{Z}$: stepping by 4 reaches 1 in three steps: $4^{-1} = 3$

How do you *find* $a^{-1}$ mod $p$? Two classical answers, both of which appear in real Ed25519 code: the extended Euclidean algorithm, and Fermat's little theorem, which says $a^{p-1} \equiv 1 \pmod{p}$ for $a \not\equiv 0$ — hence $a^{p-2}$ *is* the inverse. The dalek library computes inverses as $a^{p-2}$ with a hand-crafted chain of 254 squarings and 11 multiplications; one of the proofs you will meet in Chapter 10 verifies precisely that this chain computes what Fermat promises.

**∠ Pen and paper: inverting 19 modulo the 77-digit prime, entirely by hand**

A 77-digit modulus does not put pen-and-paper out of business — watch. We compute $19^{-1} \bmod p$ for the real $p = 2^{255} - 19$, exactly, in five lines of Euclid. The only preparation is one residue: *what is $p$ mod 19?* By Fermat's little theorem in the *small* field $\mathbb{Z}/19\mathbb{Z}$: $2^{18} \equiv 1$, and $255 = 14 \cdot 18 + 3$, so

$$2^{255} \equiv (2^{18})^{14} \cdot 2^3 \equiv 8 \pmod{19} \qquad \Longrightarrow \qquad p = 2^{255} - 19 \equiv 8 - 0 \equiv 8 \pmod{19}.$$

So $p = 19k + 8$ for the (77-digit, but never-written-out) integer $k = (p - 8)/19$. Now run Euclid on $(p, 19)$, keeping remainders only:

$$p = 19k + 8, \qquad 19 = 2 \cdot 8 + 3, \qquad 8 = 2 \cdot 3 + 2, \qquad 3 = 1 \cdot 2 + 1.$$

The gcd is 1 (of course — $p$ is prime). Back-substitute, bottom to top, expressing 1 in terms of each earlier pair:

$$\begin{aligned} 1 &= 3 - 2 \\ &= 3 - (8 - 2 \cdot 3) \;=\; 3 \cdot 3 - 8 \\ &= 3\,(19 - 2 \cdot 8) - 8 \;=\; 3 \cdot 19 - 7 \cdot 8 \\ &= 3 \cdot 19 - 7\,(p - 19k) \;=\; (3 + 7k) \cdot 19 \;-\; 7p. \end{aligned}$$

Read off the answer: $19 \cdot (3 + 7k) = 1 + 7p \equiv 1 \pmod{p}$, so

$$19^{-1} \bmod p \;=\; 3 + 7k \;=\; 3 + 7 \cdot \frac{p - 8}{19}.$$

This is an *exact, closed-form* answer about the real prime, computed with numbers that never exceeded two digits — the size of $p$ entered only through one residue. Two lessons. First, modular arithmetic is constantly this kind: gigantic numbers handled through small representatives — the same move the limb representation of Chapter 9 industrializes. Second, note what the computation *cost*: five divisions. Fermat's route costs 254 squarings of 77-digit numbers. Why does the real code use Fermat anyway? Because Euclid's division sequence *depends on the input value* — its running time leaks secrets through timing — while the Fermat chain executes identically for every input. Constant-time discipline pays a couple hundred multiplications of overhead for silence; you will meet this trade at every layer of the pyramid.

## 6.3 Why $2^{255} - 19$? A prime chosen for machines

Any large prime makes a field. Why this one? Because arithmetic mod $p$ is computed by *machines with 64-bit words*, and $p = 2^{255} - 19$ is shaped for them:

- **Reduction is a multiply-by-19.** Numbers at or beyond $2^{255}$ overflow the dial; but since $2^{255} \equiv 19 \pmod{p}$, any overflow bit re-enters the sum carrying a factor of just 19. Reducing mod $p$ costs one small multiplication — no long division, ever.
- **255 bits split evenly into five limbs of 51.** A 64-bit word holding a 51-bit limb leaves 13 bits of headroom for carries to accumulate lazily — the delayed-carry style from Chapter 1, now by design rather than accident. The bound $a_i < 2^{51+\varepsilon}$ will follow us through Chapters 9 and 10.
- **It sits just below a power of two**, so 255-bit values fit in 32 bytes with one bit to spare — and Ed25519 spends that spare bit storing a point's sign. Engineering all the way down.

**∠ Pen and paper: the multiply-by-19 reduction, performed on real overflow**

Do the reduction the code does, by hand, on the real modulus. The identity: $2^{255} = p + 19 \equiv 19 \pmod{p}$. Now take a genuinely overflowing value — say $2^{260}$, five bits past the edge, the kind of thing a lazy carry chain produces. Split off the overflow and fold:

$$2^{260} \;=\; 2^5 \cdot 2^{255} \;\equiv\; 2^5 \cdot 19 \;=\; 608 \pmod{p}.$$

One multiplication by a two-digit constant replaced a division by a 77-digit prime. Once more, with a mixed value: reduce $x = 2^{256} + 2^{255} + 7$:

$$x = (2 + 1) \cdot 2^{255} + 7 \;\equiv\; 3 \cdot 19 + 7 \;=\; 64 \pmod{p}.$$

And the general recipe, exactly as the code implements it: write $x = x_{\text{lo}} + 2^{255} x_{\text{hi}}$ (a bit-split, free in hardware); return $x_{\text{lo}} + 19\, x_{\text{hi}}$. Estimate the shrinkage like an engineer: if $x < 2^{300}$, then $x_{\text{hi}} < 2^{45}$ and the result is $< 2^{255} + 19 \cdot 2^{45} < 2^{256}$ — one pass tames 45 excess bits, and a second pass lands below $2^{256}$-ish for good. This split-multiply-add *is* the entire reduction

machinery of every curve25519 implementation on earth; you have now executed it. When Chapter 10 verifies `reduce`, the theorem is the congruence line above, quantified over every $x$ the bounds admit.

The Pasta curves verified in the companion projects (Pallas/Vesta, used in zero-knowledge proof systems) choose their $\approx 2^{254}$ primes by a different criterion — friendliness to fast Fourier transforms — and represent elements in *Montgomery form*, a representation trick we will touch in Chapter 9. Different shapes, same theory: pick a prime whose field your machine can love.

**∠ Pen and paper: shadow arithmetic — auditing big computations on small clocks**

Your grandmother's bookkeeper knew a modular-arithmetic trick worth resurrecting: *casting out nines*. Because $10 \equiv 1 \pmod 9$, a number is congruent mod 9 to its digit sum — so any claimed sum or product of large numbers can be spot-checked by comparing digit-sum shadows. The general principle: **any exact integer identity survives reduction mod anything**, so a cheap clock can audit an expensive computation. Watch it catch the classic transcription error — two leading digits transposed:

$$123456789 \times 987654321 \stackrel{?}{=} \underline{21}1932631112635269.$$

Shadow mod 9: digit-sum$(123456789) = 45 \equiv 0$, so the product's shadow must be $0\cdot(\text{anything}) = 0$; the claimed result's digit sum is $63 \equiv 0$ — *passes*. It would: transpositions never change a digit sum, which is exactly why bookkeepers paired nines with a second clock. Mod 11 ($10 \equiv -1$, so take the *alternating* digit sum from the right): each factor $\equiv 5 \pmod{11}$, so the product must be $5 \cdot 5 = 25 \equiv 3 \pmod{11}$; the claimed result's alternating sum gives $1 \not\equiv 3$ — **caught**. (The true product is 121932631112635269; the transposition slipped past one clock and not the other, which is the design lesson: independent shadows multiply their power, 1-in-9 and 1-in-11 escape odds compounding to 1-in-99.)

Why this card lives in this book: the verified field code is audited by the *same architecture*. The denotation bridge of Chapter 9 checks limb computations by their shadow in $\mathbb{Z}/p\mathbb{Z}$ — one gigantic well-chosen clock instead of two small ones — and the kernel plays the incorruptible bookkeeper. And when *you* debug a failing proof about a 77-digit computation, shadowing both sides mod 9 or mod 97 by hand remains the fastest way to find out *which* side is lying. (Exercise 6.6 hands you a broken carry chain to convict this way.)

## 6.4 Modular arithmetic in Lean, hands on

Statements about $\mathbb{F}_p$ are ordinary Lean theorems, and the automation you already own applies:

```
-- ring identities hold verbatim in ZMod n:
example (x y : ZMod 12) : (x + y)^2 = x^2 + 2*x*y + y^2 := by ring

-- small finite worlds: check every case
example : ∀ x : ZMod 12, 4 * x ≠ 1 := by decide

-- inverses exist in prime fields (Mathlib knows ZMod 11 is a field):
example (a : ZMod 11) (h : a ≠ 0) : a * a⁻¹ = 1 :=
```

```
  ZMod.mul_inv_cancel_of_ne_zero h
```

What about the crown fact, $2^{255} \equiv 19$? For *concrete numeric* statements at 77-digit scale, tactic choice suddenly matters: `decide` would ask the kernel to grind case analysis it cannot afford, while `norm_num` and reflexivity-by-computation on efficient numerals handle it comfortably. Verifying real cryptography is partly a *performance engineering* discipline — Chapter 7 turns this observation into a principle when the question becomes primality itself.

**△ Pitfall**

In `ZMod n`, the numeral `57896044618658...` and the numeral `19` can be *the same element*. Equality of elements is not equality of the numerals you typed. When a surprising `rfl` succeeds — or two "different" constants turn out equal — remember you are on the clock face, not the number line. This is a feature: the entire verification story of Chapter 9 consists of choosing, deliberately, when to view a pile of bytes as an integer and when as a clock position.

**>_ Try it yourself**

Open `exercises/Ch06.lean`. You will: compute your first inverses with `#eval`; expand $(x+y)^3$ in `ZMod 7` with one tactic; show 4 has no inverse in `ZMod 12` by `decide`; and prove the reduction identity $2^{255} = 19$ in `ZMod p`.

## Exercises

**Exercise 6.1.** Compute by hand (yes, hand): $3^{-1}$ in $\mathbb{Z}/7\mathbb{Z}$, and check that stepping by 3 on a 7-hour clock visits every position. Then confirm both with `#eval`.

**Exercise 6.2.** Prove in Lean: $\forall$ `x : ZMod 12, 4 * x` $\neq$ `1`, then change 12 to 13 and watch the same tactic refuse — find the witness it is implicitly pointing at.

**Exercise 6.3.** Fermat's little theorem is `ZMod.pow_card_sub_one_eq_one` in Mathlib. Use it to prove that in `ZMod 11`, `a^9 * a = 1` whenever `a` $\neq$ `0`.

**Exercise 6.4.** (Paper) The Ed25519 code reduces a 256-bit value $x$ by writing $x = x_{\text{low}} + 2^{255}\, x_{\text{high}}$ and returning $x_{\text{low}} + 19\, x_{\text{high}}$. Prove informally that this preserves the value mod $p$, and bound how much smaller the result is. You have just re-derived the reduction step you will verify formally in Chapter 10.

**Exercise 6.5.** (Paper) Run the worked Euclid computation for a different small constant: find $121665^{-1}$ mod $p$? No — that one needs more lines than a margin holds. Do the honest scaled version: compute $p$ mod 3 (via $2^{255}$ mod 3) and derive $3^{-1}$ mod $p$ in closed form, as the worked example did for 19. Check your first Euclid line by verifying the claimed residue.

**Exercise 6.6.** (Shadow audit) A limb computation claims $(2^{51} - 19) + (2^{51} - 1) = 2^{52} - 21$. Convict or acquit using two shadows: mod 9 and mod 5. (Powers of 2 cycle mod 9 with period 6 and mod 5 with period 4 — Card 3 of the toolkit generalizes.)

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 6.1.

*Pathway.* "Inverse of 3 mod 7" asks: three times what is one more than a multiple of 7? With seven candidates, enumerate — and while enumerating, watch the stepping-around-the-clock picture from the chapter figure happen.

*Answer.* $3 \cdot 5 = 15 = 2 \cdot 7 + 1$, so $3^{-1} = 5$ in $\mathbb{Z}/7\mathbb{Z}$. Stepping by 3 from 0: $0 \to 3 \to 6 \to 2 \to 5 \to 1 \to 4 \to 0$ — all seven positions visited before returning, because $\gcd(3, 7) = 1$. `#eval (3⁻¹ : ZMod 7)` confirms `5`. (If you try the same walk stepping by 4 on the 12-clock, you revisit 0 after three steps — the figure's left panel — and that early return *is* the nonexistence of the inverse.)

### Solution 6.2.

*Pathway.* Twelve cases is `decide` territory; the interest is in the refusal after you change the modulus. A decision procedure failing on a false statement is *pointing* at something — make it tell you what.

*Answer.* `example : ∀ x : ZMod 12, 4 * x ≠ 1 := by decide` succeeds. With modulus 13: `decide` fails, because the statement is now false — $\gcd(4, 13) = 1$, so an inverse exists. Find it as the worked examples teach: $4 \cdot 10 = 40 = 3 \cdot 13 + 1$, so the witness is $x = 10$, confirmed by `#eval (4⁻¹ : ZMod 13)` printing `10`. The general moral, third time now: a decision procedure's "no" is a counterexample's "here."

### Solution 6.3.

*Pathway.* Fermat gives $a^{10} = 1$ directly ($p - 1 = 10$ at $p = 11$); the exercise's $a^9 \cdot a$ is that same fact wearing a disguise — undo the disguise with exponent algebra ($a^9 \cdot a = a^{10}$), then apply the library lemma. Registering `Fact (Nat.Prime 11)` is the price of admission for the typeclass machinery.

*Answer.*

```
instance : Fact (Nat.Prime 11) := ⟨by decide⟩

example (a : ZMod 11) (h : a ≠ 0) : a^9 * a = 1 := by
  have hp := ZMod.pow_card_sub_one_eq_one h   -- a^10 = 1
  calc a^9 * a = a^10 := by ring
       _       = 1    := by simpa using hp
```

The two-step shape — *algebraic massage by* `ring`*, then the library fact* — is how nearly every "mathematical" step in the real proofs goes. The inversion-chain verification of Chapter 10 is this exercise, iterated 265 times.

### Solution 6.4.

*Pathway.* Congruence first (replace $2^{255}$ by 19 and check the difference is a multiple of $p$), size second (bound each summand).

*Answer.* *Congruence:* the returned value differs from $x$ by $x_{\text{high}} \cdot (2^{255} - 19) = x_{\text{high}} \cdot p$, a multiple of $p$ — so they are congruent. (That is the whole proof; write it as one displayed equation and admire how little was needed.) *Size:* for $x < 2^{256}$ we have $x_{\text{high}} < 2$, i.e. $x_{\text{high}} \in \{0, 1\}$, and $x_{\text{low}} < 2^{255}$, so the result is $< 2^{255} + 19 < 2^{256}$ — and if $x_{\text{high}} = 1$ the value dropped by $p - 19 \approx 2^{255}$: one pass halves the range. The formal statement in `dalek-ed25519-verified` carries exactly these two clauses — congruence and bound — the two-clause spec shape of Chapter 9, met here in miniature.

### Solution 6.5.

*Pathway.* Same three moves as the worked example: (i) small-field Fermat for the residue, (ii) Euclid (here trivially short), (iii) back-substitute into closed form.

*Answer.* Residue: $2 \equiv -1 \pmod 3$, so $2^{255} \equiv (-1)^{255} = -1 \equiv 2 \pmod 3$, hence $p = 2^{255} - 19 \equiv 2 - 1 \equiv 1 \pmod 3$ (since $19 \equiv 1 \bmod 3$). So $p = 3m + 1$ with $m = (p-1)/3$. Then $3m = p - 1 \equiv -1$, i.e. $3 \cdot (-m) \equiv 1$, giving

$$3^{-1} \bmod p \;=\; p - m \;=\; p - \frac{p-1}{3} \;=\; \frac{2p+1}{3}.$$

Sanity-check the closed form: $3 \cdot \frac{2p+1}{3} = 2p + 1 \equiv 1 \pmod p$ ✓. Notice the answer is one line shorter than the worked example's — because $p \bmod 3 = 1$ made Euclid collapse. The residue does all the work; the size of $p$ never mattered.

### Solution 6.6.

*Pathway.* Shadow each side independently; a mismatch on *any* clock convicts. Powers of 2 cycle: mod 9 with period 6 ($2^6 = 64 \equiv 1$), mod 5 with period 4 ($2^4 \equiv 1$) — reduce exponents by the period, exactly Card 3's move.

*Answer.* Mod 9: $51 \equiv 3 \pmod 6$ so $2^{51} \equiv 2^3 = 8$, and $19 \equiv 1$, so the left side is $\equiv (8 - 1) + (8 - 1) = 14 \equiv 5$. Claimed right side: $52 \equiv 4 \pmod 6$ so $2^{52} \equiv 2^4 = 7$, and $21 \equiv 3$, giving $7 - 3 = 4$. Shadow verdict: $5 \neq 4$ — **convicted**. Cross-examine mod 5: $2^{51} \equiv 2^3 = 3$ (period 4), left $\equiv (3 - 4) + (3 - 1) = -1 + 2 = 1$; right: $2^{52} \equiv 1$, $21 \equiv 1$, gives 0. Again $1 \neq 0$ — convicted twice. The true sum is $2^{52} - 20$ (check its shadows: mod 9: $7 - 2 = 5$ ✓; mod 5: $1 - 0 = 1$ ✓); the claim was off by exactly one — the species of error this book has been hunting since Chapter 1, caught this time with two clocks and thirty seconds.

**⋈ Checkpoint**

You should now be able to: compute in $\mathbb{Z}/n\mathbb{Z}$ and explain the notation; state exactly when division works and why primality guarantees it; give two independent reasons the constant 19 appears throughout curve25519 codebases; and prove small modular facts in Lean with `decide`, `ring`, and a Mathlib lemma found by name.

# Chapter 7

# Convincing a Paranoid Kernel That a 77-Digit Number Is Prime

## 7.1 The problem nobody warns you about

Chapter 6 ended with a quiet dependency: everything — division, field structure, the whole elliptic curve — rests on $p = 2^{255} - 19$ *being prime*. In Lean, that is a proposition like any other, and it must be *proved*:

```
theorem p_prime : Nat.Prime (2^255 - 19) := ?
```

Your Chapter 5 instincts say `decide`: primality is decidable — just try dividing. But trial division tests divisors up to $\sqrt{p} \approx 2^{127}$. At a billion billion divisions per second, that is about $10^{12}$ ages of the universe. The kernel, which happily *re-executes* every computation you feed it, cannot afford this one. And mathematicians clearly believe this number is prime — so how does *anyone* know?

## 7.2 Certificates: the deep idea hiding here

The answer reorganizes how you think about computation. *Finding* a fact and *checking* a fact can have wildly different costs. What we need is a **certificate**: a piece of data, possibly expensive to discover, that makes the fact *cheap to verify*.

You have met certificates before without the name. A composite number's certificate is a factor: finding a factor of a 77-digit number may be hard, but checking $n = a \cdot b$ is one multiplication. The beautiful surprise — Pratt's theorem, 1975 — is that *primality* has certificates too:

**⁎ The big idea**

**Pratt certificate.** To certify that $p$ is prime, exhibit a *witness* $w$ such that

$$w^{p-1} \equiv 1 \pmod{p} \qquad \text{and} \qquad w^{(p-1)/q} \not\equiv 1 \pmod{p} \ \text{ for every prime factor } q \text{ of } p-1.$$

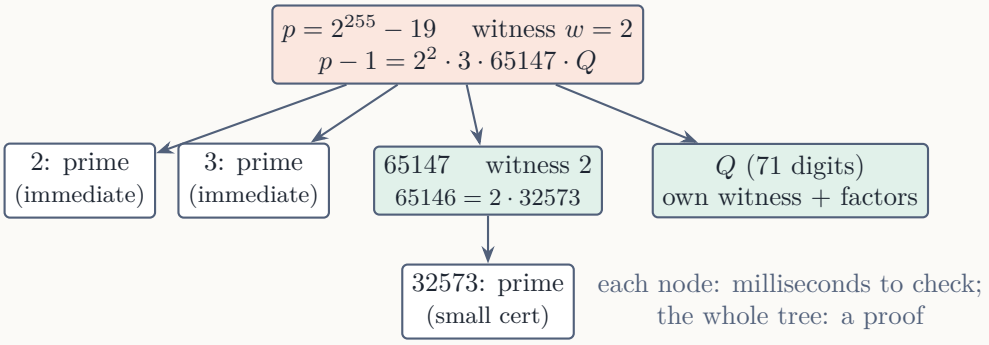
Such a $w$ generates all $p-1$ nonzero residues, which forces $\mathbb{Z}/p\mathbb{Z}$ to have $p-1$ invertible elements

— something only a prime modulus allows. Checking the certificate costs a handful of modular exponentiations (milliseconds, by fast squaring), *plus recursively certifying the prime factors $q$* — each much smaller, so the recursion collapses fast.

