



VANDERBILT UNIVERSITY
College of Connected Computing

Day 3 — Theorem Proving with Lean 4 (and AI)

FMAIV: Formal Methods & AI-Assisted Verification

Taylor Johnson — Associate Professor of Computer Science, Computer Engineering
& Electrical Engineering; Associate Dean for Graduate Education, College of
Connected Computing · taylor.johnson@vanderbilt.edu · taylortjohnson.com

Ben Wooding — Postdoctoral Scholar, Institute for Software Integrated Systems ·
ben.wooding@vanderbilt.edu · woodingben.com

Vanderbilt University

Day 3 of 4

Day 3 — Theorem Proving

Lean 4, inductive invariants, and AI-assisted proof

Run it live (no install): Codespaces (full toolset) or Colab — today's notebook
`03_day3_theorem_proving`. All materials: <https://github.com/ttj/fmaiv>

Where we are

Day 2: nuXmv proved $AG (x \leq 10)$ by exploring **all reachable states** of a finite system.

That breaks down when:

- the state space is **infinite** (unbounded integers, reals, lists), or
- the system is **parametric** ("mutual exclusion for *all* n processes"), or
- the property is **mathematical** ("there are infinitely many primes").

Today: prove properties by induction, with no enumeration — so unbounded and parametric are fine.

Today's roadmap

Block	Topic
L1	Why theorem proving; what Lean 4 is; the current moment
L2	Lean by example: tactics, library search, the counter, inductive invariants
L3	AI in the proof loop; Lean's industrial role

Running example: the same counter, now a Lean `TransitionSystem`.

Learning objectives for Day 3

By the end of today you will be able to:

- Explain why theorem proving complements model checking, and what each is for.
- Read and write basic Lean 4 — terms, types, `def`, `theorem`, core tactics.
- State a transition system and an invariant in Lean, and prove the invariant by the **inductive-invariant + strengthening** pattern.
- Use an AI assistant to draft proof steps, and recognize when it is wrong.

L1 — Why proving, and what Lean is

What a theorem prover is

A **theorem prover** (proof assistant) is software that:

- lets you state definitions and theorems in a formal language, and
- checks a proof against a tiny, trusted **kernel** — every step is verified.

You write the proof (often interactively); the machine guarantees it is correct. No step is taken on faith.

A very short history

Automath (1967) → LCF (1970s, "tactics") → HOL, Coq, Isabelle, Agda → **Lean** (2013; the Lean 4 rewrite began ~2021, first stable release 2023).

The field converged on **dependent type theory**: a single language where

- propositions *are* types, and
- proofs *are* terms (programs) of those types.

Propositions as types (Curry–Howard)

```
#check (5 : Nat)      -- 5 has type Nat
#check Nat            -- Nat has type Type
#check (5 = 5)        -- 5 = 5 has type Prop (a proposition is a type)
#check (rfl : 5 = 5)  -- rfl is a *term* (proof) of that type
```

Read `e : T` as “`e` has type `T`”. `Nat` = the naturals 0,1,2,...; `Type` = the type of ordinary data types; `Prop` = the type of propositions (statements that can be true or false); `#check` just prints a term’s type.

To prove `P` is to **construct a term of type** `P`. The kernel type-checks the term.

Everything has a type — even types

In Lean, **every** expression has a type — and a type is itself an expression, so it has a type too:

```
#check (5 : Nat)      -- 5      : Nat
#check Nat            -- Nat    : Type    (a data type lives in `Type`)
#check Type           -- Type   : Type 1  (and `Type 1 : Type 2`, ...)
#check (5 = 5)        -- Prop   (a proposition)
#check Prop           -- Prop   : Type    (propositions live in `Prop`)
```

So there are **three layers**: a *value* (`5`), its *type* (`Nat`), and the type's *type* (`Type`).
Reading bottom-up: `5 : Nat : Type`.

Sort , Prop , Type — the universes

The types-of-types are called **universes**. Two matter to us:

Universe	Holds	Example members
Prop	propositions (things to prove)	<code>5 = 5</code> , <code>x ≤ 10</code> , <code>p ∧ q</code>
Type (= Type 0)	ordinary data types	<code>Nat</code> , <code>Bool</code> , <code>CounterState</code>

`Type 0 : Type 1 : Type 2 : ...` is an infinite tower (so nothing contains itself).

`Sort` is the umbrella word covering both (`Prop = Sort 0` , `Type u = Sort (u+1)`).

(You will never need to manipulate universe levels in this course — skim this.)

The key asymmetry: in `Prop` , **all proofs of one proposition are interchangeable** — Lean's *definitional proof irrelevance* for `Prop` (any two proofs of the same proposition are treated as equal). We only care *that* it's proved, not *which* proof.