Concretely for our hero: $p - 1 = 2^{255} - 20$ factors as

$$p - 1 \;=\; 2^2 \cdot 3 \cdot 65147 \cdot Q,$$

where $Q$ is a 71-digit prime with its own (short) certificate, and 65147 recurses one more level: $65146 = 2 \cdot 32573$ with 32573 prime. The full certificate for $p$ is a small tree of witnesses and factorizations — a few hundred bytes of data standing behind a 77-digit claim:




(Check one leaf of this tree yourself right now, no computer: $4 \cdot 3 \cdot 65147 = 781764$, and $781764 \cdot Q$ must reproduce the 77-digit $p - 1$ — the *product identity* is the easiest condition of a certificate to audit, and auditing one leaf by hand is a good habit before trusting a tree. The witness conditions for $w = 2$ at the top node were re-verified computationally while writing this chapter; the kernel-checked version of this construction, for the Pallas modulus, lives in `pasta-pallas-verified`.)

### ∠ Pen and paper: checking a Pratt certificate by hand — all of it

Nothing builds trust in a certificate like verifying one completely, so here is the full check for $n = 97$, witness $w = 5$ — every modular multiplication on this page, using the same square-and-multiply ladder the real code uses (and which you will implement in the exercises). First the data: $n - 1 = 96 = 2^5 \cdot 3$, so there are three conditions: $5^{96} \equiv 1$, $5^{96/2} = 5^{48} \not\equiv 1$, and $5^{96/3} = 5^{32} \not\equiv 1 \pmod{97}$.

Build the powers of 5 by repeated squaring mod 97, reducing as you go — each line is one two-digit multiplication and one division with remainder, nothing more:

$$\begin{aligned}
5^2 &= 25 \\
5^4 &= 25^2 = 625 &&= 6 \cdot 97 + 43 \;\to\; 43 \\
5^8 &= 43^2 = 1849 &&= 19 \cdot 97 + 6 \;\to\; 6 \\
5^{16} &= 6^2 &&= 36 \\
5^{32} &= 36^2 = 1296 &&= 13 \cdot 97 + 35 \;\to\; 35
\end{aligned}$$

Now assemble the three exponents from these squares:

$$5^{48} = 5^{32} \cdot 5^{16} = 35 \cdot 36 = 1260 = 12 \cdot 97 + 96 \;\to\; 96 \equiv -1,$$

$$5^{96} = (5^{48})^2 \equiv (-1)^2 = 1. \qquad \checkmark$$

Check the conditions: $5^{96} \equiv 1$ ✓; $5^{48} \equiv 96 \neq 1$ ✓; $5^{32} \equiv 35 \neq 1$ ✓. All three hold — and by Pratt's theorem (whose reason the exercises make you articulate), 97 is prime, with the whole verification costing *seven* small multiplications. Trial division would have cost eight test divisions here — no savings at two digits. But the ladder's cost grows with the *number of bits* (one squaring per bit), while trial division grows with the *square root of the value* (one division per candidate) — linear versus exponential in the bit-length. At 77 digits that gap is the whole story, as the next box counts.

∠ **Pen and paper: costing the real certificate for $p = 2^{255} - 19$**

How much work is the full certificate check for the real prime, versus trial division? Count it, honestly, using the tree above.

*Certificate side.* One modular exponentiation with a 255-bit exponent costs at most 254 squarings plus at most 254 multiplies — call it $\leq 508$ modular multiplications, and abbreviate "modmul." The top node needs: $2^{p-1}$ (one exponentiation), and one $2^{(p-1)/q}$ for each of the four prime factors $q \in \{2, 3, 65147, Q\}$ — five exponentiations, $\leq 2540$ modmuls. The recursion adds: $Q$'s own node ($Q - 1$ has its own small factor list; generously, another five exponentiations at 71 digits, $\leq 2350$ modmuls), the 65147 node (16-bit numbers — three exponentiations of $\leq 32$ modmuls, noise), and 32573's (noise). Round the entire tree up to

$$\text{certificate check} \lesssim 5{,}000 \text{ modmuls.}$$

*Trial division side.* $\sqrt{p} \approx 2^{127.5}$, and candidate divisors (odd numbers, say) number about $2^{126.5} \approx 10^{38}$. *Ratio*:

$$\frac{10^{38} \text{ divisions}}{5 \times 10^3 \text{ modmuls}} \approx 10^{34}.$$

Thirty-four orders of magnitude — not an optimization, a different universe. And one more accounting worth doing: the certificate *data* is four factor entries and a handful of witnesses — a few hundred bytes. Someone (a computer algebra system, years of CPU time, once, offline) paid dearly to *find* the factorization of $p - 1$; every checker since pays five thousand multiplications. That asymmetry — expensive find, cheap check, tiny certificate — is the shape of every proof object the Lean kernel will ever hand you.

⋆ **Aha**

This find/check asymmetry is one of the great ideas of computer science — it is the P versus NP distinction wearing work clothes, and it is the engine of zero-knowledge proof systems (the very technology the Pasta curves serve). Proof assistants run on it too: Lean's whole architecture — clever tactics *finding*, dumb kernel *checking* — is the same asymmetry. A proof *is* a certificate.

## 7.3 The tempting shortcut, and why the house declines it

Lean offers a faster `decide`: the variant `native_decide` compiles the decision procedure to native machine code, runs it at full speed, and asserts the result. With a good primality test behind it, it can dispatch `Nat.Prime p` in seconds. Case closed?

Look at what you would be trusting. Ordinary `decide` produces a computation the *kernel* replays — the few-thousand-line paranoid core remains the only thing you trust. `native_decide` instead makes the theorem's truth depend on the Lean *compiler*, the C toolchain behind it, and the runtime — hundreds of thousands of lines promoted into your trusted base, in exchange for convenience on one theorem. Every proof downstream of the field — group law, scalars, signatures — would inherit that enlarged trust, visible forever in its axiom report (Chapter 11 shows you how to read those).

**△ Pitfall**

`native_decide` is not "cheating," and for exploratory work it is a fine tool. The trap is *silent trust inflation*: its use is invisible at the theorem statement — the cost appears only when someone audits the axioms, which is exactly what most readers never do. House rule, adopted from the projects this book accompanies: exploratory scaffolding may use it; **no shipped certificate depends on it**. The final Pallas-modulus primality proof in `pasta-pallas-verified` is a kernel-checked Lucas/Pratt certificate for precisely this reason.

## 7.4 Certificates in practice: Mathlib's toolbox

You will not hand-roll witness trees. Mathlib provides the machinery (`Nat.Prime` decision lemmas, `lucas_lehmer`-style infrastructure, and the `norm_num` extension `Nat.Prime` plugin) that constructs and checks Pratt-style certificates behind a single tactic call — while keeping every step kernel-checked. The shape in real code:

```
theorem p_prime : Nat.Prime (2^255 - 19) := by
  norm_num  -- certificate-backed primality, kernel-checked, ~seconds
```

When the built-in route struggles (very large or awkward moduli), the fallback is explicit: state the witness data as definitions, prove the two Pratt conditions with `norm_num`-driven modular exponentiation, and assemble. That is exactly the structure of the Pallas certificate in the companion repository — worth reading now with fresh eyes: `pasta-pallas-verified/verification/Proofs/Primality.lean`.

**>_ Try it yourself**

Open `exercises/Ch07.lean`. Ladder: certify 97, then 65537 (a Fermat prime beloved of RSA), then the ten-digit Mersenne prime $2^{31} - 1$, watching what each tool costs as the numbers grow. (Amusingly, the find/check asymmetry bites the *tactic* too: `norm_num` must *find* the witness tree before the kernel checks it, and at $2^{61} - 1$ the finding already takes minutes.) Finale: implement square-and-multiply modular exponentiation yourself and check the top witness condition for $p = 2^{255} - 19$ with `#eval` — your own hands on the certificate, at 77 digits, in milliseconds.

## Exercises

**Exercise 7.1.** Verify by hand that $w = 2$ is a Pratt witness for $p = 13$: compute $2^{12}$ mod 13 and $2^{12/q}$ mod 13 for each prime $q \mid 12$. Write the full certificate tree for 13, recursing into the factors of

12.

**Exercise 7.2.** Why does the witness condition force primality? Sketch the argument: if $w$ has order exactly $p-1$ in $\mathbb{Z}/p\mathbb{Z}$, then the multiplicative structure has $p-1$ elements, which fails if $p = ab$ with $1 < a, b < p$. (Full rigor optional; the shape is the point.)

**Exercise 7.3.** (Paper) Certify 65147 by hand-checkable data: given $65146 = 2 \cdot 32573$ and witness $w = 2$, list the exact conditions a checker must verify, then carry out the *cheapest* one: $2^{65146/32573} = 2^2 = 4 \not\equiv 1 \pmod{65147}$. For the remaining two conditions, count precisely how many squarings and multiplications the ladder needs (do not perform them). How many two-to-five-digit multiplications, total, stand between a skeptic and certainty about 65147?

**Exercise 7.4.** (Discussion) Bitcoin miners *find* block hashes; nodes *check* them. GPS receivers *check* satellite signals they could never *find*. Name two more systems built on the find/check asymmetry, and one system that would collapse without it.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 7.1.

*Pathway.* The data first: $12 = 2^2 \cdot 3$, so two conditions beyond $w^{12} \equiv 1$: exponents $12/2 = 6$ and $12/3 = 4$. Then power up 2 mod 13 by successive squaring, as the worked example did for 97 — at this size you can even go linearly.

*Answer.* Powers of 2 mod 13: $2^2 = 4$, $2^4 = 16 \equiv 3$, $2^6 = 2^4 \cdot 2^2 = 3 \cdot 4 = 12 \equiv -1$, and $2^{12} = (2^6)^2 \equiv (-1)^2 = 1$. Conditions: $2^{12} \equiv 1$ ✓; $2^6 \equiv 12 \neq 1$ ✓; $2^4 \equiv 3 \neq 1$ ✓. Witness confirmed. The full tree: node 13 (witness 2, $12 = 2^2 \cdot 3$) with children 2 (immediate) and 3 (witness 2: $2^2 = 4 \equiv 1 \pmod 3$, and $2^{2/2} = 2 \not\equiv 1$ — a two-line sub-certificate). Every claim in the tree is now something you have personally multiplied.

### Solution 7.2.

*Pathway.* The key concept is the *order* of $w$: the least $e > 0$ with $w^e \equiv 1$. The two witness conditions pin the order exactly; then count invertible elements two ways.

*Answer.* The condition $w^{p-1} \equiv 1$ says the order of $w$ divides $p-1$; the conditions $w^{(p-1)/q} \not\equiv 1$ for every prime $q \mid p-1$ rule out every *proper* divisor of $p-1$ (any proper divisor of $p-1$ divides some $(p-1)/q$). So the order is exactly $p-1$: the powers $w^1, w^2, \ldots, w^{p-1}$ are $p-1$ *distinct* elements, all invertible mod $p$ (each has $w^{\text{something}}$ as inverse). But if $p = ab$ with $1 < a, b < p$, the element $a$ is a zero divisor — $a \cdot b \equiv 0$ — so $a$ is not invertible, and neither are its multiples: strictly fewer than $p-1$ residues can be invertible. Contradiction; $p$ has no such factorization. (Full rigor pins down "distinct" and the divisor-covering claim — both one-liners with the order concept in hand. The shape is: *one loud element forces the whole multiplicative structure to be as big as only a prime allows.*)

### Solution 7.3.

*Pathway.* List conditions mechanically from the definition, then count ladder steps: an exponent of $b$ bits costs $\leq b-1$ squarings plus (at worst) $b-1$ multiplies; $65146 < 2^{16}$ and $32573 < 2^{15}$.

*Answer.* Conditions: (i) $2^{65146} \equiv 1 \pmod{65147}$; (ii) $2^{65146/2} = 2^{32573} \not\equiv 1$; (iii) $2^{65146/32573} = 2^2 = 4 \not\equiv 1$ ✓ (done — four is visibly not one); plus the recursive certificate for 32573. Counting: 65146 has 16 bits ($2^{16} = 65536$), so condition (i) costs $\leq 15$ squarings $+ \leq 15$ multiplies $= 30$; condition (ii), 15 bits, $\leq 28$; condition (iii) was free. Sub-certificate for 32573 ($32572 = 2^2 \cdot 17 \cdot 479$): four conditions on $\leq 15$-bit exponents, $\leq 4 \cdot 28 = 112$, plus leaves (17, 479 — another $\sim 60$ generously). Total: *under* 250 *small multiplications* — an afternoon with paper, an eyeblink for a kernel, and at the end 65147 is not "probably prime" but *prime*. (For the audit-minded: you were given 32572's factorization here the same way the checker is — as certificate data. Verifying $4 \cdot 17 \cdot 479 = 32572$ is one more hand multiplication: $68 \cdot 479 = 32572$ ✓.)

**Solution 7.4.**

*Pathway.* Look for systems where producing an artifact is costly but a short receipt convinces everyone — then for the collapse case, imagine checking costing as much as finding.

*Answer.* (Model answers.) Two more: *academic peer review of computer-assisted proofs* — the four-color theorem's checkers verify in hours what took years to construct; and *password hashing* — deriving a hash from a password is instant to check against, infeasible to invert. Others students propose: sudoku (solve vs. check), lottery tickets (draw vs. verify), digital signatures themselves (sign with secret effort-equivalent, verify publicly). A system that would collapse without the asymmetry: *blockchain consensus* — if verifying a block cost as much as mining it, every node would need a mine's electricity bill and the network could not exist. (So would mathematics as a social enterprise: if checking a proof cost as much as finding it, referees would be as rare as authors. In a sense, Lean is what happens when you drive the check cost toward zero and let anyone be a referee.)

**⋈ Checkpoint**

You should now be able to: explain why `decide` cannot prove $2^{255} - 19$ prime while a certificate can; reproduce the two Pratt witness conditions and check them on a small prime; articulate exactly what additional trust `native_decide` would introduce and why shipped certificates decline it; and recognize the find/check asymmetry as the common engine of certificates, proof assistants, and the P-vs-NP question.

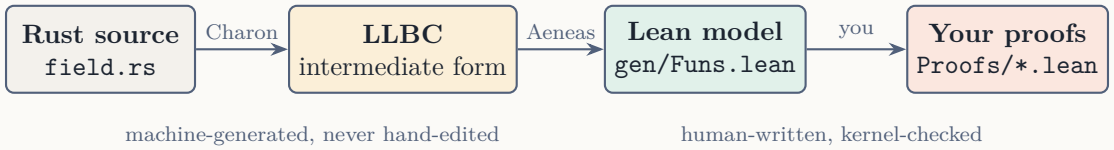
# Chapter 8

# From Rust to Lean: Verifying the Code That Actually Ships

## 8.1 The transcription problem

Everything so far proved facts about *Lean* programs. But the Ed25519 that guards your SSH connection is written in *Rust* (in our case, `curve25519-dalek` and its forks). An obvious plan: read the Rust, rewrite it in Lean by hand, verify the rewrite. The plan has a hole you could drive a key-recovery attack through: **what if you transcribe it wrong?** A hand-copy that silently fixes a bug — or introduces one — makes the proof a beautiful statement about code nobody runs.

The projects behind this book close the hole with a mechanical translation pipeline:




**Charon** compiles the Rust crate into LLBC ("low-level borrow calculus"), a simplified intermediate representation. **Aeneas** translates LLBC into pure Lean functions. The generated Lean — the *model* — lands in a `gen/` directory with a strict house rule: *never edit it.* Regenerate it from source, or don't touch it. Your proofs import the model and state theorems about it.

**⁎ The big idea**

The object of verification is the **extracted model**, produced from the shipping source by a deterministic tool — not a human transcription. The trust question shifts from "did we copy the code right?" (unauditable squinting) to "does the translator preserve meaning?" (one tool, studied once, shared by every project that uses it). You will hear this called *shrinking the trusted base*: swap many ad-hoc trusts for one well-examined trust.

## 8.2 What extracted code looks like

Here is real input and real output, lightly abridged. The Rust (from `curve25519-dalek`, radix-51 field addition):

```
impl Add for FieldElement51 {
    fn add(self, rhs: &FieldElement51) -> FieldElement51 {
        let mut output = *self;
        for i in 0..5 {
            output.0[i] += rhs.0[i];
        }
        output
    }
}
```

And the Lean model Aeneas produces for it (shape, not verbatim):

```
def fieldElement51_add (self rhs : Array U64 5) :
    Result (Array U64 5) := do
  let a0 <- Array.index_usize self 0
  let b0 <- Array.index_usize rhs 0
  let s0 <- a0 + b0            -- U64 addition: can FAIL on overflow
  let out <- Array.update self 0 s0
  ...                          -- and so on for limbs 1..4
```

Three features deserve your full attention, because every proof in the next two chapters engages them:

- **Machine integers are honest.** `U64` is not $\mathbb{N}$; it is 64-bit words. Aeneas models arithmetic on them precisely, overflow included.
- **Everything returns `Result`.** The `do`/`<-` notation threads a computation that can *fail* — returning an error value instead of a result — exactly where the Rust could panic or overflow. There is no pretending partial functions are total.
- **It is ugly.** Five limbs times load-add-store, all sequenced. Machine-generated code has no taste. The *proofs* restore the elegance; the model's job is fidelity.

## 8.3 Overflow is a proof obligation, not a footnote

In Rust, `a + b` on `u64` panics in debug mode and wraps in release mode when it overflows. In the extracted model, `a + b` returns a `Result` that is an error unless the mathematical sum fits. So the innocent theorem "add returns the right field element" *cannot even be stated* without first proving *add returns at all*:

```
theorem add_spec (a b : Array U64 5)
    (ha : LimbsBounded a) (hb : LimbsBounded b) :
    ∃ c, fieldElement51_add a b = .ok c ∧ LimbsBounded c ∧ ...
```

That hypothesis `LimbsBounded` — each limb below $2^{54}$, say — is the bounds invariant promised in Chapters 3 and 6: 51-bit payload plus headroom, so limb additions cannot reach $2^{64}$. The specification exposes what the Rust comments only whisper: this code is correct *under a discipline of bounded inputs*, the discipline must be maintained by every caller, and now there is a machine checking that it is.

**∠ Pen and paper: running the extracted model by hand, at the envelope's edge**

Trace the extracted `fieldElement51_add` on paper twice — once safely inside the operating envelope, once outside — and watch the `Result` machinery earn its keep. The model threads every step through `Result`: a step either produces `.ok v` and the `do`-block continues, or produces an error and everything after it is skipped (short-circuit).

*Run 1 — the worst legal input.* Take both arguments at the very edge of the bounds discipline: every limb of both inputs equal to $2^{54} - 1$ (the maximum the invariant `LimbsBounded` admits). Limb 0:

$$s_0 = (2^{54} - 1) + (2^{54} - 1) = 2^{55} - 2 \;<\; 2^{64}. \quad \texttt{.ok} \text{ — continue.}$$

Same for limbs 1–4; the block reaches its end and returns `.ok` of the array with every limb $2^{55} - 2$. Note the output *exceeds* $2^{54}$: addition legitimately leaves the input envelope, which is why the spec's conclusion states the *wider* bound $< 2^{55}$ — bounds flow through operations, and every theorem must say where they land. Nothing is failing here; the envelope is simply moving.

*Run 2 — one step past the cliff.* Now feed limbs of $2^{63}$ (representable in a `U64`, but far outside the invariant):

$$s_0 = 2^{63} + 2^{63} = 2^{64} \;\not<\; 2^{64} \quad \Longrightarrow \quad \text{the addition returns an error.}$$

The `do`-block short-circuits: limbs 1–4 never execute, and the whole function returns the failure — in Rust terms, this is the input on which the release build would have *silently wrapped* to 0 and kept going. The model turned undefined-ish behavior into a visible, provable event. Now reread the spec with both runs in mind: the hypotheses `LimbsBounded a`/`LimbsBounded b` are exactly the promise that run 2 cannot happen, and the conclusion `∃ c, ... = .ok c` is exactly the payoff. One theorem, and the wrap-around is not "probably absent" but *impossible for every input the discipline admits.*

**⋆ Aha**

Notice what just happened to "ugly generated code with Results everywhere": it forced us to discover, state, and prove the *implicit operating envelope* of the optimized implementation. The dalek authors knew this envelope; it lived in comments and code-review lore. Now it is a theorem. Extraction does not merely enable verification — it *interrogates* the code.

## 8.4 Practicalities: extraction as surgery

Running Charon on a whole real-world crate drags in everything the crate touches — iterators, byte serialization, trait machinery, SIMD backends — much of it irrelevant to the arithmetic core and some of it beyond what the translator supports. The working method, learned the honest way in the companion projects:

- **Extract functions, not crates.** Charon accepts specific roots (individual functions and impls); the extraction scripts in each companion repo (`extract.sh`, `extract-scalar.sh`) name exactly the arithmetic functions and get a small, clean model — 28 definitions instead of a thousand.
- **Some code will not translate.** The dalek scalar-multiplication backends use CPU-specific SIMD intrinsics no translator models. The boundary is then *documented*: those functions enter the trusted base, stated as assumptions, visible in every audit. Honest boundaries beat heroic fictions (Chapter 11 dwells on this).
- **Pin your tools.** The pipeline records exact versions of Charon, Aeneas, and Lean. A model regenerated with a different translator version is a *different model*; reproducibility of the proofs starts with reproducibility of the artifact under proof.

**∠ Pen and paper: sizing an extraction — the surgery, quantified**

"Extract functions, not crates" sounds like taste; it is arithmetic, and the numbers from the actual scalar-layer extraction of `dalek-ed25519-verified` make it vivid. First attempt: point Charon at the `Scalar` type wholesale. The dependency closure dragged in the byte-serialization path (`from_bytes`: shifts, masks, and a `wrapping_shr` on a signed type), the iterator machinery behind a zip-reverse-fold (`IterMut`, `Chunks`, three trait instantiations each), and the high-level wrapper's trait impls — roughly a *thousand* generated definitions, several using features at the translator's edge, every one of them something a proof might have to step around. Second attempt: name exactly the arithmetic roots — `Scalar52::add`, `sub`, `mul_internal`, `montgomery_reduce`, and their constants — and the generated model is *28 definitions*, every one arithmetic, every one provable. Do the auditor's division: 28/1000 — the surgery cut 97% of the material a reader would otherwise have to trust-or-verify. The lesson generalizes beyond this pipeline: **the size of the thing you verify is a design variable**, and an hour spent narrowing extraction roots buys weeks of not proving lemmas about iterator adapters. (The one-sentence version for your future code reviews: verification pressure flows backwards into interface design — arithmetic kernels with narrow waists get verified; grand unified objects do not.)

**△ Pitfall**

When an extraction fails or a model looks bizarre, the temptation is to "fix" the generated Lean by hand. Resist absolutely. A hand-edited model is a hand-transcription with extra steps — the exact hole this pipeline exists to close. The fixes live in extraction scope (choose different roots), in the source (rarely), or in documented assumptions (openly).

**>_ Try it yourself**

Open the companion repo `dalek-ed25519-verified`: read `extract.sh` (the roots), skim `verification/gen/Funs.lean` for `add`/`sub` (recognize the load-add-store pattern above), then read the first twenty lines of `verification/Proofs/FieldSpec.lean` and identify: the bounds invariant, the `Result` handling, and the statement of the theorem. You now recognize every structural element. The mathematics inside is Chapters 9 and 10.

## Exercises

**Exercise 8.1.** The Rust expression `output.0[i] += rhs.0[i]` hides four distinct failure/effect points that the Lean model makes explicit. Name them. (Hint: two indexings, one arithmetic operation, one write.)

**Exercise 8.2.** Suppose limbs are bounded by $2^{54}$. What is the largest value `a0 + b0` can take, and how many such additions can chain before a `U64` overflow becomes possible? Show the margin calculation — this is precisely why the invariant is $2^{54}$ and not $2^{63}$.

**Exercise 8.3.** (Design) Your colleague proposes verifying a hand-written Lean "reference implementation" instead of the extracted model, because it is prettier. List two failure modes this reintroduces, and one legitimate use a reference implementation still has (hint: Chapter 9 uses one as the *specification* side).

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 8.1.

*Pathway.* Expand the sugared Rust into its elementary operations, in evaluation order, and ask of each: can this one panic, wrap, or write?

*Answer.* In order: (1) *index read* `rhs.0[i]` — can panic if $i$ is out of bounds (here the loop guarantees $i < 5$, and the model makes even that guarantee a visible `Result` on `Array.index_usize`); (2) *index read* `output.0[i]` — same; (3) *the addition* — can overflow the `U64`: the failure point the worked example walked off; (4) *the write-back* into `output.0[i]` — an effect the pure model represents as `Array.update`, producing a *new* array value (functional update). Four operations, three failure modes, one effect — all hidden inside `+=` and all explicit in the extracted model, which is precisely why the model is provable and the sugar is not.

### Solution 8.2.

*Pathway.* Maximize under the hypothesis, then chain: after $k$ additions without reduction, limbs can approach $k$ times the single-input bound; find where that crosses $2^{64}$.

*Answer.* Largest single sum: $s_0 = 2 \cdot (2^{54} - 1) = 2^{55} - 2$. Chaining: adding $k$ inputs all bounded by $2^{54}$ yields limbs $< k \cdot 2^{54}$; the `U64` cliff sits where $k \cdot 2^{54} \geq 2^{64}$, i.e. $k = 2^{10} = 1024$ chained additions. The margin calculation behind "$2^{54}$, not $2^{63}$": the invariant must survive *multiplication*, whose column sums need $5 \cdot (\text{bound})^2 < 2^{128}$ (the Chapter 5 worked example: $5 \cdot 2^{108} < 2^{111}$, seventeen bits spare). A $2^{63}$ bound would give $5 \cdot 2^{126} > 2^{128}$ — overflow. So the number 54 is set by the *quadratic* consumer of the bound, not the linear one, and the addition headroom (1024 chained adds) is what falls out, not what was aimed for. Real invariants are negotiated between operations; the spec records the treaty.

### Solution 8.3.

*Pathway.* For failure modes, ask what the mechanical pipeline was bought to eliminate. For the legitimate use, ask what role *wants* to be clean and human-readable rather than faithful to machine details.

*Answer.* Two reintroduced failure modes: (1) *transcription drift* — the hand-model quietly fixes, or introduces, an off-by-one the Rust does not have, and the proof certifies the wrong artifact (the exact hole the pipeline closes); (2) *staleness* — the Rust moves on (a rebase, an optimization), nobody re-syncs the hand-model, and the certificate silently detaches from the shipping code; extraction fails loudly instead. One legitimate use: as the *specification* — the clean, obviously-correct definition (schoolbook arithmetic over $\mathbb{F}_p$, ideal group law) that the ugly extracted model is proven *equal to*. The reference implementation's prettiness is a liability in the role of "thing verified" and an asset in the role of "thing verified against" — one sentence worth keeping for every verification design review you ever attend.

**⋈ Checkpoint**

You should now be able to: draw the Rust → LLBC → Lean pipeline and say what each stage preserves; explain why generated models are never hand-edited; read a `Result`-typed extracted function and point to where overflow lives; and state why "the proof needs a bounds hypothesis" is a discovery about the *code*, not a weakness of the method.

# Chapter 9

# The Denotation Bridge: What Do Five Numbers *Mean*?

## 9.1 Two worlds, one bridge

We now hold two very different objects. On one side, mathematics: the field $\mathbb{F}_p$, where elements are abstract clock positions and $+$ means ideal modular addition. On the other side, the extracted model: arrays of five `U64` words, shuffled by loads, adds, and stores. The entire question of verified cryptography is how to say — precisely — that the second *implements* the first.

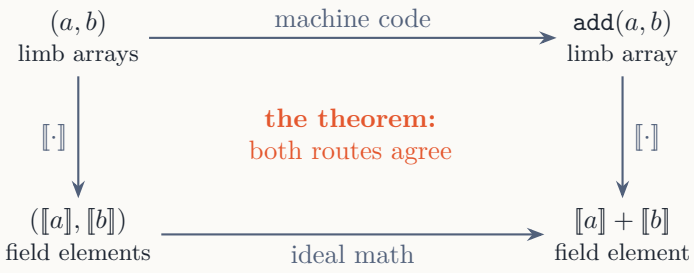
The answer is a single function, small enough to write on one line and important enough to carry this whole book. Given limbs $a = (a_0, a_1, a_2, a_3, a_4)$, define the **denotation**:

$$[\![a]\!] \;=\; a_0 + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4 \;\in\; \mathbb{F}_p.$$

Read $[\![a]\!]$ as "the field element these limbs *mean*." It is positional notation, nothing more — base $2^{51}$ instead of base 10, with the result interpreted on the clock face of $\mathbb{F}_p$. In Lean:

```
def denote (a : Array U64 5) : ZMod p :=
  a[0].val + 2^51 * a[1].val + 2^102 * a[2].val
           + 2^153 * a[3].val + 2^204 * a[4].val
```

With the bridge in hand, correctness of an operation becomes a *commuting square* — one picture you should internalize until you see it in your sleep:



**⁎ The big idea**

**The correctness of an implementation is the statement that denotation commutes with every operation:**

$$[\![\texttt{add}(a,b)]\!] = [\![a]\!] + [\![b]\!], \qquad [\![\texttt{mul}(a,b)]\!] = [\![a]\!] \cdot [\![b]\!], \qquad \dots$$

The left-hand side lives in the machine world (with its bounds hypotheses and `Results`); the right-hand side is pure mathematics. One equation per operation, and the ugly optimized code is pinned, forever, to the textbook meaning. Every verified-crypto project you will ever read is this diagram, instantiated.

## 9.2 Why redundancy is freedom (and where bugs hide)

A subtlety with consequences: denotation is **many-to-one**. The limb arrays $(19, 0, 0, 0, 0)$ and $(p + 19 \bmod 2^{\cdots}, \dots)$ — or more mundanely, unreduced sums whose limbs exceed $2^{51}$ — can denote the *same* field element. The representation has slack, and the implementation *exploits* it: the fast `add` from Chapter 8 just adds limbs pairwise, letting values drift above $2^{51}$, and nobody reduces until a cheaper moment. The commuting square still closes because $[\![\cdot]\!]$ doesn't care how bloated the limbs are — positional value is positional value.

**∠ Pen and paper: denoting the prime itself — a telescope in radix 51**

The canonical example of a collision uses the most important limb array in the codebase: the representation of $p$ itself,

$$P \;=\; (2^{51} - 19,\; 2^{51} - 1,\; 2^{51} - 1,\; 2^{51} - 1,\; 2^{51} - 1).$$

Claim: $[\![P]\!] = p \equiv 0$, so $P$ collides with $(0, 0, 0, 0, 0)$. Verify by hand — the sum telescopes beautifully. Write out the positional value:

$$[\![P]\!]_{\mathbb{Z}} = (2^{51} - 19) + (2^{51} - 1)2^{51} + (2^{51} - 1)2^{102} + (2^{51} - 1)2^{153} + (2^{51} - 1)2^{204}.$$

Expand each product: $(2^{51} - 1)2^{51k} = 2^{51(k+1)} - 2^{51k}$. So the sum is

$$(2^{51} - 19) + (2^{102} - 2^{51}) + (2^{153} - 2^{102}) + (2^{204} - 2^{153}) + (2^{255} - 2^{204}).$$

Every power except the last cancels against its neighbor — slide a pen along the chain and watch $2^{51}, 2^{102}, 2^{153}, 2^{204}$ annihilate — leaving

$$[\![P]\!]_{\mathbb{Z}} = 2^{255} - 19 = p \;\equiv\; 0 \pmod{p}. \qquad \checkmark$$

This computation is why the constant array $P$ (and its multiple $16P$, Chapter 2's worked example) appears throughout the dalek source: adding $P$ limb-wise changes nothing denotationally, and that "nothing" is exactly the freedom the subtraction trick spends. When the verified `sub` spec says `denote (sub a b) = denote a - denote b`, the proof's central move is this telescope, done once, as a lemma.

This is also exactly where the carry bugs of Chapter 1 live: code that is correct only while the drift

stays within headroom, and wrong on the rare inputs where it spills. In the verified development that danger becomes a visible, machine-checked pair of clauses attached to every operation:

```
theorem add_spec (ha : Bnd54 a) (hb : Bnd54 b) :
    ∃ c, add a b = .ok c
      ∧ Bnd55 c                          -- (1) bounds: the envelope holds
      ∧ denote c = denote a + denote b -- (2) value: the meaning is right
```

Clause (2) is the commuting square. Clause (1) feeds the *next* operation's hypothesis — correctness composes only because every theorem hands the following one the envelope it requires. A chain of such specs is the formal skeleton of "this sequence of optimized operations computes the formula we claim."

**⋆ Aha**

The denotation idea is vastly older and bigger than cryptography. Compilers prove "optimized code means the same as naive code"; databases prove "this query plan means the same query"; hardware verifies "this pipelined circuit means this instruction set." The pattern — map both sides into a mathematical meaning-space and prove the square commutes — is called *denotational semantics*, and you have now used it for real. It is the single most transferable idea in this book.

## 9.3 Multiplication: where the bridge earns its keep

Addition's square closes in an afternoon. Multiplication is the boss fight, and seeing *why* teaches you what verified arithmetic is really like. Schoolbook multiplication of two 5-limb numbers produces nine columns of partial products $\sum_{i+j=k} a_i b_j$; each column then owes a *carry* to the next; and columns $k \geq 5$ — weights $2^{255}$ and up — must be folded back using the Chapter 6 identity $2^{255} \equiv 19$. The implementation interleaves all three concerns for speed. The proof must un-interleave them:

$$[\![\texttt{mul}(a,b)]\!] \stackrel{?}{=} \left(\sum_{k=0}^{8} 2^{51k} \sum_{i+j=k} a_i b_j\right) \bmod p \stackrel{?}{=} [\![a]\!] \cdot [\![b]\!].$$

The right equality is algebra — `ring` territory. The left is a walk through the extracted code: every intermediate `U128` product bounded (no overflow — the $2^{54}$ headroom at work), every carry accounted, every $\times 19$ fold placed. In the companion projects this is a long `calc`-and-`have` museum: dozens of small steps, each dispatched by `omega` or a bound lemma, composed into one commuting square.

**∠ Pen and paper: the $\times 19$ fold, derived at the real weights**

The fold is where students usually first believe the whole enterprise is black magic; five lines of weight arithmetic dispel it, at full scale. Schoolbook multiplication of $a$ and $b$ (limbs $a_0..a_4$, $b_0..b_4$) produces columns $c_k = \sum_{i+j=k} a_i b_j$ at weights $2^{51k}$ for $k = 0, \ldots, 8$. Columns 5–8 carry weights $2^{255}, 2^{306}, 2^{357}, 2^{408}$ — all off the top of the representation. Reduce each weight with the Chapter 6 identity $2^{255} \equiv 19$:

$$2^{51(k+5)} \;=\; 2^{255} \cdot 2^{51k} \;\equiv\; 19 \cdot 2^{51k} \pmod{p}, \qquad k = 0, 1, 2, 3.$$

So column 5 re-enters at weight $2^0$ scaled by 19, column 6 at weight $2^{51}$ scaled by 19, and so

on — each overflow column lands exactly one radix position below where it left, times 19. The folded five-column result is therefore

$$\begin{aligned} r_0 &= c_0 + 19\,c_5 \\ r_1 &= c_1 + 19\,c_6 \\ r_2 &= c_2 + 19\,c_7 \\ r_3 &= c_3 + 19\,c_8 \\ r_4 &= c_4 \end{aligned}$$

— which, written with the products expanded ($r_0 = a_0b_0 + 19(a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1)$, etc.), is *character-for-character* the mysterious formula block in `curve25519-dalek`'s `mul`, the one decorated with the comment "see the comment above" that every reader of that file has squinted at. You have now derived it. Two bound checks make it safe, both already yours: each $19c_k < 2^5 \cdot 2^{111} = 2^{116}$ (the Chapter 5 column bound plus five bits for the 19), still comfortably inside a `U128`. And the formal proof of `mul`'s commuting square is precisely this box: the four weight identities as `have`s, the bound checks as `omega` goals, and `ring` to shuffle the expanded polynomial.

One more representational dialect, because you will meet it in the Pasta repos: **Montgomery form** stores $x$ as $x \cdot R \bmod p$ (with $R = 2^{256}$) because it makes reduction after multiplication cheap. The bridge absorbs the twist without complaint — define $[\![a]\!]_{\mathrm{M}} = (\text{positional value of } a) \cdot R^{-1}$ and the same commuting squares govern everything. Denotation is a *policy about meaning*, and it bends to fit the representation, not the other way around.

### ∠ Pen and paper: the cast — moving an equation from ℕ to the clock face

Every denotation proof ends with the same quiet move — "now reduce mod $p$" — and in a proof assistant that move must be performed, not gestured at. Here is the by-hand version of what `push_cast` and friends do, on the Interlude's own equation. You hold an *exact* identity of natural numbers:

$$\underbrace{v}_{\text{output value}} + 15\,c_2 = \underbrace{(a_0 + 4a_1)}_{[\![a]\!]\text{'s integer}} + \underbrace{(b_0 + 4b_1)}_{[\![b]\!]\text{'s integer}} \qquad \text{in } \mathbb{N}.$$

You want its shadow in $\mathbb{Z}/15\mathbb{Z}$. The bridge is the *cast homomorphism* $\iota : \mathbb{N} \to \mathbb{Z}/15\mathbb{Z}$ (send each number to its clock position), and the three facts that make it usable: (i) $\iota(x + y) = \iota(x) + \iota(y)$ and $\iota(x \cdot y) = \iota(x)\,\iota(y)$ — casting commutes with arithmetic, so the equation's *shape* survives; (ii) equal naturals cast to equal clock positions — so the equation's *truth* survives; (iii) $\iota(15) = 0$ — the modulus dies. Apply (ii), then distribute $\iota$ through both sides by (i), then kill the 15 by (iii):

$$\iota(v) + 0 \cdot \iota(c_2) = \iota(a_0) + 4\,\iota(a_1) + \iota(b_0) + 4\,\iota(b_1) \quad \Longrightarrow \quad [\![\text{out}]\!] = [\![a]\!] + [\![b]\!]. \ \blacksquare$$

That is all `push_cast` does: drive $\iota$ inward through $+$ and $\times$ by fact (i), normalizing the statement so facts (ii)/(iii) can fire. Knowing the hand version buys you two things. You can *predict* when the tactic will fail — fact (i) has no clause for subtraction or division in $\mathbb{N}$ (truncation! Chapter 2), so casts do not push through them, which is precisely why the proof pathway insists on an exact identity with the correction term *added on the left* rather than subtracted on the right. And you can *read* the real proofs' cast steps as what they are:

the one-line ceremony where a theorem about machine words becomes a theorem about field elements — the bridge being crossed, visibly, in both media.

**△ Pitfall**

When a denotation proof refuses to close, the failure is information — read it like a detective, in order: (1) Is the *bound* hypothesis strong enough for the intermediate products? (Count bits, on paper.) (2) Is the *denotation* right for this representation — radix, limb count, Montgomery factor? (3) Only then suspect the code. In the companion projects this checklist ran hundreds of times; its order reflects the actual base rates of what was wrong.

**>_ Try it yourself**

Open `exercises/Ch09.lean`. It builds a miniature of the whole story you can hold in your head: a *2-limb, radix-4* representation of $\mathbb{Z}/15\mathbb{Z}$ (limbs are values 0–3, denotation $a_0 + 4a_1$, and $16 \equiv 1$ makes the fold trivial). You will write `denote`, prove the commuting square for the provided `add` with carry, then for `mul` with its fold — every conceptual ingredient of the dalek proof, at a scale where `decide` can double-check your work.

## Exercises

**Exercise 9.1.** Compute by hand the denotation of the limb arrays $(19, 0, 0, 0, 0)$ and $(0, 0, 0, 0, 2^{51})$ in the radix-51 system, reducing mod $p = 2^{255} - 19$. Conclude that $[\![\cdot]\!]$ is not injective by exhibiting the collision.

**Exercise 9.2.** In the mini-system of the Try It box, find two distinct limb pairs denoting the same element of $\mathbb{Z}/15\mathbb{Z}$, and check that the provided `add` treats them interchangeably *as far as denotation goes* — compute both sides.

**Exercise 9.3.** Sketch the multiplication column sums $\sum_{i+j=k} a_i b_j$ for the 2-limb system and carry out the fold $16 \equiv 1$ by hand for $a = (3, 2)$, $b = (1, 3)$. Check against direct computation in $\mathbb{Z}/15\mathbb{Z}$.