Function types, and *dependent* function types

A function type $A \rightarrow B$ is read “give me an A , get back a B ”:

```
#check (Nat.succ : Nat → Nat)      -- successor: a Nat in, a Nat out
#check (fun n => n + n : Nat → Nat) -- `fun x => ...` is an anonymous function
```

A **dependent** function type lets the *result type depend on the input value* — written $(n : A) \rightarrow B\ n$:

```
-- `∀ n, 0 + n = n` is a dependent function type:
--   give it a Nat `n`, get back a PROOF of `0 + n = n` (a different proposition per n)
#check (zero_add : ∀ n : Nat, 0 + n = n)
```

So $\forall$ (“for all”) is just a dependent function type whose outputs are **proofs**. That single idea is what makes “propositions as types” powerful enough for real math.

Curry–Howard, in full: the logic $\leftrightarrow$ types dictionary

Every logical connective is a **type constructor**; every proof is a **term** that builds or uses it:

Logic	Type	To prove it...	To use it...
$P \rightarrow Q$ (implies)	function $P \rightarrow Q$	write <code>fun (h : P) => ...</code>	apply it: <code>h pq</code>
$P \wedge Q$ (and)	pair $P \times Q$	<code><hp, hq></code> (give both)	project: <code>h.1</code> , <code>h.2</code>
$P \vee Q$ (or)	tagged union	<code>Or.inl hp</code> / <code>Or.inr hq</code>	<code>cases</code> on which side
$\exists x, P\ x$ (exists)	dependent pair	<code><w, hw></code> (witness + proof)	<code>obtain <w, hw> := h</code>
$\neg P$ (not)	$P \rightarrow \text{False}$	assume <code>P</code> , derive <code>False</code>	feed it a <code>P</code> to get <code>False</code>
<code>True</code>	one-element type	<code>trivial</code>	—
<code>False</code>	empty type	(impossible)	<code>absurd</code> — anything follows

A proof is a *program*; running the logic is *type-checking the program*.

A tiny term-mode proof of each connective

No tactics — just *build the term* the dictionary prescribes. The slogan “proofs **are** programs” is *literal*:

```
-- the simplest proof there is:  $P \rightarrow P$  is the identity function
example (p : Prop) : p → p :=
  fun hp ⇒ hp                                -- given a proof of p, hand it straight back

-- implication in general: still a function (this one ignores its 2nd input)
example (p q : Prop) : p → (q → p) :=
  fun hp ⇒ fun _ ⇒ hp                        -- given p (and anything), return the p

-- and: a pair (anonymous constructor { })
example (p q : Prop) (hp : p) (hq : q) : p ∧ q :=
  {hp, hq}

-- or: pick a side
example (p q : Prop) (hp : p) : p ∨ q :=
  Or.inl hp                                  -- "left" injection

-- exists: witness + proof
example : ∃ n : Nat, n + 1 = 4 :=
  {3, rfl}                                  -- witness 3; rfl proves 3 + 1 = 4
```

These compile with **no Mathlib** — the connectives are core Lean.

What the kernel actually checks

Everything reduces to **type-checking one term**:

- Tactics, `simp`, `omega`, library lemmas, even an AI — all only *produce* a candidate proof **term**.
- The **kernel** re-checks that term against the theorem's type, from scratch, with a small fixed set of rules.
- It is tiny, rarely changed, and the **only** thing you must trust.

So a 2-million-line library and an LLM are *equally untrusted*: whatever they emit, the kernel re-derives it.

What “re-check from scratch” means

A traditional math proof “conveys a message” — peer review checks the *ideas*, and the fiddly steps are “too cumbersome” to verify line by line. Lean is the opposite: **every** step must exist as a term, and the kernel re-derives all of them.

```
theorem two_plus_two : 2 + 2 = 4 := by
  rfl -- tactic mode: short to WRITE
-- but what the kernel RECEIVES and re-checks is a finished term:
#print two_plus_two
-- theorem two_plus_two : 2 + 2 = 4 := Eq.refl (2 + 2) (roughly)
```

You wrote one tactic; the kernel checked a concrete term (`Eq.refl (2 + 2)`) and confirmed both sides really compute to `4`. The tactic was just a *convenient way to produce* that term — it earns no trust of its own.

def, theorem, example

```
def double (n : Nat) : Nat := n + n          -- a definition
theorem double_zero : double 0 = 0 := rfl      -- a proof term
example : 2 + 2 = 4 := by rfl                 -- anonymous; `by` enters tactic mode
```

Definitions and theorems are the **same kind of thing**: named terms with a type.