**Exercise 9.4.** (Paper, challenge) For the radix-51 system with limbs bounded by $2^{54}$: bound one column $\sum_{i+j=4} a_i b_j$ of partial products and confirm it fits a `U128`. How much headroom remains? This number — not elegance — is why the invariant chose $2^{54}$.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

**Solution 9.1.**

*Pathway.* Compute both denotations as integers, then reduce mod $p$. One of them should ring a bell from the reduction identity.

*Answer.* $[\![(19,0,0,0,0)]\!]_{\mathbb{Z}} = 19$. For $(0,0,0,0,2^{51})$: the top limb sits at weight $2^{204}$, so the value is $2^{51} \cdot 2^{204} = 2^{255} \equiv 19 \pmod{p}$ — the crown identity of Chapter 6 again. Two visibly different arrays, one field element: $[\![\cdot]\!]$ is not injective, witnessed by the pair $((19,0,0,0,0), (0,0,0,0,2^{51}))$. Note that the second array even violates the "reduced" bound ($2^{51} \not< 2^{51}$) — it is exactly the kind of drifted-but-meaningful value lazy carries produce, and the denotation handles it without complaint.

### Solution 9.2.

*Pathway.* In the toy system, hunt for a nonzero array whose positional value is a multiple of 15. The largest representable value is $3 + 4 \cdot 3 = 15$ itself — convenient.

*Answer.* $[\![(3,3)]\!] = 3 + 4 \cdot 3 = 15 \equiv 0 = [\![(0,0)]\!]$. Interchangeability under `add` with, say, $(1,2)$: the machine runs differ — `add` $(3,3)$ $(1,2)$ produces carries, `add` $(0,0)$ $(1,2)$ does not — but both outputs denote $0 + (1 + 4 \cdot 2) = 9$. (With the definitions in `exercises/Ch09.lean`: `add` $(3,3)$ $(1,2)$: $s_0 = 4$, $s_1 = 3+2+1 = 6$, output $(0+1, 2) = (1,2)$, denotation 9 ✓; `add` $(0,0)$ $(1,2) = (1,2)$, denotation 9 ✓.) The spec never promised equal *arrays* — only equal *meanings*; representation freedom is preserved by every operation, which is the entire content of "the square commutes."

### Solution 9.3.

*Pathway.* Write the three columns of the $2 \times 2$ schoolbook product, fold the top one with $16 \equiv 1$, then cross-check against a direct computation in $\mathbb{Z}/15\mathbb{Z}$ — at toy scale you can afford both routes, which is the point of having a toy.

*Answer.* For $a = (3,2)$, $b = (1,3)$: columns $c_0 = a_0 b_0 = 3$, $c_1 = a_0 b_1 + a_1 b_0 = 9 + 2 = 11$, $c_2 = a_1 b_1 = 6$ at weights $1, 4, 16$. Fold: $16 \equiv 1$, so $c_2$ re-enters at weight 1:

$$\text{value} \equiv (c_0 + c_2) + 4c_1 = 9 + 44 = 53 \equiv 53 - 45 = 8 \pmod{15}.$$

Direct route: $[\![a]\!] = 3 + 8 = 11$, $[\![b]\!] = 1 + 12 = 13$; $11 \cdot 13 = 143 = 9 \cdot 15 + 8 \equiv 8$ ✓. Both routes, one answer — you have closed a commuting square numerically. The Lean theorem `mulVal_spec` is this computation with the numbers replaced by universally quantified variables: same fold, all inputs at once.

### Solution 9.4.

*Pathway.* This is the Chapter 5 worked example — reconstruct it from memory before checking back.

*Answer.* Each product $a_i b_j < 2^{54} \cdot 2^{54} = 2^{108}$; column $c_4$ has five of them, so $c_4 < 5 \cdot 2^{108} < 2^{111} < 2^{128}$, leaving 17 bits of headroom — consumed, in the real code, by the carry-in from the previous column and the $\times 19$ fold factor ($19 < 2^5$: the fold costs five of the seventeen bits, which is why the margin looks generous and is not). Run the same audit at bound $2^{55}$: $5 \cdot 2^{110} < 2^{113}$, fold to $2^{118}$ — still fits, but the *addition* chain budget halves; at $2^{63}$: $5 \cdot 2^{126} > 2^{128}$ — broken outright. The invariant $2^{54}$ is where the operations' competing demands settle, and now you have audited the settlement yourself.

**⋈ Checkpoint**

You should now be able to: write the radix-51 denotation from memory; draw the commuting square and label which side owns bounds and `Results`; explain why many-to-one representation is both the performance trick and the bug habitat; and recognize the two-clause shape (bounds propagation + value equation) as the universal skeleton of implementation-correctness theorems.

# Interlude: A Complete Verification, Entirely by Hand

Before the full field campaign of the next chapter, we pause to do something no other book chapter will ask of you again: verify an arithmetic implementation *completely*, on paper, with nothing left to the machine — and then watch the same proof re-enacted in Lean, line by line. The system is the two-limb miniature from Chapter 9's exercises; the point is that after this interlude, nothing in the real proofs is a new *kind* of thing — only a bigger instance of what your own pen has already done.

## I.1 The system, restated in full

- **Representation.** A value is a pair of natural-number limbs $(a_0, a_1)$, intended as base-4 digits.
- **Bounds invariant.** $\mathrm{Bnd}(a) \iff a_0 < 4 \wedge a_1 < 4$.
- **Denotation.** $[\![a]\!] = a_0 + 4a_1 \in \mathbb{Z}/15\mathbb{Z}$. The modulus 15 is chosen so that the radix squared folds: $16 \equiv 1 \pmod{15}$ — the toy twin of $2^{255} \equiv 19$.
- **The implementation under verification.**

  $$\texttt{add}(a, b) := \big(\, s_0 \bmod 4 + \lfloor s_1/4 \rfloor,\ \ s_1 \bmod 4 \,\big), \quad \text{where } s_0 = a_0 + b_0,\ s_1 = a_1 + b_1 + \lfloor s_0/4 \rfloor.$$

  In words: add low limbs; carry into the high sum; the high sum's own overflow — weight $16 \equiv 1$ — folds back into the low limb.

## I.2 The specification

The two-clause shape of Chapter 9, in full:

$$\textbf{(Spec)} \qquad \mathrm{Bnd}(a) \to \mathrm{Bnd}(b) \to \underbrace{\big(\texttt{add}(a,b)_0 < 5 \wedge \texttt{add}(a,b)_1 < 4\big)}_{\text{bounds clause}} \wedge \underbrace{[\![\texttt{add}(a,b)]\!] = [\![a]\!] + [\![b]\!]}_{\text{value clause}}.$$

Pause on the asymmetric output bound: the low limb is only promised $< 5$, not $< 4$ — the fold can push it one past a clean digit. An honest spec records where the envelope actually lands, and the next operation's hypotheses must accept what this one delivers. (In the real field code this is the $2^{51}$-in, $2^{52}$-out pattern of Chapter 8's worked example.)

Run the Chapter 11 substitution test before proving anything: would "return $(0, 0)$ always" pass? Bounds clause: yes. Value clause: no — $a = (1, 0), b = (0, 0)$ gives $1 \neq 0$. The spec has teeth. Now we earn it.

## I.3 The bounds clause, proved by hand

Assume $\mathrm{Bnd}(a)$, $\mathrm{Bnd}(b)$; so $a_0, a_1, b_0, b_1 \le 3$. Chase every intermediate through its interval:

$$\begin{array}{lcll}
s_0 & = & a_0 + b_0 & \le \; 3+3 = 6 \\
\lfloor s_0/4 \rfloor & \le & \lfloor 6/4 \rfloor & = \; 1 \\
s_1 & = & a_1 + b_1 + \lfloor s_0/4 \rfloor & \le \; 3+3+1 = 7 \\
\lfloor s_1/4 \rfloor & \le & \lfloor 7/4 \rfloor & = \; 1 \\
\texttt{add}(a,b)_0 & = & s_0 \bmod 4 + \lfloor s_1/4 \rfloor & \le \; 3+1 = 4 \;<\; 5 \quad \checkmark \\
\texttt{add}(a,b)_1 & = & s_1 \bmod 4 & \le \; 3 \;<\; 4 \quad \checkmark
\end{array}$$

Done — and notice the proof *is* the headroom audit of Chapter 1, in miniature: every line asks "how big can this intermediate get," and the two ✓ lines are the treaty the spec promised. Notice also that the bound $< 5$ is *tight*: at $a = b = (2,3)$, $s_0 = 4$ folds a carry and $s_1 = 7$ folds another, giving low limb $0 + 1 = 1\ldots$ try $a = b = (3,3)$: $s_0 = 6$, $s_1 = 7$, output $(2+1, 3) = (3,3)$ — hmm, low limb 3. Exercise I.1 asks you to find the input that actually achieves 4; it exists, and finding it will teach you more about tightness than ten theorems.

## I.4 The value clause, proved by hand

The pathway (identical to the one you will use in Lean): first prove an *exact integer identity* — no mod, no hand-waving — then let the clock face absorb the multiple of 15.

**Step 1: the division algorithm, twice.** For any naturals, $x = 4\lfloor x/4 \rfloor + (x \bmod 4)$. Apply to $s_0$ and $s_1$:

$$s_0 = 4\lfloor s_0/4 \rfloor + (s_0 \bmod 4), \qquad s_1 = 4\lfloor s_1/4 \rfloor + (s_1 \bmod 4).$$

**Step 2: compute the output's integer value, plus a correction.** Write $c_1 := \lfloor s_0/4 \rfloor$ (low carry) and $c_2 := \lfloor s_1/4 \rfloor$ (the folded top carry). The output's positional value is

$$\begin{array}{rcll}
[\![\texttt{add}(a,b)]\!]_{\mathbb{Z}} & = & (s_0 \bmod 4 + c_2) + 4\,(s_1 \bmod 4) & \\
& = & (s_0 - 4c_1 + c_2) + 4\,(s_1 - 4c_2) & \text{(Step 1, both equations)} \\
& = & s_0 - 4c_1 + 4s_1 - 15\,c_2 & \\
& = & s_0 - 4c_1 + 4\,(a_1 + b_1 + c_1) - 15\,c_2 & \text{(definition of } s_1\text{)} \\
& = & (a_0 + b_0) + 4\,(a_1 + b_1) \;-\; 15\,c_2. &
\end{array}$$

Watch what happened in the last two lines: the low carry $c_1$ *cancelled exactly* — $-4c_1$ against $+4c_1$ — because a carry is value moved between positions, not value created; and the top carry survived only as $-15\,c_2$, because folding weight 16 down to weight 1 loses exactly 15 per unit. The bookkeeping *is* the arithmetic meaning of "carry" and "fold," laid bare.

**Step 3: pass to the clock face.** Reducing mod 15, the term $15\,c_2$ vanishes:

$$[\![\texttt{add}(a,b)]\!] \;=\; (a_0 + 4a_1) + (b_0 + 4b_1) \;=\; [\![a]\!] + [\![b]\!] \quad \text{in } \mathbb{Z}/15\mathbb{Z}. \qquad \blacksquare$$

The value clause is proved — for *every* bounded input, by three steps of eighth-grade algebra. You have now verified an arithmetic implementation by hand, completely: no case was sampled, no input untested, because no input was *tested* at all. The quantifier came from algebra, not enumeration.

## I.4b Multiplication's fold, by the same method

One more clause, to see the method survive a second operation. The toy multiplication (as in `exercises/Ch09.lean`) computes the three schoolbook columns and folds the top one:

$$\texttt{mulVal}(a,b) := \underbrace{a_0b_0}_{c_0} + \underbrace{a_1b_1}_{c_2\ \text{(folded)}} + 4\,\underbrace{(a_0b_1 + a_1b_0)}_{c_1},$$

claim: $[\![\texttt{mulVal}(a,b)]\!] = [\![a]\!]\cdot[\![b]\!]$ — this time with *no bounds hypotheses at all*. Same three-step pathway, but Step 2's identity is now pure algebra. Expand the right-hand side:

$$(a_0 + 4a_1)(b_0 + 4b_1) = a_0b_0 + 4\,(a_0b_1 + a_1b_0) + 16\,a_1b_1.$$

Compare with `mulVal`: the only mismatch is the top column's weight — $16\,a_1b_1$ there, $1\cdot a_1b_1$ here. So the exact integer identity is

$$\texttt{mulVal}(a,b) + 15\,a_1b_1 \;=\; (a_0 + 4a_1)(b_0 + 4b_1),$$

— verify it by cancelling the displayed expansions — and Step 3 kills the $15\,a_1b_1$ on the clock face. ■

Two observations before we translate. First, *why* no bounds were needed: this theorem speaks only of values, and positional value is indifferent to digit discipline; bounds enter only when machine words must contain the intermediates (the toy `mulVal` returns a bare natural number — the real `mul` returns limbs, and *re-limbing is where its bounds clauses live*). Second, the correction term's shape: 15 per unit of *folded column*, exactly the "cost of a fold" law from I.4 — one law, two operations, and in the real system the same law reads "$p$ per folded column," four times over (Chapter 9's fold worked example).

## I.5 The same proof, in Lean, line by line

Here is the machine version (it is `solutions/Ch09.lean`, compiled and axiom-audited), interleaved with the paper steps it enacts:

| **Paper (sections I.3–I.4)** | **Lean** |
|---|---|
| unpack the four digit bounds | `obtain ⟨ha1, ha2⟩ := ha ...` |
| the interval chase of I.3 | `simp only [add]; omega` |
| Steps 1–2: the exact integer identity, carries cancelling — stated with the $15c_2$ correction on the left | `have key : (add a b).1 + 4*(add a b).2 + 15*(...) = ... := by simp only [add]; omega` |
| Step 3: cast to $\mathbb{Z}/15\mathbb{Z}$; the 15 becomes 0 | `push_cast at this; rw [show (15 : ZMod 15) = 0 by decide]` |
| "$\ldots = [\![a]\!] + [\![b]\!]$. ■" | `simp only [denote]; linear_combination this` |

Read the correspondence twice, because it is the book's whole method in one table. The interval chase you did line-by-line is one `omega` call — the machine *decides* what you *derived*, which is why Chapter 5 called these procedures contracts. The delicate part on paper — Step 2's cancellation — is

delicate in Lean too: it is the `key` identity, and getting its statement right (which correction term? on which side?) is where the human does the thinking in both media. Step 3 is bookkeeping in both media. The division of labor between you and the machine is the same division between insight and verification your paper proof already had.

And now the punchline of the whole interlude: the field-layer proof of `dalek-ed25519-verified` is *this proof* with five limbs instead of two, radix $2^{51}$ instead of 4, fold constant 19 instead of 1 (times the fold identity's bookkeeping), and five carry cancellations instead of one. Every step type — interval chase, exact identity with correction term, cast and kill the modulus — you have now executed by hand.

## Interlude exercises

**Exercise I.1.** Find bounded inputs achieving the tight output bound: $\texttt{add}(a,b)_0 = 4$. (Systematic pathway: you need $s_0 \bmod 4 = 3$ *and* a fold, $c_2 = 1$; work out what $s_0$ and $s_1$ must be, then pick digits realizing them.)

**Exercise I.2.** Re-run the *entire* I.3–I.4 proof for the variant system with radix 8 and modulus 63 (so $64 \equiv 1$): two limbs $< 8$, denotation $a_0 + 8a_1$. Every step should transpose mechanically — when one doesn't, you have found a place where you were pattern-matching instead of understanding; welcome, that is the exercise working.

**Exercise I.3.** (Bridge to the real thing) In the I.4 cancellation, the low carry vanished and the top carry cost 15 per unit. For the real radix-51 system: state what the analogous "cost per unit of top carry" is, and verify your answer against the fold identity of Chapter 9's worked example.

**Exercise I.4.** (The missing clause) Section I.4b's `mulVal` returns a bare number, not limbs — so its theorem has a value clause but no bounds clause. Complete the design on paper: define $\texttt{relimb}(v) := (v \bmod 4 + (\text{fold}), \dots)$ — that is, re-express a value $v \le 90$ (the largest `mulVal` of bounded inputs — verify this bound first!) as two limbs using the division algorithm and one fold. How many fold rounds does $v \le 90$ need before both limbs are $< 4$? State the two-clause spec your `relimb` would carry.

## Interlude solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution I.1.

*Pathway.* Work backwards from the two requirements. $c_2 = 1$ needs $s_1 \ge 4$; $s_0 \bmod 4 = 3$ with a useful range of $s_0$: since $s_0 \le 6$, "$s_0 \bmod 4 = 3$" means $s_0 = 3$ (no carry out) — so the fold must come entirely from $a_1 + b_1$.

*Answer.* Take $s_0 = 3$ (say $a_0 = 3, b_0 = 0$), no low carry; then $s_1 = a_1 + b_1 \ge 4$ (say $a_1 = b_1 = 2$, $s_1 = 4$): output low limb $= 3 + 1 = 4$. Concretely $a = (3,2)$, $b = (0,2)$: $\texttt{add} = (4,0)$. Check the value clause holds anyway: $[\![(4,0)]\!] = 4$ and $[\![a]\!] + [\![b]\!] = 11 + 8 = 19 \equiv 4$ ✓ — the bound is tight but the meaning is intact, which is exactly why the spec's bounds clause said $< 5$ and not $< 4$. (If you tried

to strengthen the spec to $< 4$, this input is the counterexample the failed proof would point at — Chapter 12's graduation exercise mechanism, met early.)

**Solution I.2.**

*Pathway.* Transcribe I.3–I.4 replacing $(4, 15, 16)$ by $(8, 63, 64)$ and the digit bound 3 by 7.

*Answer.* Bounds chase: $s_0 \le 14$, $c_1 \le 1$, $s_1 \le 15$, $c_2 \le 1$, output low $\le 7 + 1 = 8 < 9$, output high $\le 7 < 8$. Value identity: output value $= s_0 - 8c_1 + 8s_1 - 63\,c_2 = (a_0 + b_0) + 8(a_1 + b_1) - 63\,c_2$ — low carry cancels $(-8c_1 + 8c_1)$, top carry costs $64 - 1 = 63$ per unit, which vanishes mod 63. Every step transposed; the only places requiring thought were the two spots where the old numbers were *related* ($15 = 16 - 1$ became $63 = 64 - 1$; $\lfloor 14/8 \rfloor = 1$ needed re-checking, not copying) — and those two spots are precisely the design constraints of the system: fold constant = radix$^2$−modulus relation, and single-bit carries. Now you know which numbers in such a system are load-bearing.

**Solution I.3.**

*Pathway.* In I.4 the cost was (weight folded from) − (weight folded to) $= 16 - 1 = 15$ per unit of top carry. Transpose the weights.

*Answer.* The real system folds weight $2^{255}$ down to weight $2^0$ *with a factor of 19*: one unit of top overflow re-enters as $+19$ instead of $+2^{255}$, so the correction is $2^{255} - 19 = p$ per unit — the cost is exactly one prime $p$, which is why it vanishes mod $p$, and why the fold constant *must* be 19 and nothing else: it is the unique value making the correction a multiple of the modulus. Cross-check with the fold worked example: $2^{51(k+5)} \equiv 19 \cdot 2^{51k}$ says each folded column's correction is $2^{51k}(2^{255} - 19) = 2^{51k}\,p$ ✓. The toy's "15 per carry" and the field's "$p$ per carry" are the same sentence at different type sizes — carry the sentence with you into the next chapter.

**Solution I.4.**

*Pathway.* Bound `mulVal` first by maximizing each column, then design `relimb` as "split by the division algorithm, fold the overflow, repeat until in range," tracking the bound through each round exactly as I.3's chase did.

*Answer.* Bound: columns give $\le 9 + 9 + 4 \cdot 18 = 90$ ✓. One `relimb` round on $v$: limbs $(v \bmod 4 + \lfloor v/16 \rfloor,\ \lfloor v/4 \rfloor \bmod 4)$ — value preserved mod 15 since the fold costs 15 per unit (I.4's law). Chase the bound: round one sends $v \le 90$ to $v' \le 3 + 5 + 12 = 20$; a value $< 16$ needs no fold and splits into two clean digits, so only $v' \in \{16, \dots, 20\}$ needs a further round, which lands $\le 3 + 1 + 4 = 8 < 16$ — **at most three rounds**, usually two. The spec your `relimb` carries is precisely the two-clause shape: (bounds) both output limbs $< 4$; (value) $[\![\texttt{relimb}(v)]\!] = v$ in $\mathbb{Z}/15\mathbb{Z}$. And now compose: `relimb`∘`mulVal` has a full multiplication spec, assembled from two lemmas — which is, in miniature, exactly how the real `mul` proof is architected (fold theorem + carry-chain re-limbing), and why Chapter 10's campaign put `reduce` before `mul`.

**⋈ Checkpoint**

You have now: proved both clauses of a two-clause spec with your own pen; watched each paper step map onto one Lean tactic; transposed the proof to a sibling system and located its load-bearing constants; and priced a fold in multiples of the modulus. The next chapter's campaign is this interlude at production scale — when it feels big, return here and find the step type you are on.

# Chapter 10

# Verifying a Field: The Full Campaign

## 10.1 The summit statement

Every thread so far — specs as types, tactics, automation, $\mathbb{F}_p$, primality, extraction, denotation — was preparation for one theorem. In the companion repositories it is called the *field implementation certificate*, and (lightly paraphrased) it says:

```
theorem fieldImplementation :
    -- p is prime, so ZMod p is genuinely a field
    Nat.Prime p
    -- and for all bounded limb arrays, every operation's
    -- commuting square closes:
  ∧ (∀ a b, Bnd a → Bnd b → AddSquare a b)
  ∧ (∀ a b, Bnd a → Bnd b → SubSquare a b)
  ∧ (∀ a b, Bnd a → Bnd b → MulSquare a b)
  ∧ (∀ a,   Bnd a →         SquareSquare a)
  ∧ (∀ a,   Bnd a →         InvertSquare a)   -- Fermat chain: a^(p-2)
  ∧ ...                                       -- reduce, negate, encode
```

One theorem, kernel-checked, quantified over *every* input the representation admits: the extracted dalek field arithmetic implements $\mathbb{F}_p$. This chapter is the story of the campaign that proves it — told honestly, including the two places where the terrain fought back, because the failures teach more than the victories.

## 10.2 Order of battle

You do not prove such a conjunction by heroism; you prove it by sequencing. The campaign order in the real projects, and the reason for each position:

1. **Bounds lemmas first** — pure `omega` facts about limb sizes, no denotation at all. Cheap, and everything depends on them.
2. **Primality** (Chapter 7) — independent of the code entirely; it dignifies `ZMod p` into a field.

3. **add, sub, negate** — linear operations; the commuting squares close with `omega` plus the denotation unfolds. Confidence builders.

4. **reduce** — the $\times 19$ fold in isolation. Proving it alone, before mul uses it, halves the hardest proof's size.

5. **mul, square** — the boss fight of Chapter 9: column sums, carries, folds. Squaring is mul with algebraic shortcuts — a separate code path in dalek, hence a separate theorem. No shortcuts in the proof: the *code's* shortcut is precisely what needs checking.

6. **invert** — the 254-squaring Fermat chain, verified as a `calc` of exponent bookkeeping on top of `mul`/`square` specs, ending at $a^{p-2}$; Fermat's little theorem (Mathlib's) closes the square.

Notice the shape: *each layer consumes only the specs of the layer below*, never reaching into implementations. By step 6 you are doing exponent arithmetic, blissfully ignorant of carries. That is the compositionality that Chapter 9's two-clause specs (bounds + value) were designed to buy.

**∠ Pen and paper: why $16p$ — auditing a design constant to the bit**

Step 3's subtraction hides the field layer's most quotable design decision, and you can audit it completely. The problem: compute $a - b$ limb-wise in unsigned words, where limbs of $b$ may be as large as $2^{54} - 1$ (the invariant's edge) while limbs of $a$ may be as small as 0. Unsigned subtraction would truncate (Chapter 2); the fix is to compute $a + (kp - b)$ for a suitable multiple $k$ of the prime — denotationally free, since $kp \equiv 0$ — chosen so that every limb of $kp - b$ stays positive. How large must $k$ be? The limb constants of $kp$ scale the telescoped representation from Chapter 9:

$$kP = (k(2^{51} - 19),\ k(2^{51} - 1),\ k(2^{51} - 1),\ k(2^{51} - 1),\ k(2^{51} - 1)),$$

whose *smallest* limb is $k(2^{51} - 19)$. Positivity of every limb of $kp - b$ demands

$$k\,(2^{51} - 19)\ \geq\ 2^{54} - 1.$$

Try the candidates, because powers of two are what the code wants:

$$k = 8: \quad 8(2^{51} - 19) = 2^{54} - 152\ <\ 2^{54} - 1 \qquad \times \text{ fails, by } 151;$$

$$k = 16: \quad 16(2^{51} - 19) = 2^{55} - 304\ \geq\ 2^{54} - 1 \qquad \checkmark \text{ with } 2^{54} - 303 \text{ to spare.}$$

So 16 is the *least* power of two that works — 8 misses by a margin of 151, invisible to any test that doesn't drive a limb of $b$ within 151 of the very top of its envelope. Two more checks complete the audit: the result's limbs are bounded by $2^{54} + 2^{55} < 2^{56} < 2^{64}$ (no overflow on the additive side ✓), and the denotation shifts by exactly $16p \equiv 0$ ✓. In the shipped proof this box is one lemma with three `omega` obligations; in the git history of more than one real crypto library, the analogous constant is a patched CVE. Now you know how to tell which side of that line a codebase is on.

**∠ Pen and paper: verifying the inversion chain's bookkeeping — all 265 steps**

Step 6 sounds heroic — verify a hand-crafted chain of 254 squarings and 11 multiplications computes $a^{p-2}$ — until you see that the whole verification is *exponent arithmetic*, and the

exponents obey one identity you can check at every scale. The chain builds values $a^e$ for a cascade of exponents $e$; squaring doubles $e$, multiplying adds them. The dalek chain's opening moves:

$$\begin{aligned} z_2 &= a^2 && e = 2 \\ z_9 &= (z_2)^{2^2} \cdot a && e = 2 \cdot 4 + 1 = 9 \\ z_{11} &= z_9 \cdot z_2 && e = 9 + 2 = 11 \\ z_{2^5-1} &= (z_{11})^2 \cdot z_9 && e = 22 + 9 = 31 = 2^5 - 1. \end{aligned}$$

From $31 = 2^5 - 1$ the chain climbs by a doubling ladder whose single identity you should verify once symbolically:

$$(2^k - 1) \cdot 2^k + (2^k - 1) \;=\; 2^{2k} - 1 \qquad \text{("shift a block of } k \text{ ones up by } k \text{ and fill").}$$

Check it at $k = 5$: $31 \cdot 32 + 31 = 992 + 31 = 1023 = 2^{10} - 1$ ✓. The ladder then assembles mixed blocks the same way: $2^{10} - 1 \to 2^{20} - 1 \to 2^{40} - 1$, then $(2^{40} - 1)2^{10} + (2^{10} - 1) = 2^{50} - 1$, then $2^{100} - 1 \to 2^{200} - 1$, then $(2^{200} - 1)2^{50} + (2^{50} - 1) = 2^{250} - 1$. Final move: shift five and add the stashed $z_{11}$:

$$(2^{250} - 1) \cdot 2^5 + 11 \;=\; 2^{255} - 32 + 11 \;=\; 2^{255} - 21 \;=\; p - 2. \qquad \checkmark$$

Every line above is hand-checkable, and together they *are* the correctness of the inversion chain — modulo only "squaring squares and multiplication adds exponents," which is the already-proven `mul` and `square` specs feeding Fermat (Chapter 6). Count the cost while you are here. Squarings, stage by stage: 1 (to $z_2$) $+2$ (to $z_9$) $+1$ (to $z_{31}$) $+5 + 10 + 20$ (to $2^{10}, 2^{20}, 2^{40}$) $+10$ (to $2^{50}$) $+50 + 100$ (to $2^{100}, 2^{200}$) $+50$ (to $2^{250}$) $+5$ (final shift) $= 254$. Multiplications: one per block-join — count the "·"s above — exactly 11. The advertised $254 + 11$ is not folklore; you just audited it. That is the point of this box: "verify 265 field operations" collapsed into ten lines of exponent algebra a patient undergraduate audits over coffee.

## 10.3 Dispatches from the terrain

**The wall that was really there.** Partway up, one proof style hit a genuine limit of the tool: correctness certificates for `mul`-scale goals, when handed to a general decision procedure in one monolithic call, generate internal certificates with coefficients on the order of $2^{256}$ — and checking them can exhaust the proof checker's memory. One such call, during the development of the Pasta field proofs, consumed twelve gigabytes and crashed the machine (Chapter 5 told you this story from the tactic side). The cure was never cleverness — it was *decomposition*: isolate each carry step as its own small lemma with a tiny context, prove the value identity with `linear_combination` (a tactic that checks a *stated* linear certificate rather than searching for one), and let the big theorem be an assembly of small checked parts. The same discipline, plus hard memory caps on the checker process, became house infrastructure.

**∠ Pen and paper: the wall, quantified — why the monolithic proof dies**

The memory catastrophe is not mystical; you can estimate it on one page, and the estimate teaches the cure. When a decision procedure like `omega` proves a linear goal, it does not just say "true" — it emits a *certificate*: a linear combination of the hypotheses with explicit integer

coefficients that the kernel then re-checks (Chapter 7's find/check split, again). Now count what a *monolithic* field-multiplication goal hands it. The goal relates quantities at wildly different scales: raw limbs ($\sim 2^{51}$), column sums ($\sim 2^{111}$), folded values ($\sim 2^{116}$), and the final positional stack, whose top weight is $2^{204}$ — so eliminating variables across the whole system produces certificate coefficients up to the *products* of these scales: numbers of order $2^{256}$ to $2^{512}$, i.e. integers of 80–150 decimal digits. Per coefficient that is only tens of bytes — but the elimination is quadratic-ish in the hypothesis count: with $\sim 60$ hypotheses (five limbs each for two inputs, nine columns, nine carries, plus bounds for everything), the certificate and the kernel's intermediate terms multiply into *millions* of big-number nodes, each participating in further products. Twelve gigabytes is not even surprising once you draw the multiplication table of scales — and this estimate, run *before* the tactic, is how the wall gets predicted rather than hit.

The cure now reads directly off the arithmetic: cut the elimination range. Prove each carry step as its *own* lemma with only its own few hypotheses in scope (coefficients stay near the step's own scale); where a large linear identity is genuinely needed, state it yourself and let `linear_combination` *check* your stated certificate instead of searching for one (checking is cheap; finding at scale was the memory bomb). The companion repos' rule — no blanket automation in fat contexts — is this box, written as policy. And note the shape of the lesson: it is the *same* headroom discipline as the code's (bound your intermediates, spend your budget consciously), applied one level up, to the proof itself.

**⋆ Aha**

There is a deep symmetry in that fix worth savoring: the *proof* was restructured exactly the way the *code* was — into small steps with controlled intermediate size. Delayed carries in the implementation; small lemmas in the verification. Bounded limbs; bounded contexts. Good proofs and good fast code turn out to obey the same engineering aesthetics. This is not a coincidence; both are fighting combinatorial growth with structure.

**Four forks, one method, real divergence.** The companion projects verify not just upstream `curve25519-dalek` but three production forks (Solana's, RISC Zero's, Betrusted's) — each against *its own* extraction. Worth it? The audit found the forks implement the same constant-time conditional selection three different ways (a `subtle` crate trait, a volatile-read `black_box`, a hand-rolled arithmetic mask), and one fork reorders instructions in point doubling. All correct — *provably*, now — but the divergence is exactly the kind of thing that silently breaks when someone "harmonizes" code during a rebase. Per-fork verification is not pedantry; it is how you notice that "the same library" isn't.

**△ Pitfall**

A verified fork is verified *at a commit*. Change one line of arithmetic and the certificate is stale — that is a feature (the proof *should* break when the code changes), but it means verification is a *process wired into maintenance*, not a trophy. The companion repos ship `check.sh` scripts that re-extract and re-verify from scratch; treat those as the project's pulse, not as CI decoration.

## 10.4 Reading a certificate like a professional

Suppose a stranger hands you a repository claiming "formally verified field arithmetic." Chapter 11 gives you the full audit protocol, but the field-layer questions you can already ask are these:

- **What is denoted?** Find the denotation function. Does it map the *extracted* representation (good) or a hand-written lookalike (Chapter 8's hole)?
- **Is the square complete?** Value equation *and* bounds propagation *and* the `.ok` clause — a spec that assumes success proves nothing about overflow.
- **Is the quantifier honest?** ∀ `a b` with bounds hypotheses, or a handful of `examples` on constants dressed up as coverage?
- **Does anything say `sorry`?** One `sorry` anywhere in the dependency chain and the certificate is decorative. The checker will tell you; ask it.

**>_ Try it yourself**

Do the audit for real: open `dalek-ed25519-verified`, find the denotation, pick the `sub` spec, and check the three clauses against the list above. Then run the repo's `check.sh` and watch the whole pyramid rebuild. Total time: one coffee. Skill acquired: permanent.

## Exercises

**Exercise 10.1.** Field subtraction in dalek computes $a - b$ as $a + (16p - b)$ limb-wise (adding a multiple of $p$ keeps limbs positive — recall `Nat` truncation!). State the commuting square for `sub` including the exact multiple-of-$p$ fact you would need as a lemma, and explain why $16p$ rather than $p$.

**Exercise 10.2.** The Fermat inversion chain computes $a^{p-2}$ via 254 squarings and 11 multiplications. Verify the exponent bookkeeping for the first three steps of dalek's actual chain: $a^2$, $a^9 = (a^2)^{2\cdot 2} \cdot a$, $a^{11} = a^9 \cdot a^2$. (The full chain is exercise-by-induction: each step's exponent is a sum of previous ones — addition-chain arithmetic.)

**Exercise 10.3.** (Discussion) Step 5 refuses to trust the squaring shortcut and verifies `square` separately from `mul`. A colleague argues "square is just `mul a a`, reuse the theorem." What, concretely, would that argument miss about the code under verification?

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

**Solution 10.1.**

*Pathway.* Assemble from parts you own: the two-clause spec shape (Chapter 9), the multiple-of-$p$ freedom (the telescope worked example), and the positivity audit (this chapter's $16p$ worked example). The exercise is really asking you to *compose* them into one statement.

*Answer.* The commuting square for `sub`, with the lemma named:

$$\text{Bnd54}(a) \to \text{Bnd54}(b) \to \exists c,\ \texttt{sub}\, a\, b = \texttt{.ok}\ c\ \wedge\ \text{Bnd56}(c)\ \wedge\ [\![c]\!] = [\![a]\!] - [\![b]\!],$$

resting on the lemma $[\![16P]\!] = 16p \equiv 0 \pmod p$ (the telescope, scaled by 16), which justifies $[\![a + (16P - b)]\!] = [\![a]\!] + 0 - [\![b]\!]$. Why 16 and not $p$ itself (i.e. $k = 1$): positivity of every limb of $kP - b$ requires the *smallest* limb constant, $k(2^{51} - 19)$, to dominate the *largest* allowed limb of $b$, $2^{54} - 1$; $k = 1$ gives $2^{51} - 19 \ll 2^{54}$ — hopeless — and even $k = 8$ falls 151 short, as the worked example computed. First power of two that clears the bar: 16.

### Solution 10.2.

*Pathway.* Squaring doubles the exponent; multiplying adds. Track only exponents and check each claimed identity by substitution — the field elements are irrelevant to the bookkeeping, which is the insight that makes the whole verification tractable.

*Answer.* $a^2$: exponent 2 ✓ (one squaring). $a^9 = (a^2)^{2 \cdot 2} \cdot a$: squaring $z_2$ twice gives exponent $2 \cdot 2 \cdot 2 = 8$; multiplying by $a$ adds 1: exponent 9 ✓ (two squarings, one multiplication). $a^{11} = a^9 \cdot a^2$: $9 + 2 = 11$ ✓ (one multiplication). The induction that carries the rest: each ladder stage claims $(2^j - 1) \cdot 2^k + (2^k - 1) = 2^{j+k} - 1$ for the block sizes $(j, k)$ it uses — verify it once in general: $(2^j - 1)2^k + 2^k - 1 = 2^{j+k} - 2^k + 2^k - 1 = 2^{j+k} - 1$ ✓ — and every remaining line of the chain is an instance. Addition-chain verification in full: substitute, cancel, done.

### Solution 10.3.

*Pathway.* Ask what *differs between the two code paths*, not between the two mathematical functions. The math of `square a` and `mul a a` agree; the *instructions* do not.

*Answer.* Three concrete misses. (1) `square` is a *different routine*: it exploits symmetry ($a_i a_j$ appears twice for $i \neq j$, so it computes $2a_i a_j$ once), which changes the column expressions, the constant factors, and therefore the *bounds* — the doubled terms eat one extra headroom bit that `mul`'s proof never had to account for. (2) A bug in that symmetry trick — the classic one is a missing factor of 2 on exactly one cross term — lives *only* in `square`'s code path; a theorem about `mul` quantifies over `mul`'s instructions and says nothing about it. (3) Even the `.ok` clause differs: overflow behavior depends on the actual intermediate expressions, not on the mathematical value. The one-line summary for design reviews: *we verify code paths, not functions — if the optimizer wrote a second path, the verifier owes a second theorem.* (This is also exactly why the four forks each got their own proofs: same math, four instruction streams.)

**⋈ Checkpoint**

You should now be able to: state the field implementation certificate and every quantifier in it; recite the campaign order and justify why bounds and `reduce` come early; tell the memory-wall story and its decomposition moral; and audit a stranger's field-layer claim with four pointed questions.

# Chapter 11

# Honesty, Axioms, and the Art of Trusting Proofs

## 11.1 "Formally verified" is a claim, not a spell

The phrase *formally verified* has marketing gravity, and marketing gravity attracts abuse. This chapter arms you with the auditor's toolkit: what a Lean certificate actually rests on, how to interrogate it in one command, and the specific ways a verification claim can be hollow while every file compiles. Nothing here is hypothetical — each failure mode appears in the wild, and the discipline below is the one the companion projects hold themselves to.

## 11.2 The trust ledger

When the kernel accepts `fieldImplementation`, what exactly are you being asked to believe? Lean can tell you — precisely:

```
#print axioms fieldImplementation
-- 'fieldImplementation' depends on axioms:
-- [propext, Classical.choice, Quot.sound]
```

`#print axioms` walks the *entire* dependency tree of a theorem — every lemma, every lemma's lemmas, down to bedrock — and reports every assumption found there. The three names above are Lean's standard trio (propositional extensionality, classical choice, quotient soundness): ordinary classical mathematics, accepted by working mathematicians, scrutinized by logicians for a century. A certificate reporting exactly these three is called **axiom-clean**. Anything *else* in that list is a custom assumption someone added — and the whole audit consists of reading that list and asking whether you believe each entry.

**⁎ The big idea**

A machine-checked theorem is a receipt with a complete list of its own assumptions — something prose mathematics has never had. But the receipt only protects people who read it. **The**

**one-command audit:** run `#print axioms` on the headline theorem; anything beyond `[propext, Classical.choice, Quot.sound]` is where the bodies are buried. Make this reflex, and no verification claim can hide from you.

∠ **Pen and paper: reading three audit transcripts — a training set**

Here are three `#print axioms` outputs. Before reading the verdicts, write your own one-line assessment of each; the answers follow.

```
-- Transcript A
'fieldImplementation' depends on axioms:
[propext, Classical.choice, Quot.sound]

-- Transcript B
'mulCorrect' depends on axioms:
[propext, Classical.choice, Quot.sound, sorryAx]

-- Transcript C
'groupLawComplete' depends on axioms:
[propext, Classical.choice, Quot.sound, edDenomNonzero]
```

*Transcript A*: the standard trio and nothing else — axiom-clean. Remaining questions are the human ones (is the *statement* non-trivial? is the *model* the shipping code?), but the logical ledger is closed.
*Transcript B*: `sorryAx` is the placeholder axiom behind `sorry` — somewhere in this theorem's dependency tree an obligation was skipped, and the tree remembers even though the file that contains it may be ten imports away. This certificate is decorative, *regardless of anything else about it.* (Note how cheap the detection was: the tool walked ten thousand dependencies so you didn't have to.)
*Transcript C*: the interesting one. A custom axiom, `edDenomNonzero` — from the name, someone *assumed* the Edwards denominators are nonzero rather than proving completeness. Two possible worlds: it is declared in the trusted-base ledger with a justification (defensible — but then why does Chapter 12 prove it from the non-squareness of $d$ with modest effort? suspicious laziness at best), or it is undocumented — in which case the headline "group law verified, complete addition!" is quietly assuming the very fact that makes completeness interesting. Your move as auditor: `grep` for the axiom's declaration, check the ledger, and ask the vendor Chapter question (3). The general skill: **a custom axiom's *name* tells you which claim to stop believing until shown the ledger entry.**

## 11.3 A field guide to hollow certificates

Compiling proofs can still be worthless. The four classic ways, in increasing order of subtlety:

**1. The `sorry`.** Lean's placeholder accepts any goal with a warning. Fine for work in progress; fatal in a certificate, because downstream theorems inherit the hole silently. Detection: the compiler warns, and `#print axioms` shows `sorryAx`. Trivial to catch — if you look.