- **rfl** = **reflexivity**: proves an equation that holds *by computation* — both sides reduce to the same value (here `double 0` → `0` and `2 + 2` → `4`).

rfl computes; induction reasons

rfl succeeds when both sides reduce to the *same* value — **definitional equality**:

```
example : 2 + 2 = 4 := by rfl          -- both sides compute to 4
example (n : Nat) : n + 0 = n := by rfl -- ✓ (+ recurses on its 2nd argument)
```

But $0 + n = n$ is **not** rfl — n is a variable on the recursing side, so it needs **induction**:

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero      ⇒ rfl          -- base: 0 + 0 = 0
  | succ k ih ⇒ rw [Nat.add_succ, ih] -- step: ih : 0 + k = k ⊢ 0 + (k+1) = k+1
```

Induction, the way you already know it

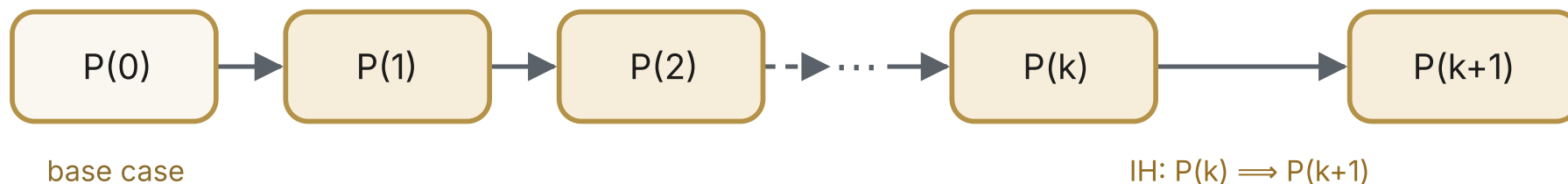
The pen-and-paper recipe for “**P** holds for *all* naturals **n**”:

- **Base case** — prove **P(0)**.
- **Inductive step** — *assume* **P(k)** (the **induction hypothesis**), prove **P(k+1)**.

Classic example: $0 + 1 + \dots + n = \frac{n(n+1)}{2}$.

- Base $n = 0$: left side = 0, right side = $\frac{0 \cdot 1}{2} = 0$. ✓
- Step: assume $0 + \dots + k = \frac{k(k+1)}{2}$ (the **IH**); then

$$0 + \dots + k + (k+1) = \underbrace{\frac{k(k+1)}{2}}_{\text{by IH}} + (k+1) = \frac{(k+1)(k+2)}{2}. \checkmark$$



The same induction, in Lean (from scratch)

```
theorem zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero =>
    -- BASE.    goal:  ⊢ 0 + 0 = 0
    rfl                                     -- both sides compute to 0
  | succ k ih =>
    -- STEP.    ih : 0 + k = k          (the induction hypothesis!)
    --          goal:  ⊢ 0 + (k + 1) = k + 1
    rw [Nat.add_succ]                    -- 0 + (k+1) ↔ (0 + k) + 1
    -- now goal:  ⊢ (0 + k) + 1 = k + 1
    rw [ih]                             -- rewrite 0 + k ↔ k using the IH
    -- now goal:  ⊢ k + 1 = k + 1      - true by rfl, rw closes it automatically
```

`induction n` splits into the two ways a `Nat` is built: `zero`, and `succ k` (= `k+1`) **plus** the hypothesis `ih` for `k`. `Nat.add_succ` and `rw` are **core** Lean — no Mathlib. These snippets (and the term-mode proofs earlier) are collected, compiled, and checked in `SlideExamples.lean`.

Why `succ k ih` has that shape

induction gives one case per **constructor** of the type, and a hypothesis for each recursive argument:

```
inductive Nat where
  | zero : Nat          -- → case `zero`  (no IH: nothing recursive)
  | succ (k : Nat) : Nat -- → case `succ k ih` (IH because `k : Nat` is recursive)
```

The induction principle is *generated from the type's definition*. That's the bridge to the counter: because **Reachable** is an **inductive** predicate (next block), it comes with its own induction principle — and that principle is exactly what proves an invariant.

Tactics: building the proof term

by opens a tactic block; each tactic transforms the **goal state** — the hypotheses, a turnstile $\vdash$, and the target to prove.

```
example (p q : Prop) (hp : p) (hpq : p → q) : q := by
  -- goal state:  hp : p,  hpq : p → q  ⊢  q
  exact hpq hp      -- applies hpq to hp — this is modus ponens; "no goals"
```

The InfoView (Lean's live panel) shows this state after each tactic, until "no goals." Read $\vdash q$ as "we still must prove q "; applying a proof of $p \rightarrow q$ to a proof of p is function application.

- Editor setup: install the **Lean 4 VS Code extension** (`leanprover.lean4`) for this live InfoView, and keep **Claude Code** open in the same window (integrated terminal or a side panel) — goal state and AI partner side by side.

Reading a goal state (the $\vdash$ turnstile)

The InfoView shows, at every moment: the **hypotheses** (what you know) above a line, the **turnstile** $\vdash$, and the **goal** (what's left to prove):

```
p q : Prop      -- two propositions are in scope
hp : p          -- we have a proof of p, named hp
hpq : p → q     -- we have a proof of p → q, named hpq
⊢ q            -- GOAL: we still must prove q
```

- Read $\vdash q$ as “**turnstile** q” = “under the hypotheses above, prove q .”
- Each tactic *rewrites this whole picture*. The proof is done when the goal becomes “**no goals.**”

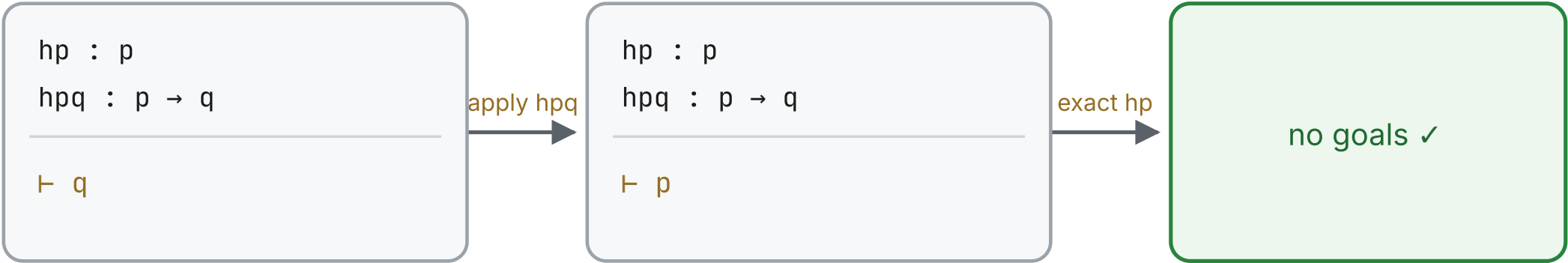
A tactic either **closes** the goal, **transforms** it, or **splits** it into several subgoals (each with its own $\vdash$).

Proof-state evolution, step by step

Watch the state change as each tactic fires:

```
example (p q : Prop) (hpq : p → q) (hp : p) : q := by
  -- ⊢ q
  apply hpq
  -- ⊢ p
  exact hp
  -- no goals
  ✓ done
```

After tactic	Goal	Why it changed
(start)	⊢ q	the theorem's claim
apply hpq	⊢ p	to get q via p → q , now prove p
exact hp	no goals	hp is exactly a proof of p



The current Lean moment (1)

- **Polynomial Freiman–Ruzsa conjecture** — informal proof by Gowers, Green, Manners & Tao (Nov 2023); formalized in Lean in **~3 weeks** (completed Dec 2023) by a 20+-person collaboration led by Tao with Yael Dillies and Bhavik Mehta.
- **Mathlib** — the community math library, **~2M lines**, ~800 contributors.

The current Lean moment (2)

- **AlphaProof + AlphaGeometry 2** (DeepMind, 2024) — **IMO 2024 silver-medal-equivalent** score (28/42); AlphaProof solved 3 problems with proofs generated and checked in Lean.
- **DeepSeek-Prover-V2** (671B, April 2025) — **88.9%** on miniF2F at a large pass@8192 budget (~82% at pass@32), the competition-Lean benchmark.

miniF2F-test SOTA (illustrative; sample budgets differ by year):
2022 ~35% → 2023 ~50% → 2024 ~65-70% → 2025 ~89%

Why mathematicians specifically

You already have the prerequisites CS undergrads spend two years acquiring: type discipline, induction, structural reasoning.

Three payoffs:

- **Research** — formalize your own results; catch your own errors.
- **Teaching** — Lean-graded problem sets that give real feedback.
- **Public good** — a checkable gate on AI-generated mathematics.

L1 recap

- A **proof assistant** checks every step against a tiny trusted **kernel**.
- Lean 4 is **dependent type theory**: propositions are types, proofs are terms.
- **Tactics** build the proof term interactively by transforming the goal.
- The moment is real: PFR, Mathlib (2M+ lines), AlphaProof, DeepSeek-Prover.



Break

We resume after the break with Lean by example — tactics, Mathlib, and the counter.

L2 — Lean by example: the counter

The workhorse tactics

Tactic	Use
<code>intro h</code>	move a hypothesis from goal into context
<code>exact e</code>	close the goal with term <code>e</code>
<code>rfl</code>	reflexivity (<code>a = a</code>)
<code>simp</code>	simplify with the simp-set
<code>omega</code>	decide linear arithmetic over <code>Nat</code> / <code>Int</code>
<code>cases h</code> / <code>rcases</code>	split a hypothesis (inductive, <code>∧</code> , <code>∨</code> , <code>∃</code>)
<code>by_cases h : P</code>	classical case split on <code>P</code>
<code>constructor</code>	prove an <code>∧</code> /structure by parts

More tactics you'll meet in the files

The example files use a few beyond the workhorse set — terse meanings:

Tactic	What it does
<code>induction h with \l ...</code>	prove a goal by cases on how <code>h</code> was built; the recursive case gets an induction hypothesis
<code>rw [h]</code>	rewrite the goal left-to-right using an equation <code>h : a = b</code>
<code>subst h</code>	use <code>h : a = b</code> to replace <code>a</code> by <code>b</code> everywhere
<code>refine {?_, ?_}</code>	like <code>exact</code> , but leave holes <code>?_</code> to fill as new goals
<code>obtain {a, b} := h</code>	destructure an <code>^</code> / <code>∃</code> / structure into named pieces
<code>decide</code>	close a <i>decidable</i> goal by computing the answer (e.g. <code>off ≠ on</code>)
<code>native_decide</code>	<code>decide</code> compiled to native code — faster, but adds a compiler-trust axiom
<code>absurd h hn</code>	derive anything from contradictory <code>h : P</code> and <code>hn : ¬P</code>
<code>t <;> s</code>	run tactic <code>s</code> on every subgoal that <code>t</code> produced

"Batteries-included": `decide` and `Decidable`

Some propositions can be **settled by computation** — Lean has an algorithm that returns `true` / `false`. Those are `Decidable`, and `decide` just runs the algorithm:

```
example : (2 + 2 = 4) := by decide      -- compute: yes
example : (3 < 10)    := by decide      -- compute: yes
example : ModeVal.off ≠ ModeVal.on := by decide  -- compare constructors: yes
example : ∀ n, n < 5 → n < 100 := by decide  -- x can't: range of n is infinite
```

- A goal is `decide`-able only when the search is **finite** (equality of concrete values, small bounded checks). "For all naturals" is *not* finite, so `decide` declines.
- `decide` is **core** Lean — no Mathlib. The kernel re-runs the computation, so it stays trusted (unlike `native_decide`, which compiles it and trusts the compiler).

omega : the linear-arithmetic workhorse

omega is a **decision procedure** for linear arithmetic over Nat / Int — it *decides* goals built from +, -, ≤, <, = and integer constants, with no lemmas from you:

```
example (x : Nat) (h : x < 10) : x + 1 ≤ 10 := by omega    -- ✓  
example (a b : Nat) (h : a + b = 10) (hb : b ≤ 3) : 7 ≤ a := by omega    -- ✓  
example (x : Nat) : x * x ≥ 0 := by omega    -- x `x*x` is NON-linear – omega declines
```

- **Linear** = variables only added/subtracted and compared, never multiplied together. $x + 1$, $2*x$ are fine; $x*y$ is not.
- omega is **core** Lean. It closes essentially *every* arithmetic leaf in the counter proof — e.g. from $x < 10$ conclude $x + 1 ≤ 10$.

Finding lemmas (library search)

```
example (a b : Nat) : a + b = b + a := by exact?    -- suggests Nat.add_comm
```

- `exact?` / `apply?` — **core Lean** tactics; they search every *imported* declaration for a lemma that closes/advances the goal.
- **Naming convention:** `Nat.add_comm`, `List.length_append` — namespace + what it says.
- `#loogle` / the Loogle website search by *type pattern* (a separate tool, not a tactic).

Our `CounterDemo` project is **Mathlib-free**, so `exact?` searches Lean's *core* library (where `Nat.add_comm` lives). Big math projects add **Mathlib** (~2M lines) for everything else.

Inductive types: data built one way at a time

An **inductive type** lists the *only* ways to construct its values (its **constructors**). You've already seen one — **Nat** is **zero** and **succ**:

```
inductive ModeVal | off | on      -- exactly two values; nothing else
inductive Bool    | false | true -- the same shape
```

The payoff: because the constructors are the *complete* list, Lean derives a matching **induction / recursion principle** automatically — “to handle any value, handle each constructor.” That principle is what **cases** and **induction** run on.

```
example (m : ModeVal) : m = .off ∨ m = .on := by
  cases m -- splits into the off case and the on case – no others possible
  · exact Or.inl rfl
  · exact Or.inr rfl
```

The recursor: an induction principle per type

For every inductive type, Lean generates a **recursor** (`.rec`) — the formal statement of its induction principle. You rarely call it directly; `induction` / `cases` use it internally.