**2. The smuggled axiom.** Stuck on a lemma? `axiom` makes it true. Sometimes legitimate (see the

trusted-base section below); rotten when undisclosed — a proof "of" the group law that axiomatizes the hard half of the group law is theater. Detection: `#print axioms`, always.

**3. The trivial specification.** The most instructive one. Consider:

```
theorem mul_correct : ∀ a b, ∃ c, mul a b = c := by
  intro a b; exact ⟨_, rfl⟩   -- checks! and says NOTHING
```

Kernel-approved, axiom-clean, and utterly empty: "mul returns whatever it returns." No tool catches this, because nothing is wrong *formally* — the defect is that the statement doesn't say what the reader assumes it says. The only detector is a human reading the *statement* (never mind the proof) and asking: *if the code were wrong, would this theorem fail?* For the trivial spec, the answer is no — a buggy `mul` satisfies it identically.

**∠ Pen and paper: the substitution test — executing "would a wrong program pass?"**

The italicized question deserves a mechanical procedure, because you will ask it of every certificate you ever audit. The procedure: *invent the worst program of the right type, substitute it into the statement, and see whether the statement notices.* Run it on three candidate specs for field multiplication, with the adversarial implementation $\texttt{mul}_0$ := "ignore the inputs, return the zero array."

*Spec A (the trivial one):* $\forall a\, b,\ \exists c,\ \texttt{mul}\ a\ b = c$. Substitute $\texttt{mul}_0$: is there a $c$ with $\texttt{mul}_0\, a\, b = c$? Yes — $c = (0,0,0,0,0)$, for every input. **Spec A holds for the adversary.** Verdict: empty.

*Spec B (existence-of-ok):* $\forall a\, b,\ \mathrm{Bnd}(a) \to \mathrm{Bnd}(b) \to \exists c,\ \texttt{mul}\ a\ b = \texttt{.ok}\ c \wedge \mathrm{Bnd}(c)$. Substitute: $\texttt{mul}_0$ always returns `.ok` of the zero array, and the zero array is certainly bounded. **Spec B holds for the adversary too.** Verdict: overflow-safety only — honest as far as it goes, but a reader who took it for correctness was reading the title, not the theorem.

*Spec C (the commuting square):* add the clause $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$. Substitute and pick the one-line counterexample: $a = b = (1,0,0,0,0)$, so $[\![a]\!] \cdot [\![b]\!] = 1 \cdot 1 = 1$, while $[\![\texttt{mul}_0\, a\, b]\!] = [\![(0,\ldots,0)]\!] = 0 \neq 1$ in $\mathbb{F}_p$. **Spec C refutes the adversary.** Verdict: this one has teeth.

The test took four lines of substitution per spec and required no Lean at all — it is a *reading* skill. Note also what it is *not*: it cannot certify a spec as strong (a cleverer adversary might slip through where $\texttt{mul}_0$ was caught); it is a smoke detector, not a proof. But every hollow certificate in the field-guide above fails it within a minute, which is a remarkable return on four lines of pen and paper.

**4. The wrong model.** The proof is real, the spec is strong — but the thing verified isn't the thing that ships: a hand-transcription (Chapter 8), a stale extraction, a simplified semantics. Detection: trace the chain from artifact to source — extraction scripts, pinned tool versions, regeneration instructions. If the chain can't be replayed, the claim is about an orphan.

**△ Pitfall**

Rank these by danger and notice the inversion: the crude failures (`sorry`, smuggled axioms) are machine-detectable in seconds, while the subtle ones (trivial specs, wrong models) defeat every automated check and yield only to a thoughtful reader. Verification does not eliminate the need for human judgment; it *concentrates* all of it into two small, well-lit places — the statement and

the model. That concentration is the gift; squandering it by not reading the statement is the sin.

## 11.4 Honest boundaries: the trusted base

Real projects meet real limits: SHA-512's compression function, SIMD backends no translator models, foreign function calls. The honest move is not to pretend, but to *declare*: state the unverified piece as an explicit assumption, document it in a ledger (the companion repos call theirs `TRUSTED-BASE.md`), and let `#print axioms` carry the disclosure to every downstream theorem automatically.

```
-- Declared, documented, and visible in every audit forever:
axiom sha512_spec : ∀ msg, Sha512.hash msg = SHA512_ideal msg
```

A signature-layer certificate honestly reads: *EdDSA verification is correct, GIVEN the hash behaves ideally and GIVEN the documented backend assumptions* — with both *given*s machine-visible. Compare the two postures: "everything verified!" (and hope nobody checks) versus "these two assumptions, this ledger, audit me" — the second is both humbler and *stronger*, because its claim survives the audit.

The companion projects add one more layer of candor worth copying: a *failure map*. Their control repository documents the dead ends — tactic patterns that exhaust memory, extraction scopes that drag in the world, proof styles that don't scale — each with the tell that identifies it early. Knowledge of where the cliffs are is part of the method, and pretending the cliffs don't exist is how the next person walks off one.

**⋆ Aha**

Notice the running theme: at every level, the methodology converts *invisible* trust into *visible* trust. Extraction made the code-to-model step visible; two-clause specs made operating envelopes visible; `#print axioms` makes logical debts visible; the trusted-base ledger makes engineering limits visible. Formal verification's deepest product is not certainty — it is **legibility of exactly what remains uncertain**.

**>_ Try it yourself**

Audit the real thing. In `dalek-ed25519-verified`, run `#print axioms` on `fieldImplementation` and `edwardsImplementation` — confirm the clean trio. Then read `TRUSTED-BASE.md` and match each entry to where it would surface in an audit. Finally, write a deliberately trivial spec for `add`, prove it in one line, and observe that every automated check passes. Keep that file open for one full minute. That minute is the chapter.

## Exercises

**Exercise 11.1.** For each hollow-certificate species, name its detector: (a) `sorry`; (b) smuggled axiom; (c) trivial spec; (d) wrong model. Which two can a CI pipeline catch mechanically, and what CI check

would you write for each?

**Exercise 11.2.** Strengthen this spec until a buggy implementation would fail it: `theorem sub_ok : ∀ a b, ∃ c, sub a b = .ok c`. (List what's missing: bounds hypotheses? bounds propagation? the value equation? Compare with Chapter 9's two-clause shape.)

**Exercise 11.3.** A vendor's whitepaper says: "Our signature library is formally verified in Lean." Draft the five questions you would send them, in priority order, and the answer you would require for each before relying on the claim. (You now know all five.)

**Exercise 11.4.** (Discussion) The trusted-base `axiom` for SHA-512 and the smuggled `axiom` for a hard lemma are the *same language feature*. Articulate the difference in one sentence — it is not technical.

**Exercise 11.5.** (Paper — the substitution test, on the toy system) For the mini-arithmetic of Chapter 9's exercises, a vendor ships: $\forall a\, b,\ \mathrm{Bnd}(a) \to \mathrm{Bnd}(b) \to [\![\mathtt{add}\, a\, b]\!] + [\![(0,0)]\!] = [\![\mathtt{add}\, a\, b]\!]$. It compiles; it is axiom-clean. Run the substitution test: does the buggy `add'` of `exercises/Ch12.lean` pass this spec? What is the spec actually about?

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 11.1.

*Pathway.* For each species, ask *where the defect physically lives* — in a warning, in a dependency list, in the meaning of a statement, in the relation between artifact and world — and match it to the tool that inspects that place.

*Answer.* (a) `sorry`: detector is the compiler warning and `sorryAx` in `#print axioms`; CI check: grep build output for `uses 'sorry'` and fail (the companion repos' `check.sh` does exactly this — warnings, unlike comments, cannot be spoofed by prose). (b) Smuggled axiom: detector is `#print axioms` on every shipped certificate; CI check: assert the output is exactly `[propext, Classical.choice, Quot.sound]`, plus a source-level gate refusing `axiom` declarations outside the sanctioned model directory. (c) Trivial spec: *no mechanical detector* — the substitution test, executed by a human reading the statement. (d) Wrong model: no mechanical detector either — the replay test: can a stranger re-extract from pinned sources and reproduce the artifact? CI can *support* it (pin versions, re-run extraction, diff), but the judgment "this model is the shipping code" remains human. The pattern of the answers: mechanical failures get mechanical detectors; semantic failures get disciplined humans. Budget your skepticism accordingly.

### Solution 11.2.

*Pathway.* Apply the substitution test to your own strengthening: after each added clause, ask which adversarial implementation still passes. Stop when the only survivor is a correct one.

*Answer.* Stepwise. Start: $\exists c$, `sub` $a\ b$ = `.ok` $c$ — passes for "return zero always." Add bounds hypotheses and propagation (Bnd54 on inputs, Bnd56($c$)): still passes for return-zero. Add the value

clause $[\![c]\!] = [\![a]\!] - [\![b]\!]$: return-zero dies ($a = (1, 0, \dots), b = 0$ gives $1 \neq 0$), and so does every other wrong-value implementation, by definition. Final spec — the full two-clause square:

$$\mathrm{Bnd54}(a) \to \mathrm{Bnd54}(b) \to \exists c,\ \texttt{sub}\, a\, b = \texttt{.ok}\, c \wedge \mathrm{Bnd56}(c) \wedge [\![c]\!] = [\![a]\!] - [\![b]\!].$$

Each clause pulls real weight: `.ok` excludes panics/overflow, the bound feeds the next operation, the value equation carries correctness. Delete any one and re-run the substitution test to see what sneaks back in — that exercise-within-the-exercise is the fastest way to internalize why all three clauses recur through the entire companion codebase.

**Solution 11.3.**

*Pathway.* Order the questions by how cheaply they kill a hollow claim: axioms first (one command), then sorries, then statement, then model, then reproducibility.

*Answer.* The five, in priority order, each with its required answer: (1) *"What does `#print axioms` report on the headline theorems?"* — exactly the standard trio, or each extra axiom documented in a trusted-base ledger. (2) *"Does any shipped file contain `sorry`, and does CI fail on the warning?"* — no, and yes. (3) *"Show me the exact statement of the main theorem"* — it must contain a universally quantified value equation against an independent mathematical definition (substitution test on the spot). (4) *"What artifact was verified, and how does it relate to the code you ship?"* — a model extracted mechanically from the shipping sources, tool versions pinned. (5) *"Can I reproduce the check?"* — a one-command script re-extracts, re-compiles, re-audits. A vendor who answers all five without flinching is, empirically, not hiding anything — and a vendor who stumbles on (3) has told you which chapter of this book to reread on the flight home.

**Solution 11.4.**

*Pathway.* Strip away everything technical — both are `axiom` — and what remains is a relationship between author and reader.

*Answer.* One sentence: *the trusted-base axiom is disclosed as a debt — named, documented, and priced for the reader's own judgment — while the smuggled axiom is disclosure's opposite: a debt hidden so the certificate can impersonate a stronger claim.* Same instruction to the kernel; opposite speech act to the human. (Which is why the audit command matters so much: it makes the speech act verifiable.)

**Solution 11.5.**

*Pathway.* Substitution test, by the book: does the statement mention the program's *relationship to anything external*? Look closely at what appears on both sides of the equation.

*Answer.* `add'` passes — and so does every function of the correct type, including "return $(3, 1)$ always." The spec has the form $X + [\![(0, 0)]\!] = X$ where $X := [\![\texttt{add}\, a\, b]\!]$ appears identically on both sides; since $[\![(0, 0)]\!] = 0$, it reduces to $X + 0 = X$ — a true fact *about the field*, in which `add` occurs only as an inert label. The spec is about $\mathbb{Z}/15\mathbb{Z}$'s additive identity, dressed in the vendor's function name. This species — *true theorem, wrong subject* — is the subtlest hollow certificate: it even survives a careless substitution test if you only check that the statement mentions `add`. The refined test: does the implementation appear on *one side only* of an equation whose other side is independent of it? If it cancels, it was never being tested.

**⋈ Checkpoint**

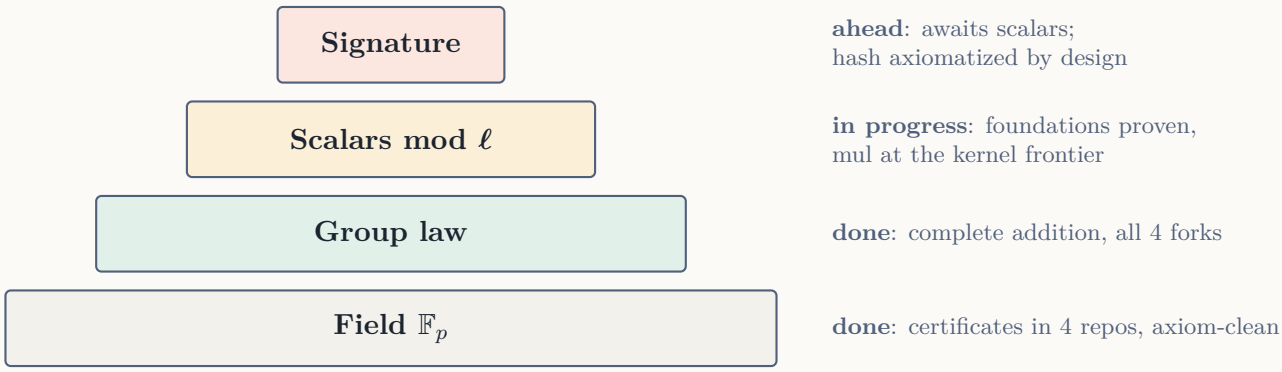
You should now be able to: run and interpret the one-command audit; name the standard three axioms and greet anything else with suspicion; explain why trivial specs and wrong models defeat automation and what defeats *them*; and argue — with conviction — why a declared trusted base is stronger, not weaker, than a claim of totality.

# Chapter 12

# The Pyramid: From Field to Signature, and Where You Come In

## 12.1 The view from the field layer

Chapter 10 left us holding a verified field. A signature scheme is still three stories up. This closing chapter walks the remaining layers — what each one *states*, what makes each one *hard*, and where the campaign stands as this book goes to press — then hands you the map and the keys.




## 12.2 The group law: geometry becomes algebra

An elliptic curve is a set of points $(x, y)$ satisfying an equation; for Ed25519 it is the *twisted Edwards* curve $-x^2 + y^2 = 1 + d\,x^2y^2$ over $\mathbb{F}_p$. The miracle: these points form a *group* under the addition law

$$(x_1, y_1) + (x_2, y_2) \;=\; \left( \frac{x_1y_2 + x_2y_1}{1 + d\,x_1x_2y_1y_2}, \; \frac{y_1y_2 + x_1x_2}{1 - d\,x_1x_2y_1y_2} \right).$$

Two facts make this law a verifier's dream, and both carry Edwards-curve signatures for exactly this reason. First, it is **complete**: for the Ed25519 parameters those denominators are *never zero* — no special cases for doubling, no branch for the identity, hence constant-time-friendly code with no rarely-taken paths for bugs to hide in. (The proof, due to Bernstein and Lange, is a jewel of quiet

algebra: if a denominator vanished, $d$ would have to be a square in $\mathbb{F}_p$ — and it is not, which is a `decide`-scale fact away from primality.)

**∠ Pen and paper: the completeness argument, derived to its hinge**

The Bernstein–Lange proof rewards a full pen-and-paper walk — symbols, not toy numbers, because the argument *is* the real one at every size. We run it on the Edwards curve $x^2 + y^2 = 1 + dx^2y^2$ with $d$ a non-square (Ed25519's twisted form adds decorations; the skeleton is identical, and the exercises hand you the twist). Suppose, for contradiction, points $(x_1, y_1)$, $(x_2, y_2)$ on the curve make a denominator vanish: $\varepsilon := d\,x_1x_2y_1y_2 \in \{\pm 1\}$. A product equal to $\pm 1$ has no zero factor, so all four coordinates are nonzero. Three moves, each checkable by expansion:
*Move 1 — square the assumption.* From $\varepsilon^2 = 1$: $d^2x_1^2x_2^2y_1^2y_2^2 = 1$, which rearranges to

$$1 \;=\; dx_1^2y_1^2 \cdot dx_2^2y_2^2.$$

*Move 2 — expand a well-chosen square.* Using $x_1^2 + y_1^2 = 1 + dx_1^2y_1^2$ (the curve, point 1) and $\varepsilon x_1y_1 = dx_1^2y_1^2\,x_2y_2$ (multiply the definition of $\varepsilon$ by $x_1y_1$):

$$(x_1 + \varepsilon y_1)^2 = x_1^2 + y_1^2 + 2\varepsilon x_1y_1 = 1 + dx_1^2y_1^2 + 2\,dx_1^2y_1^2\,x_2y_2.$$

*Move 3 — substitute Move 1's* 1 *and factor.* Replace the leading 1 by $dx_1^2y_1^2 \cdot dx_2^2y_2^2$ and pull out $dx_1^2y_1^2$:

$$(x_1 + \varepsilon y_1)^2 = dx_1^2y_1^2(dx_2^2y_2^2 + 1 + 2x_2y_2) = dx_1^2y_1^2(x_2^2 + y_2^2 + 2x_2y_2) = d\,\big(x_1y_1(x_2 + y_2)\big)^2,$$

where the middle equality used the curve equation for point 2 backwards ($1 + dx_2^2y_2^2 = x_2^2 + y_2^2$). Now the hinge: if $x_2 + y_2 \neq 0$, divide —

$$d = \left(\frac{x_1 + \varepsilon y_1}{x_1y_1(x_2 + y_2)}\right)^2,$$

$d$ **is a square**. And if $x_2 + y_2 = 0$, rerun Moves 2–3 with $(x_1 - \varepsilon y_1)^2$ to get $d \cdot (x_1y_1(x_2 - y_2))^2$ instead — $x_2 - y_2$ cannot *also* vanish (both would force $x_2 = y_2 = 0$). Either way $d$ is a square in $\mathbb{F}_p$. But Ed25519's $d$ is *not* — one Legendre-symbol computation, $d^{(p-1)/2} \equiv -1 \pmod p$, checkable by exactly the square-and-multiply ladder of Chapter 7, established once as a constant fact in the verified development. Contradiction; no denominator ever vanishes. Savor the architecture: one quadratic-residue bit about one constant buys the *total absence of special cases* from every point addition ever executed — and thereby the absence of the rarely-taken branches where Chapter 1's bugs live. That is what "a curve chosen for verifiability" means in practice.

Second, the implementation represents points *projectively* (extended coordinates $(X : Y : Z : T)$, avoiding division entirely) — so the layer has its own denotation, $(X : Y : Z : T) \mapsto (X/Z, Y/Z)$, and its own commuting squares built on the field layer's specs. Same movie, one floor up: the verified group law in the companion repos is precisely the statement that projective point addition implements the rational formula above, all bounds included, for each fork's own extraction.

## 12.3 Scalars: a second field, and a frontier

The group of curve points has order $8\ell$ with $\ell = 2^{252} + 27742\ldots$ prime. Signature arithmetic happens in exponents — multiples of points — so it is arithmetic mod $\ell$: a *second* finite field, with its own Rust implementation (radix-52 limbs, Montgomery multiplication) and its own denotation bridge. Nothing conceptually new — which is itself the lesson: the method *scales sideways* without new ideas.