```
#check @Nat.rec
-- to prove `motive n` for all n, supply:
--   • a proof of `motive 0`                (base case)
--   • for each k, `motive k → motive (k+1)` (step, given the IH)
```

So `induction n` is just `Nat.rec` with the base case and step filled in — *exactly* the base/step recipe from the warm-up. **Key idea:** define a thing inductively and you get its proof-by-induction principle **for free**.

A transition system, in Lean

From `CounterDemo/TransitionSystem.lean`:

```
structure TransitionSystem (State : Type) where
  init : State → Prop
  next : State → State → Prop

inductive Reachable {S} (ts : TransitionSystem S) : S → Prop where
| init   : ts.init s → Reachable ts s
| step   : Reachable ts s → ts.next s s' → Reachable ts s'

def Invariant {S} (ts) (P : S → Prop) : Prop :=
  ∀ s, Reachable ts s → P s
```

`structure` bundles named fields (a record); `inductive` defines a type by listing the only ways to build its values; `→` is “function / implies”; `∀ s` is “for all states `s`”. (Snippets elide some explicit type binders — see the file.)

`Reachable` is an **inductive predicate** — the Lean version of “states reachable in finitely many steps.”

The inductive-invariant method

```
def InductiveInvariant {S} (ts) (P : S → Prop) : Prop :=  
  (∀ s, ts.init s → P s) ∧ -- holds initially  
  (∀ s s', P s → ts.next s s' → P s') -- preserved by every step  
  
theorem inductive_invariant_holds :  
  InductiveInvariant ts P → Invariant ts P
```

If **P** holds initially and every step preserves it, then **P** holds on **all** reachable states — proved once, for any system.

Why those two obligations? `Reachable`'s induction

`Reachable` was defined with **two constructors** — so its induction principle has **two cases**, and they *are* the two proof obligations:

```
inductive Reachable (ts) : S → Prop where
| init : ts.init s → Reachable ts s           -- → base case
| step : Reachable ts s → ts.next s s' → Reachable ts s' -- → step case (with IH)
```

To prove $\forall s, \text{Reachable } ts \ s \rightarrow P \ s$, run `induction` on the `Reachable` proof:

- **init case** — `s` is initial $\implies$ show `P s`. (*This is the base obligation.*)
- **step case** — `s'` follows a step from a reachable `s`, with **IH** `P s` $\implies$ show `P s'`. (*This is the step obligation.*)

That's it — the method isn't a separate trick, it's the recursor of `Reachable`. Finite or infinite state space, the induction is the same.

The counter as a Lean system

From `CounterDemo/Counter.lean` (auto-translated from `counter.smv` by `smv2lean`):

```
inductive ModeVal | off | on
structure CounterState where
  mode : ModeVal ; press : Bool ; x : Nat

def CounterTS : TransitionSystem CounterState where
  init s := s.mode = .off ∧ s.x = 0
  next s s' := ∃ p', s'.press = p' ∧
    (if s.mode = .off ∧ s.press = false then s'.mode = .off
     else if ...) ∧ (if ... then s'.x = s.x + 1 else ...)
```

Same four guards as the SMV `next(...)` and the Z3 `step()`.

The same `smv2lean` turns **any** Day-2 model into a Lean transition system:

`NuXMV/{Gcd, Mutex, Elevator}` ship *with proofs*; `Peterson`, `Prodcons` are translated samplers — prove the true INVARSPECs, refute the deliberately-false ones.

Bundling the counter's invariants

For *our* counter, $x \leq 10$ is already inductive on its own — the increment is guarded by $x < 10$, so $x' = x + 1 \leq 10$ falls out directly. (Not every property is so lucky — see the next slide.)

We still **bundle** the three safety properties into one invariant and prove it once:

$$\Phi(s) \equiv (s.x \leq 10) \wedge (s.\text{mode} = \text{off} \rightarrow s.x = 0)$$

then read each INVARSPEC off Φ via **invariant_strengthening** — one induction, three guarantees. (Φ — capital “phi” — names the bundled invariant; $\wedge$ is “and”.)

Why a too-weak invariant fails (worked)

A cleaner illustration. Two counters: x starts 0 and counts **up**, y starts m and counts **down**, in lockstep:

```
init:  x = 0,  y = m
step:  if x < m then  x := x + 1 ;  y := y - 1
```

Claim: $0 \leq y \leq m$ is an invariant. **Try to prove it inductive — and watch it fail:**

- Step case: assume only $0 \leq y \leq m$. Take a state with $y = 0$ but (say) $x = 0$. The guard $x < m$ is **enabled**, so we step: $y := y - 1 \rightarrow y$ underflows below 0. *X so $0 \leq y \leq m$ fails to be inductive.*
- The catch: that state ($x = 0, y = 0$ with $m > 0$) is **not actually reachable** — so $0 \leq y \leq m$ really *is* an invariant; it just isn't inductive. The weak invariant can't see unreachability: it never tied x and y together.

The fix: strengthen with the missing relationship

The repair I use: add the conserved quantity the program maintains — $x + y = m$:

$$\Psi(s) \equiv (0 \leq y \leq m) \wedge (x + y = m)$$

(Ψ — capital “psi” — the strengthened invariant; it **implies** the original $0 \leq y \leq m$.)

Now the step case goes through: if $x < m$, then since $x + y = m$ we get $y \geq 1$, so $y - 1 \geq 0$ — no underflow. The conjunct we added is exactly the fact that rules out the bogus state.

General proof rule (sound, but *incomplete* — you must find Ψ yourself):

To prove Φ is an invariant: find Ψ with $\Psi \Rightarrow \Phi$, and show Ψ is **inductive**. The strongest possible Ψ is “the reachable states themselves.”

The proof, in three parts

```
theorem counterInv_init   : ∀ s, CounterTS.init s → counterInv s
theorem counterInv_step   : ∀ s s', counterInv s → CounterTS.next s s' → counterInv s'
theorem counterInv_inductive : InductiveInvariant CounterTS counterInv :=
  {counterInv_init, counterInv_step}
```

1. holds initially, 2. preserved by every step, 3. bundle into **InductiveInvariant** (the **$\langle \dots, \dots \rangle$** is Lean's *anonymous constructor* — here it packages the two proofs into the **$\wedge$**).

Walk-through: `counterInv_step` case split

```
theorem counterInv_step : ∀ s s', counterInv s → CounterTS.next s s' → counterInv s' := by
  intro s s' ⟨hx, hmode⟩ ⟨p', hp, hmode_next, hx_next⟩
  cases hm : s.mode with
  | off ⇒ ... -- from off: press? → on/off, x unchanged
  | on ⇒
    by_cases hp : s.press = true
    · ... -- on + press → off, x := 0
    · by_cases hlt : s.x < 10
      · ... -- on, !press, x < 10 → on, x := x+1
      · ... -- on, !press, x ≥ 10 → off, x := 0
```

Each leaf: `simp [hm, ...]` to reduce the if-then-else, then `omega` for the arithmetic.

One leaf, fully worked: `on, ¬press, x < 10`

Take the branch where `mode = on`, the button is **not** pressed, and `x < 10`. The counter ticks: `x := x + 1`, mode stays `on`. We must re-establish $\Phi(s') = (s'.x \leq 10) \wedge (s'.mode = \text{off} \rightarrow s'.x = 0)$.