**∠ Pen and paper: sizing the group — real constants, three-line audits**

The scalar layer's constants invite the same pen-and-paper audits as the field's. The group order is $8\ell$ with

$$\ell \;=\; 2^{252} + 27742317777372353535851937790883648493,$$

that 38-digit tail being an inseparable companion of anyone who works on this layer. Three audits, each a few lines:
*(1) Consistency with the curve.* A theorem of Hasse says an elliptic curve over $\mathbb{F}_p$ has $p + 1 - t$ points with $|t| \le 2\sqrt{p}$ — so about $2^{255}$ points, within $2^{128.5}$-ish. Check the claimed order: $8\ell = 2^3 \cdot 2^{252} + 8 \cdot (38\text{-digit}) = 2^{255} + (\text{a number} < 2^{129})$. Sits exactly in Hasse's window around $p + 1 \approx 2^{255}$ ✓. The claimed structure is at least arithmetically possible — a thirty-second sanity check worth running on *any* curve parameter set someone hands you.
*(2) The tail is not decoration.* Could a signature library "round" $\ell$ to $2^{252}$ — who would notice? Anyone reducing a 256-bit hash output mod $\ell$: the reductions differ on roughly a $2^{-124}$ slice of inputs (the interval lengths differ by the tail), and the certified theorem `L_val` in all four companion repos — *the transpiled constant equals $\ell$, digit for digit* — exists precisely because "a constant nobody can eyeball" is where typos retire. The proof is one `decide`-scale comparison, and it has teeth: change one digit of the Rust constant and `check-scalar.sh` fails.
*(3) Why the 8s in the verification equation.* The full group has order $8\ell = 2^3 \cdot \ell$, so (by the structure of finite abelian groups) it decomposes as $\mathbb{Z}_8$-part $\times$ $\mathbb{Z}_\ell$-part: every point splits as $X = T + Y$ with $T$ of order dividing 8 ("torsion") and $Y$ of order dividing $\ell$. Multiply by 8:

$$8X \;=\; 8T + 8Y \;=\; \mathcal{O} + 8Y \;=\; 8Y$$

— the torsion component is annihilated, whoever chose it. An attacker who tampers with a public key by adding a small-order point $T$ changes $X$ but not $8X$; the cofactored equation $8sB = 8R + 8kA$ is therefore immune to a whole class of malleability games that the uncofactored $sB = R + kA$ is not. Three multiplications by 8, bought by exactly the three-line computation above.

The engineering, however, has a frontier, and this book has told you enough truth to locate it precisely. Scalar Montgomery multiplication mixes $2^{256}$-scale coefficients into single certificate steps; this is the kernel-capacity wall of Chapter 10, and it marks the current working edge of the campaign: additions and the foundational constants are certified (including the pleasing theorem that the code's constant `L` *is* $\ell$); the multiplication path is a construction site with scaffolding — decomposed lemmas, isolated carry steps — mid-assembly, honestly labeled in-repo.

## 12.4 The apex: what "verified signature" will say

EdDSA verification accepts $(R, s)$ on message $m$ under key $A$ iff

$$8sB \;=\; 8R + 8\,H(R, A, m)\,A$$

in the curve group ($B$ the base point, $H$ = SHA-512, the 8s absorbing the cofactor). The apex certificate will state: *the extracted verification routine returns true exactly when this equation holds* — given the two declared trusted-base entries you can already predict: SHA-512 as an ideal hash (axiomatized by design — hash function correctness is a different mathematical universe), and the SIMD point-multiplication backends (untranslatable, documented). Everything between those declared boundaries and the field bedrock: kernel-checked, axiom-clean, per fork.

Read that sentence again with Chapter 11 eyes: it is a *smaller* claim than "Ed25519 is verified!" — and that is exactly why you can believe it.

## 12.5 What you now know, and where to take it

Take inventory. You can read a goal state and drive a proof; you know which decision procedure owns which arithmetic fragment; you can build a denotation bridge and state a two-clause spec; you can certify a prime with a witness tree; you can audit anyone's certificate in one command and four questions. That skill set is not Ed25519-specific — it is the working method of machine-checked mathematics applied to systems, and elliptic curves were merely your first campaign.

Where to go from here, in increasing order of ambition:

- **Read a real proof end-to-end.** `FieldSpec.lean` in `dalek-ed25519-verified`, top to bottom, with this book as the decoder ring. Budget an afternoon; expect the odd hour of humility.
- **Extend the pyramid.** The scalar layer's open lemmas are decomposed, labeled, and waiting; the repos' `CONTRIBUTING` notes state exactly what a finished brick looks like (spec shape, axiom audit, check-script entry). Frontier work, undergraduate-accessible.
- **Verify something of yours.** Pick a 200-line pure function you actually use — a parser, a checksum, a data structure — write its denotation (what does it *mean*?), state the square, prove it. The first solo bridge is the moment this stops being a course.
- **Go deeper into the theory.** *Theorem Proving in Lean 4* (the official text), *Mathematics in Lean* (Mathlib's course), and the Lean Zulip — an unusually welcoming expert community — are the standard next doors.

### Further reading, annotated

- *Theorem Proving in Lean 4* (Avigad, de Moura, et al.; free online) — the official text. Read it *after* this book's Chapters 2–5 and it will feel like meeting the extended family of ideas you already know; its dependent-type chapters go far beyond our needs and are worth the trip.
- *Mathematics in Lean* (the Mathlib community course) — hands-on Mathlib fluency: naming conventions, search strategies, the algebra hierarchy. The fastest cure for "I know the fact exists but not its name," which will be your main bottleneck after this book.

- *The Lean Zulip* (`leanprover.zulipchat.com`) — where the community lives. Unusually welcoming to beginners; search before asking, then ask well: a minimal example plus the goal state gets expert answers in hours.
- Bernstein & Lange, *Faster addition and doubling on elliptic curves* (2007) — the completeness proof this chapter's worked example walked; readable with this book's preparation, and a model of what "designed for implementers" mathematics looks like.
- The RFC for EdDSA (RFC 8032) — the signature scheme as deployed, cofactor-8s and encoding details included. Read the verification equation section against this chapter and notice how much sharper your questions have become.
- Project Everest / HACL* and Fiat Crypto — the two other major verified-crypto lineages (F*-based and Coq-based respectively), both shipping in real TLS stacks and browsers. Reading their claims with your Chapter 11 toolkit is instructive in both directions: the methods differ, the honest-boundary discipline rhymes.

**⋆ Aha**

One last reframe, the one this book was secretly about. "Formal verification" sounds like bureaucracy — forms, stamps, compliance. What you actually practiced is closer to *engineering's version of the scientific method*: make the claim precise enough to be falsifiable, then let an incorruptible referee try to falsify it, then publish the referee's report with the assumptions itemized. Cryptography needed that discipline first because its failures are silent and adversarial. It will not need it last.

**>_ Try it yourself**

The graduation exercise. In the mini-system from `exercises/Ch09.lean`, the file `exercises/Ch12.lean` plants a *deliberate off-by-one carry bug* in a variant `add'` — of exactly the species from Chapter 1: correct on all limb pairs except a thin boundary slice. Your final tasks: (1) write the spec — watch it *refuse to prove*; (2) extract the counterexample from the stuck goal state; (3) confirm by `#eval`; (4) fix the code and finish the proof. That arc — spec, refusal, counterexample, fix, certificate — is the entire profession in miniature. Welcome to it.

## Exercises

**Exercise 12.1.** (Paper) Verify Move 2 and Move 3 of the completeness worked example by full expansion — every term written out, nothing skipped. Then adapt the argument's *first* move to the twisted curve $-x^2 + y^2 = 1 + dx^2y^2$: where does the $-1$ enter, and why does the argument want $-1$ to be a *square* mod $p$? (Hint: $p = 2^{255} - 19 \equiv 1 \pmod 4$, and for such primes $-1$ is a quadratic residue — which is not an accident of the curve designers.)

**Exercise 12.2.** (Paper) In the group decomposition $X = T + Y$ (torsion of order dividing 8 plus a $\mathbb{Z}_\ell$ component), verify: (a) $8X = 8Y$; (b) $8Y \neq \mathcal{O}$ whenever $Y \neq \mathcal{O}$ — why does this need $\gcd(8, \ell) = 1$, and where does the argument use that $\ell$ is prime and $> 8$? (c) Conclude what an attacker who adds a small-order point to a public key changes, and what they provably cannot change.

**Exercise 12.3.** (Audit drill) Write down, from memory, the complete list of what the apex certificate will *assume* (its trusted base) and what it will *establish*, then check yourself against this chapter's apex section. Anything you forgot is the thing to reread before you audit a real system.

## Solutions and pathways

Attempt every exercise before reading on — a pathway teaches ten times as much after you have been genuinely stuck. Each solution below starts with the *pathway* (how you find the answer) and only then the answer itself.

### Solution 12.1.

*Pathway.* For the expansion: Move 2 is the binomial square plus two substitutions — write $(x_1+\varepsilon y_1)^2 = x_1^2+2\varepsilon x_1y_1+y_1^2$, then replace $x_1^2+y_1^2$ via the curve and $\varepsilon x_1y_1$ via the definition. Move 3 is distributing $dx_1^2y_1^2$ and recognizing a perfect square. For the twist, transport the curve equation and re-run Move 2.

*Answer.* Move 2 fully expanded: $(x_1+\varepsilon y_1)^2 = x_1^2+2\varepsilon x_1y_1+y_1^2$; curve gives $x_1^2+y_1^2 = 1+dx_1^2y_1^2$; and $\varepsilon x_1y_1 = (dx_1x_2y_1y_2)(x_1y_1) = dx_1^2y_1^2x_2y_2$ — sum the three pieces to get the displayed line ✓. Move 3: $dx_1^2y_1^2(dx_2^2y_2^2+1+2x_2y_2)$; the curve for point 2 says $1+dx_2^2y_2^2 = x_2^2+y_2^2$, so the bracket is $x_2^2+2x_2y_2+y_2^2 = (x_2+y_2)^2$, and $dx_1^2y_1^2(x_2+y_2)^2 = d(x_1y_1(x_2+y_2))^2$ ✓. For the twisted curve: $x_1^2+y_1^2$ no longer appears — the curve supplies $y_1^2-x_1^2$ — so the well-chosen square must mix a factor $\sqrt{-1}$ into the $x$'s (expand $(\sqrt{-1}\,x_1+\varepsilon y_1)^2 = -x_1^2+y_1^2+2\varepsilon\sqrt{-1}\,x_1y_1$: the curve's left-hand side appears exactly). That $\sqrt{-1}$ must *exist* in $\mathbb{F}_p$ for the argument to run — hence the requirement that $-1$ be a square, guaranteed by $p \equiv 1 \pmod 4$. The designers chose the twist $a=-1$ *because* it is a square mod this $p$: speed came from the twist, completeness survived because of the residue class. Parameters this well-matched are chosen, not lucky.

### Solution 12.2.

*Pathway.* All three parts are order bookkeeping: $nZ = \mathcal{O}$ exactly when the order of $Z$ divides $n$.

*Answer.* (a) $8X = 8T+8Y$; the order of $T$ divides 8, so $8T = \mathcal{O}$, leaving $8Y$ ✓. (b) The order of $Y$ divides the prime $\ell$, so it is 1 or $\ell$. If $Y \neq \mathcal{O}$ the order is $\ell$; then $8Y = \mathcal{O}$ would force $\ell \mid 8$ — impossible since $\ell > 8$ (it is $\approx 2^{252}$). This is where both primality (order is 1 or $\ell$, nothing between) and size come in; $\gcd(8,\ell) = 1$ is the compact way to say "multiplying by 8 is invertible on the $\mathbb{Z}_\ell$ part." (c) The attacker changes the point $X$ (so: byte-level equality checks, hashes of the key, uniqueness assumptions *can* be affected — real protocols have been bitten) but provably cannot change $8X$, hence cannot affect the truth value of any cofactored verification equation. The formal apex certificate inherits exactly this robustness, and the "8" in its statement is this exercise, immortalized.

### Solution 12.3.

*Pathway.* Close the book. Write two columns: *assumes* / *establishes*. Then open the apex section and diff.

*Answer.* The list your memory should reproduce — *assumes*: (1) SHA-512 behaves as an ideal hash (axiomatized by design, in the trusted-base ledger); (2) the untranslatable SIMD point-multiplication backends meet their stated specs (documented, per fork); (3) the three standard Lean axioms; (4) the extraction pipeline preserves meaning (one tool, pinned versions). *Establishes*: for every input in

the bounds discipline, the extracted verification routine returns true *iff* $8sB = 8R + 8\,H(R, A, m)\,A$ in the curve group — with field arithmetic, group law, scalar arithmetic, and encoding each carried by its own kernel-checked layer below. If your two columns match this, you can audit a verification paper's abstract in ninety seconds — which was the promise on the book's cover, kept.

**⋈ Checkpoint**

The book's ending is a beginning, so the final checkpoint is prospective: you should be able to (1) state what each pyramid layer claims and which denotation it rides on; (2) explain to a security engineer why completeness of the Edwards law matters to *code*; (3) locate the current frontier and say precisely why it is hard; and (4) name the next proof *you* intend to write. The authors of the companion repositories left the scaffolding up on purpose.

# Appendix A

# The Pen-and-Paper Toolkit

Every worked example in this book leaned on a small set of hand-computation techniques. This appendix collects them as recipe cards — the reference you will reach for when auditing numbers in the wild, where there is no chapter telling you which trick applies. Each card ends with a thirty-second drill; answers close the appendix.

## A.1 Card 1: powers of two into powers of ten

The single most-used conversion in cryptographic sanity-checking:

$$\log_{10} 2 \approx 0.30103 \qquad \Longrightarrow \qquad 2^n \approx 10^{\,0.301\,n}.$$

Anchor points worth memorizing outright: $2^{10} \approx 10^3$ (the "kibi ≈ kilo" fact, 2.4% off), $2^{20} \approx 10^6$, $2^{64} \approx 1.8 \times 10^{19}$, $2^{128} \approx 3.4 \times 10^{38}$, $2^{256} \approx 1.2 \times 10^{77}$. The last one explains the phrase "77-digit prime" used throughout this book: $255 \times 0.301 = 76.8$, so $2^{255}$ has 77 digits. For headline comparisons, two reference magnitudes: atoms in the observable universe $\sim 10^{80}$; age of the universe $\sim 4 \times 10^{17}$ seconds; and the accidental gem 1 year $\approx \pi \times 10^7$ seconds.

*Drill 1.* How many decimal digits does $\ell \approx 2^{252}$ have? And is $2^{130}$ more or fewer than the atoms in a kilogram of silicon ($\sim 2 \times 10^{25}$)?

## A.2 Card 2: reducing big powers with the modulus identity

To reduce $2^N$ modulo $p = 2^{255} - 19$: split off multiples of 255 in the exponent and replace each $2^{255}$ by 19:

$$2^N = 2^{255q+r} \equiv 19^{\,q} \cdot 2^{\,r} \pmod{p}.$$

The same works in any system with an identity $R \equiv c$: powers of the radix fold down by repeated substitution. Keep the numbers honest by reducing intermediate results whenever they exceed the modulus.

*Drill 2.* Reduce $2^{511}$ mod $(2^{255} - 19)$ to a product of small numbers. (Split $511 = 2 \cdot 255 + 1$.)

## A.3 Card 3: small-field residues via little Fermat

To find $M$ mod $q$ for gigantic $M = 2^N$ and small prime $q$: reduce the *exponent* mod $q-1$ (Fermat: $2^{q-1} \equiv 1 \bmod q$), then compute the small remaining power. Used in Chapter 6's worked example to get $p$ mod 19 from $255 = 14 \cdot 18 + 3$.

*Drill 3.* Compute $p$ mod 7 for $p = 2^{255} - 19$. ($2^3 \equiv 1 \bmod 7$, so reduce the exponent mod 3.)

## A.4 Card 4: extended Euclid, back-substitution form

To invert $a$ modulo $n$: run remainders downward $(n, a, r_1, r_2, \ldots, 1)$, then walk back up expressing 1 in terms of each pair. The Chapter 6 worked example is the template; the discipline that prevents errors is to write each line as $1 = (\text{coeff}) \cdot x + (\text{coeff}) \cdot y$ with both terms visible before substituting the next level. Sanity-check the final line by multiplying it out — thirty seconds that catches ninety percent of slips.

*Drill 4.* Find $7^{-1}$ mod 15 by back-substitution (two lines), and check by multiplication.

## A.5 Card 5: square-and-multiply, tabular form

To compute $w^e$ mod $n$ by hand: build the squares $w, w^2, w^4, w^8, \ldots$ (each line: square the previous, reduce), then multiply together the squares selected by the binary digits of $e$. Bookkeeping form used in Chapter 7's certificate check: one column of exponents, one column of reduced values; never carry an unreduced number to the next line. Cost: $\lfloor \log_2 e \rfloor$ squarings plus (ones in $e$'s binary) $-1$ multiplies — knowing the cost formula lets you *decline* hand computations that are too big, which is itself a toolkit skill.

*Drill 5.* Compute $3^{22}$ mod 23 by the tabular method ($22 = 16 + 4 + 2$), and interpret the answer via Fermat.

## A.6 Card 6: the headroom audit

For any limb design, three lines locate the overflow cliff:

headroom = word bits$-$radix bits; add budget = $2^{\text{headroom}}$; mul check: (limb count)$\cdot 2^{2\cdot\text{bound bits}} \overset{?}{<} 2^{\text{wide wo}}$

Run all three whenever anyone shows you a limb representation — Chapter 1 (budget), Chapter 5 (mul check), and Chapter 10's $16p$ audit are all instances.

*Drill 6.* A proposal uses radix-29 limbs in 32-bit words, nine limbs, 64-bit wide multiplies, bound $2^{31}$ after growth. Does the mul check pass?

## A.7 Card 7: quadratic residues by the ladder

Is $a$ a square modulo an odd prime $p$? Euler's criterion decides:

$$a^{(p-1)/2} \equiv \begin{cases} +1 \pmod p & a \text{ is a square,} \\ -1 \pmod p & a \text{ is not.} \end{cases}$$

Compute the power by Card 5's ladder. Small-scale sanity check first — 2 modulo 7: $2^3 = 8 \equiv 1$, so 2 *is* a square mod 7 (indeed $3^2 = 9 \equiv 2$; finding the root is genuinely harder than testing — another find/check asymmetry). This card is load-bearing in Chapter 12: completeness of the Ed25519 addition law rests on "$d$ is not a square," one Euler evaluation; and the twist's legitimacy rests on "$-1$ *is* a square mod $p$," which Euler settles by pure exponent parity: $(-1)^{(p-1)/2} = +1$ exactly when $(p-1)/2$ is even, i.e. $p \equiv 1 \pmod 4$ — true for $2^{255} - 19$ (check: $2^{255} \equiv 0 \pmod 4$, so $p \equiv -19 \equiv -3 \equiv 1 \pmod 4$ ✓, no ladder required).

*Drill 7a.* Is 3 a square mod 11? ($3^5 \bmod 11$ by ladder; then find the root or trust the sign.)

## A.8 Card 8: the substitution test (specs)

Given a claimed specification: substitute the worst implementation of the right type ("return zero," "return the input," "ignore arguments") and evaluate the statement by hand. If the adversary passes, the spec is not about correctness. Refinement for equations: if the implementation appears identically on *both* sides, it cancels and was never tested (Chapter 11, exercise 11.5).

*Drill 8.* Does the spec $\forall ab,\ [\![\texttt{mul}(a,b)]\!] = [\![\texttt{mul}(b,a)]\!]$ survive the substitution test as a *correctness* spec for multiplication?

## Drill answers

**Solution Drill 1.** $252 \times 0.301 = 75.9$: $\ell$ has 76 digits. $2^{130} \approx 10^{39.1}$, vastly *more* than $2 \times 10^{25}$ atoms — brute force over 130-bit keys loses to physics, not just to patience.

**Solution Drill 2.** $2^{511} = (2^{255})^2 \cdot 2 \equiv 19^2 \cdot 2 = 722 \pmod{2^{255} - 19}$.

**Solution Drill 3.** $255 = 3 \cdot 85$, so $2^{255} = (2^3)^{85} \equiv 1^{85} = 1 \pmod 7$; and $19 = 2 \cdot 7 + 5 \equiv 5$, so $p \equiv 1 - 5 \equiv -4 \equiv 3 \pmod 7$.

**Solution Drill 4.** $15 = 2 \cdot 7 + 1$, so $1 = 15 - 2 \cdot 7$ directly: $7^{-1} \equiv -2 \equiv 13 \pmod{15}$. Check: $7 \cdot 13 = 91 = 6 \cdot 15 + 1$ ✓.

**Solution Drill 5.** Squares mod 23: $3^2 = 9$; $3^4 = 81 \equiv 12$; $3^8 = 144 \equiv 6$; $3^{16} = 36 \equiv 13$. Then $3^{22} = 3^{16} \cdot 3^4 \cdot 3^2 = 13 \cdot 12 \cdot 9$: $13 \cdot 12 = 156 \equiv 156 - 138 = 18$; $18 \cdot 9 = 162 \equiv 162 - 161 = 1$. So $3^{22} \equiv 1 \pmod{23}$ — which is Fermat's little theorem announcing itself ($22 = 23 - 1$), and incidentally the first condition of a Pratt witness check for 23.

**Solution Drill 6.** Products: $2^{31} \cdot 2^{31} = 2^{62}$; nine terms: $9 \cdot 2^{62} > 2^3 \cdot 2^{62} = 2^{65} > 2^{64}$ — **fails** by at least one bit. (Exactly: $9 \cdot 2^{62} = 2^{62} \cdot 9 > 2^{64} \Leftrightarrow 9 > 4$ ✓ fails.) The design must either bound limbs

more tightly than $2^{31}$, reduce more often, or accumulate in wider precision — and now you can say so in a design review with three lines of arithmetic as your citation.

**Solution Drill 7a.** $3^5 = 243 = 22 \cdot 11 + 1 \equiv 1 \pmod{11}$ (ladder: $3^2 = 9$, $3^4 = 81 \equiv 4$, $3^5 = 4 \cdot 3 = 12 \equiv 1$) — so 3 *is* a square mod 11; indeed $5^2 = 25 \equiv 3$.

**Solution Drill 8.** No. Substitute $\texttt{mul}_c(a, b) :=$ "return the constant array $c$": both sides become $[\![c]\!]$ — the adversary passes. Commutativity-of-the-implementation is a *symmetry* spec; symmetric garbage satisfies it. (Real correctness needs the other side of the square: $[\![a]\!] \cdot [\![b]\!]$, a quantity the implementation cannot influence.)

# Appendix B

# Guided Walkthroughs of the Exercise Files

The `exercises/` folder contains eight Lean files with `sorry` holes; `solutions/` contains their completed twins (every one compiled, with no `sorry`, against the pinned toolchain). This appendix is the middle path between the two: for every hole, the *pathway* — what to look at, what to try, where you will probably get stuck and why — and then the resolution. Use it when a hole has genuinely defeated you; the file order follows the book.

## B.1 Ch02.lean — definitions by recursion

**`mul` (the Try It exercise).** Pathway: mirror `add`'s skeleton — the answer to "`mul m` of *how many*?" is decided by the second argument's constructor. Base: $m \cdot 0 = 0$ (not $m$! — evaluate your candidate on `mul 3 0` before believing it). Step: $m \cdot (n+1) = m \cdot n + m$, i.e. `mul m n + m`, using only `+` as instructed. Common stall: writing `mul m n + n` — catches the wrong variable; the check `mul 6 7 = 42` exposes it instantly ($6 \cdot 7$ would come out as 49... work out why: 7 added 7 times).

**`pow`.** Pathway: same skeleton, one level up the operation ladder — `pow b 0 = 1` (the empty *product*), step multiplies by `b`. The deliberate lesson: `add`, `mul`, `pow` are the same two-line shape with different base cases/combinators — iteration all the way up. **`fib`.** The two-base-case pattern `| 0`, `| 1`, `| n + 2` is new; Lean happily recurses on *both* predecessors because the pattern `n + 2` makes both `n + 1` and `n` structurally smaller. **`Rational.add`.** Direct transcription of $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ with anonymous constructor syntax ⟨`num, den`⟩; the type-level lesson (what *can't* be enforced) is in the chapter solutions.

## B.2 Ch03.lean — term-mode logic

The file's discipline (no tactics) makes every hole a smallish program. The reliable procedure for all six: (1) unfold the connectives into arrow/pair/tagged-union shape; (2) write `fun` for every arrow in the goal; (3) inside, build the result with `And.intro` / `Or.inl` / `Or.inr` / projections / application. Where people actually stall: `or_swap` — the urge is to project (`h.1`) out of a disjunction, which is a type error since only one side exists; the resolution is `match h with | Or.inl p => ... | Or.inr q => ...`, and the arms must *swap the tags*. And `not_not_intro` — stare at the unfolded type `P → (P → False) → False` until you see it is *modus ponens with the arguments flipped*: `fun p f => f p`. If a

hole type-checks but feels like luck, re-run the Chapter 3 worked example's hand-check on your own term — that skill is the file's real deliverable.

## B.3 Ch04.lean — tactic mode, own addition

The file defines `myAdd` (recursion on the second argument) so the standard library cannot spoil the induction. **`and_swap`**: warm-up per the chapter — `intro h; constructor; · exact h.2; · exact h.1`. **`zero_myAdd`**: the worked example's board trace, executed: `induction n with`; zero case `rfl`; succ case `simp only [myAdd]; rw [ih]`. The stall here is almost always *unfolding*: if the goal shows `myAdd 0 (k + 1)` and nothing seems to apply, remember `simp only [myAdd]` unfolds one definitional layer — then the `ih` rewrite site becomes visible. **`succ_myAdd`**: same shape exactly; prove it *without* looking at the previous one, as calibration. **`myAdd_comm`**: induction on `n`, then each case is a chain of the two lemmas you just proved plus `ih`; if your rewrite chain grows past four steps, you are fighting the wrong induction variable. **`calc_repair`**: the broken step is the last (`6 * b`); fix the arithmetic, and note *how* you found it — the error message pointed at the exact line whose sides differ, which is the maintenance experience of Chapter 10 in one line.

## B.4 Ch05.lean — ten goals, right tool each

The file rejects overkill by intent; the classification procedure is Chapter 5's table. G1, G2, G6, G10: linear (constants multiply variables) → `omega`. G3, G8: ring identities → `ring`. G4, G7: concrete numerals → `norm_num`. G5: finite decidable → `decide`. G9 is the file's teeth — the two-step pattern: `have hab : a * b < 100 * 100 := Nat.mul_lt_mul" ha hb` then `omega`. The realistic stall is finding the lemma name: use `exact?` on the `have`'s goal and let the library search work for you — that, too, is a skill the file is deliberately installing.

## B.5 Ch06.lean — clock worlds

The `#evals` at the top are answered in the solutions file — predict before running, especially `(4⁻¹ : ZMod 12)`, whose junk answer ($1$ — *not* an inverse; check $4 \cdot 1$) teaches the totality convention. **6.A**: `ring` — the modulus is irrelevant to a ring identity. **6.B**: `decide` — twelve cases. **6.C**: the same `decide` *fails* at modulus 13; the witness it implicitly found is $x = 10$ (`#eval (4⁻¹ : ZMod 13)`). **6.D**: two stalls by design — the `Fact (Nat.Prime 11)` instance must exist before the Fermat lemma applies (the file provides it; understand why it is needed — typeclass search cannot prove primality, it can only *look up* registered facts), and the library lemma gives `a ^ (11 - 1) = 1`, which `simpa` normalizes to `a ^ 10 = 1`. **6.E**: the reduction identity — the solution routes through `ZMod.natCast_self p` ($p$ becomes 0 on its own clock) plus a `norm_num`-checked equation $p + 19 = 2^{255}$; if you tried `decide`, you have discovered the kernel-cost lesson of Chapter 7 experimentally.

## B.6 Ch07.lean — the certificate ladder

**7.A**: `decide` — fast at two digits. **7.B**: try `decide`, feel the pause, switch to `norm_num`; both are honest, one is wiser. **7.C**: `norm_num` *after* converting the goal to a literal with `show Nat.Prime 2147483647`

— the extension pattern-matches on numerals, a real-tool wrinkle worth having met in a safe place. **7.D**: the missing `#evals` are `(2:ZMod 13)^6` (expect 12) and `(2:ZMod 13)^4` (expect 3) — hand-check against Card 5 of Appendix A. **7.E**: the `powModAux` hole wants exactly the recipe in its comment — `if e = 0 then acc else powModAux fuel (b*b % m) (e/2) (if e % 2 = 1 then acc*b % m else acc)` — mind the argument order matching the signature. The sanity `#evals` catch the two classic slips (squaring the accumulator; forgetting the final odd-bit multiply). Then the finale prints `1` at 77 digits in milliseconds, and you have *built* the tool the whole chapter was about.

## B.7 Ch09.lean — the miniature bridge

**9.A** `denote`: one line — `(a.1 : ZMod 15) + 4 * (a.2 : ZMod 15)`; the casts matter (the addition must happen *on the clock face*, not in `Nat`). **9.B** `add_spec`: follow the Interlude, which is precisely this proof on paper. Bounds clause: `simp only [add]; omega`. Value clause — the one genuine difficulty in the file — state the exact integer identity with its correction term: `have key : (add a b).1 + 4 * (add a b).2 + 15 * ((a.2 + b.2 + (a.1 + b.1) / 4) / 4) = (a.1 + 4*a.2) + (b.1 + 4*b.2)`, discharge with `simp only [add]; omega`, then cast (`push_cast`), kill the modulus (`rw [show (15 : ZMod 15) = 0 by decide]`), and close with `linear_combination this`. If your `key` won't close, your correction term is wrong — recompute $c_2$ on paper (Interlude Step 2); `omega`'s refusal is, as always, a counterexample pointing at the boundary. **9.C** `mulVal_spec`: same cast-and-kill scaffold, but the integer identity is pure algebra — `mulVal a b + 15 * (a.2 * b.2) = (a.1 + 4*a.2) * (b.1 + 4*b.2)` by `ring` — and *no bounds hypotheses are needed*, a fact worth noticing (denotation does not care about digit discipline; only machine words do).

## B.8 Ch12.lean — graduation

**Task 1** is supposed to fail: run the 9.B proof shape against `add'` and watch `omega` refuse the `key` identity — because it is false. **Task 2**: extract the counterexample from the refusal: the divergence is $\lfloor s_0/5 \rfloor \neq \lfloor s_0/4 \rfloor$, which within bounds happens only at $s_0 = 4$; realize it with $a = (2, 0)$, $b = (2, 0)$ and confirm `denote (add' (2,0) (2,0)) = 0 ≠ 4`. **Task 3**: change `s0 / 5` to `s0 / 4`; the 9.B proof now goes through verbatim — run it and read the moral out loud: *the proof failed exactly while the code was wrong and succeeded exactly when it was fixed.* **Task 4** wants the Chapter 12 reflection in your own words; the solutions file has a model paragraph, but yours counts double if it mentions which *specific* tests you would have had to write to catch $s_0 = 4$ by accident — and how many you actually wrote.

That is the last hole in the last file. If you worked them all: the companion repositories' open scalar-layer lemmas are shaped exactly like 9.B — bigger constants, same bones — and the `CONTRIBUTING` notes there will treat you as what you now are: someone who has done this before.

# Appendix C

# A Guided Tour of the Real Repositories

The companion projects are working code, laid out for auditors rather than tourists. This appendix is the tourist map: what lives where, what to read first, and how to run the machinery yourself. Everything below names `dalek-ed25519-verified`; the other three ed25519 forks are structured identically, and `pasta-pallas-verified` differs only where Montgomery form demands it.

## C.1 The floor plan

| Path | What it is, and the rule that governs it |
|---|---|
| `README.md` | the claims: what is proven, what is frontier — the honest-boundary statements of Chapter 11 |
| `TRUSTED-BASE.md` | the ledger: every assumption, each with its reason — read this *before* being impressed by anything |
| `verification/extract.sh` | extraction roots for the field layer (Charon → LLBC → Aeneas); `extract-scalar.sh` likewise for scalars |
| `verification/gen/` | the extracted model. **Never hand-edited** (Chapter 8); regenerated or left alone |
| `verification/Proofs/` | the human-written theorems — denotations, bounds lemmas, layer certificates. **No `axiom` may appear here** (the check script enforces it) |
| `verification/check.sh` | THE button: recompiles every shipped file in dependency order and axiom-audits every certificate; if a file is in the repo, this script checks it |
| `verification/lean-guard` | the resource-capped `lean` wrapper every compile routes through — the postmortem-hardened tooling mentioned in Chapters 5 and 10 |

## C.2 A reading order that works

1. `TRUSTED-BASE.md` (five minutes). Count the entries; for each, ask the Chapter 11 question — "declared debt or smuggled axiom?" — and notice each has its justification attached.
2. **The denotation** (in `Proofs/`, near the top of the field spec file). One definition; confirm it is Chapter 9's radix-51 sum, mapping the *extracted* type.

3. **One two-clause spec end to end** — `sub` is the best first read: bounds hypotheses, `.ok` clause, bounds propagation, value equation, and inside the proof, the $16p$ constant you audited in Chapter 10's worked example.
4. **The certificate** — the conjunction theorem (`fieldImplementation`) and its `#print axioms` line. This is the artifact all the marketing language ultimately refers to; note how unglamorous it looks.
5. **The frontier** — the scalar layer's in-progress files, clearly marked. Read one open lemma statement and recognize its shape (it is 9.B with bigger constants). This is where Chapter 12's invitation points.

## C.3 Running the machinery

With the toolchain installed (the repos pin exact versions — Chapter 8's reproducibility discipline):

```
$ ./verification/check.sh
=== Phase 1: stub audit ===
  clean: no trivial stubs, no True targets, no axioms outside gen/
=== Phase 2: compile ===
  [gen] CurveField/Funs ... ok
  [proofs] FieldSpec ... ok        (no 'sorry' warnings -- enforced)
=== Phase 3: axiom audit ===
  fieldImplementation: [propext, Classical.choice, Quot.sound]  OK
ALL CHECKS PASSED
```

Three details make this script worth imitating in your own projects. *Phase 1 runs before compilation*: trivial-stub patterns (`by trivial` specs, `True` targets) and `axiom` declarations under `Proofs/` are cheap textual gates — the mechanical half of the Chapter 11 field guide, automated. *Phase 2 treats warnings as failures*: the string `uses 'sorry'` in compiler output fails the build — warnings, unlike source text, cannot be hidden in comments. *Phase 3 is the one-command audit*, run on every certificate, every time, so the axiom-clean property is continuously enforced rather than occasionally asserted.

## C.4 The control repository

`formal-verification-control` is the method distilled — written for the next person (or the next automated agent) to extend the pyramid without relearning its lessons:

- `INVARIANTS.md` — the non-negotiables (honesty, safety, rigor); the source of house rules this book has been quoting.
- `TERRAIN.md` — the map: pipeline, toolchain, layer status, representation costs.
- `METHOD.md` — ways of thinking that worked, stated as practices rather than commandments.
- `FAILURES.md` — mapped dead ends *with their tells*: the memory-exhausting tactic patterns, the extraction scopes that drag in the world, the kernel-capacity wall. Chapter 10 told two of these stories; the file has the rest, and reading failure maps before starting work is the cheapest experience money can't buy.

A closing observation to carry out of the tour: nothing in these repositories asks to be trusted. The claims are in the READMEs, the assumptions in the ledgers, the checks in a script anyone can run, the axioms in a one-command audit. That shape — *auditability as the default posture* — is the real deliverable of the whole verification enterprise, and the standard this book hopes you now hold everything else to.

# Glossary

**Axiom-clean.** Of a theorem: `#print axioms` reports exactly Lean's standard trio `[propext, Classical.choice, Quot.sound]` and nothing else. The gold standard for shipped certificates (Chapter 11).

**Bounds invariant.** A predicate limiting how large limbs may grow (e.g. every limb $< 2^{54}$), maintained across operations so that machine arithmetic never overflows. One of the two clauses of every operation spec (Chapters 8–9).

**Carry.** Value moved from one limb position to the next when a limb exceeds its radix. Delayed ("lazy") carries are the central performance trick of fast field arithmetic and the habitat of its characteristic bugs (Chapter 1; Interlude).

**Certificate.** Data that makes a fact cheap to *check* regardless of how expensive it was to *find*: a Pratt witness tree for primality, a proof object for a theorem (Chapter 7).

**Charon / Aeneas.** The two-stage extraction pipeline: Charon compiles Rust to the LLBC intermediate representation; Aeneas translates LLBC into pure Lean definitions (Chapter 8).

**Cofactor.** The factor 8 in the Ed25519 group order $8\ell$; multiplying by it annihilates the small-torsion component of any point, which is why the verification equation reads $8sB = 8R + 8kA$ (Chapter 12).

**Commuting square.** The diagram — machine operation along the top, ideal operation along the bottom, denotation down the sides — whose closure *is* implementation correctness (Chapter 9).

**Complete (addition law).** An addition formula with no exceptional cases: valid for every pair of points, including doubling and identity. Ed25519's Edwards law is complete because $d$ is a non-square (Chapter 12).

**Decision procedure.** An algorithm that settles *every* statement in a defined logical fragment — `omega` for linear arithmetic, `decide` for finite computations, `ring` for ring identities. Failure on an in-fragment goal means the goal is false (Chapter 5).

**Definitional equality.** Two terms being identical after the kernel computes (unfolds definitions, reduces recursion). What `rfl` checks; blocked by opaque variables in recursion position (Chapter 3).

**Denotation.** The function $[\![\cdot]\!]$ mapping a machine representation (limb array) to the mathematical value it *means* (an element of $\mathbb{F}_p$). The bridge on which all correctness statements stand (Chapter 9).

**Euler's criterion.** $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$ decides whether $a$ is a square modulo the odd prime $p$ ($+1$: square; $-1$: non-square). Settles both completeness facts of Chapter 12 (toolkit Card 7).

**Extraction.** Mechanical translation of source code (Rust) into a proof assistant's language via

Charon/Aeneas, producing the *model* — the artifact actually verified, never hand-edited (Chapter 8).

**Fermat's little theorem.** $a^{p-1} \equiv 1 \pmod{p}$ for prime $p$ and $a \not\equiv 0$; hence $a^{p-2} = a^{-1}$, the identity behind the verified inversion chain (Chapters 6, 10).

**Find/check asymmetry.** The gap between the cost of discovering a fact and the cost of verifying a certificate for it — the engine of Pratt certificates, proof kernels, and (in disguise) the P-vs-NP question (Chapter 7).

**Hasse bound.** An elliptic curve over $\mathbb{F}_p$ has $p+1-t$ points with $|t| \le 2\sqrt{p}$; the thirty-second sanity check for any claimed group order (Chapter 12).

**Fold.** Reducing an overflow of the representation (weight $2^{255}$ and above) back into range using the modulus identity $2^{255} \equiv 19$; costs exactly one multiple of $p$ per unit folded (Chapter 9; Interlude).

**Goal state.** The proof assistant's board: hypotheses above the turnstile $\vdash$, obligation below. Reading it is the core tactic skill (Chapter 4).

**Headroom.** Bits of slack between a limb's payload (e.g. 51 bits) and its machine word (64 bits); the budget lazy carries spend (Chapter 1).

**Inductive type.** A type defined by listing its constructors exhaustively (`Nat`: `zero` and `succ`). Grants both pattern matching and the induction principle (Chapters 2, 4).

**Kernel.** The small, paranoid core of a proof assistant that re-checks every proof object against a fixed rule set; the only component whose correctness soundness depends on (Chapter 1).

**Limb.** One machine word of a multi-word big-number representation; Ed25519 field elements use five 51-bit limbs in 64-bit words (Chapters 1, 9).

**Model.** The extracted Lean rendition of the source code, living in `gen/`; the object theorems quantify over (Chapter 8).

**Montgomery form.** Representing $x$ as $x \cdot R \bmod p$ (typically $R = 2^{256}$) to make post-multiplication reduction cheap; absorbed by adjusting the denotation (Chapter 9).

**Pratt witness.** An element $w$ with $w^{p-1} \equiv 1$ and $w^{(p-1)/q} \not\equiv 1$ for every prime $q \mid p-1$; its existence certifies $p$ prime, given certificates for the $q$'s (Chapter 7).

**Radix.** The base of a limb representation ($2^{51}$ for the dalek field, 4 for this book's toy system).

**Specification (spec).** The precise statement a program is proven to satisfy. The two-clause shape for arithmetic: bounds propagation plus value equation. A proof is only as good as its spec (Chapters 9, 11).

**Substitution test.** Auditing a spec by substituting an adversarial implementation and checking whether the statement notices; detects trivial specs no tool can flag (Chapter 11).

**Tactic.** A command in Lean's interactive proof mode that transforms the goal state (`intro`, `rw`, `induction`, `omega`, ...), assembling a proof object behind the scenes (Chapter 4).

**Torsion.** The small-order component of a curve point (order dividing the cofactor); killed by multiplying by 8, hence invisible to cofactored verification (Chapter 12).

**Trusted base.** Everything a verification result assumes rather than proves: the kernel, the extraction tool, declared axioms (SHA-512, untranslatable backends). Honest projects keep it small, documented, and machine-visible (Chapters 8, 11).

**Two-clause spec.** This book's name for the standard operation theorem: *(1)* the operation succeeds and its output satisfies the (possibly widened) bounds invariant; *(2)* the output's denotation equals the ideal result (Chapter 9; Interlude).