```
• -- context here:
  -- hx      : s.x ≤ 10                      (from  $\emptyset$  s)
  -- hmode   : s.mode = .off → s.x = 0      (from  $\emptyset$  s)
  -- hm      : s.mode = .on                 (this branch)
  -- hp      : ¬ (s.press = true)           (this branch)
  -- hlt     : s.x < 10                     (this branch)
  -- hx_next, hmode_next : s'.mode/s'.x via the if-then-else
simp [hm, hp, hlt] at hx_next hmode_next
-- simp used the three guards to collapse the if-then-else:
-- hx_next   : s'.x = s.x + 1
-- hmode_next : s'.mode = .on
refine {?, ?_} -- split  $\emptyset$  s' into its two conjuncts
• omega      -- ⊢ s'.x ≤ 10 : from s'.x = s.x+1 and s.x < 10 ✓
• intro hoff -- ⊢ s'.mode = .off → s'.x = 0
  simp [hmode_next] at hoff -- hoff : .on = .off – impossible, closes goal
```

Why one blanket tactic is *not* enough

Tempting: split all the cases, then close them uniformly —

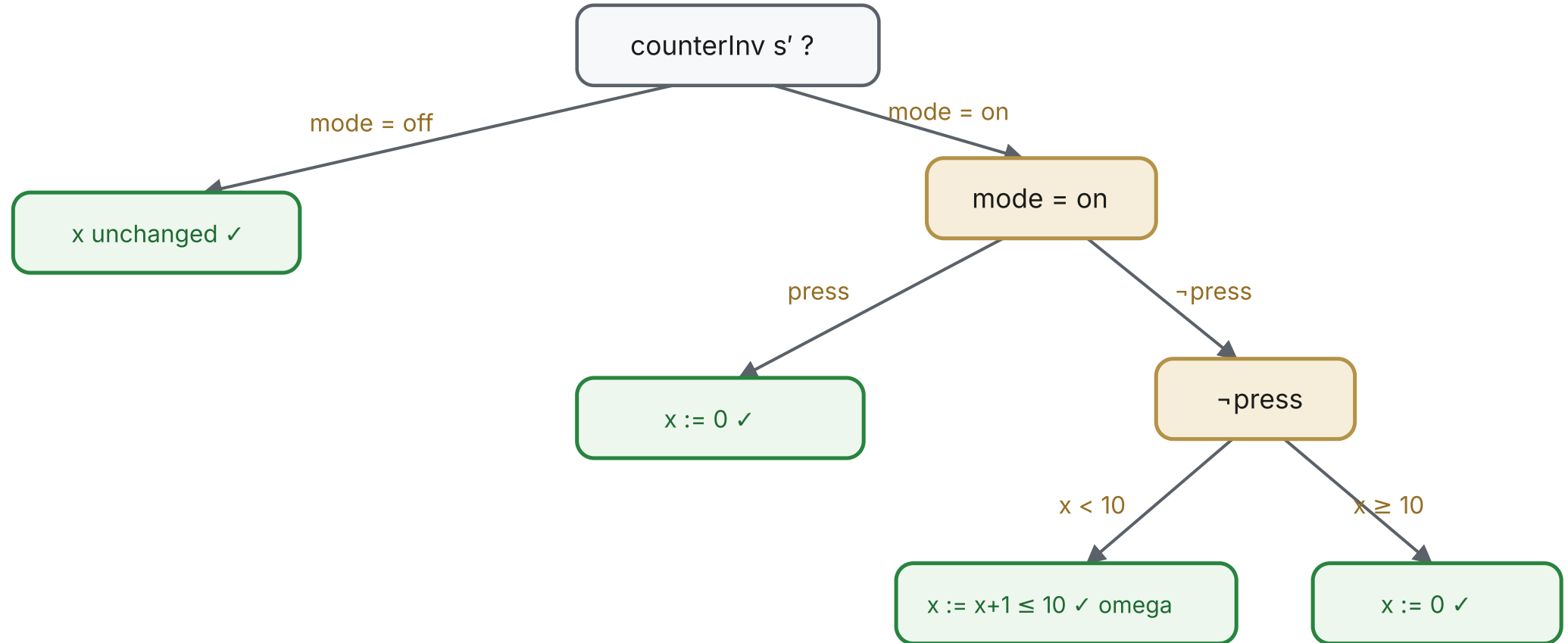
```
cases hm : s.mode <;> by_cases hpr : s.press = true <;>  
  by_cases hlt : s.x < 10 <;> simp_all <;> omega    -- x doesn't close every leaf
```

`t <;> s` runs `s` on **every** subgoal `t` produced — perfect for the shared work. But the leaves are **not** uniform:

- **on-branches** end in *arithmetic* (`x < 10 ⊢ x + 1 ≤ 10`) — `omega` closes these.
- **off-branches** end in a *constructor fact* (the goal reduces to `on ≠ off`) — not arithmetic, so `omega` fails; you need `decide` / `absurd`.

So the shipped proof closes each leaf with the *right* tool — `simp` + `omega` on the counting leaves, `absurd ... (by decide)` on the impossible-mode leaves — rather than one blanket tactic.

The same proof, as a tree



Three nested case splits ($\text{mode} \rightarrow \text{press} \rightarrow x < 10$); each of the four leaves closes by $\text{simp} + \text{omega} . \text{cases } \dots <;> \text{simp } \dots <;> \text{omega}$ collapses the whole tree into a couple of lines.

Strengthening: read off what you wanted

```
theorem CounterTS_inv1_proved : Invariant CounterTS (fun s => s.x ≤ 10) :=  
  invariant_strengthening CounterTS counterInv _ counterInv_inductive (fun _ h => h.1)  
  
theorem CounterTS_inv2_proved : Invariant CounterTS (fun s => s.mode = .off → s.x = 0) :=  
  invariant_strengthening CounterTS counterInv _ counterInv_inductive (fun _ h => h.2)
```

The user-facing invariants are **one-line corollaries** of the strong invariant. (`fun _ h`
`⇒ h.1` is an anonymous function that projects the first half of the `^`; `h.2` takes the
second.)

Checking it's really proved: no **sorry**

lake build **succeeds even with** **sorry** (it's a warning, not an error). To be sure a theorem is genuinely proved:

```
#print axioms CounterTS_inv1_proved
-- 'CounterTS_inv1_proved' depends on axioms: [propext, Quot.sound, Classical.choice]
```

Those three — **propext**, **Quot.sound**, **Classical.choice** — are the standard Lean axioms; seeing only them is the healthy result. If **sorryAx** appears, the proof has a hole. (This is exactly what the Day-3 autograder checks.)

Beyond safety: termination via ranking functions

Safety = “nothing bad happens” (what we just proved). **Liveness** = “something good *eventually* happens” — proved with a **ranking function**: a **Nat**-valued measure that strictly *decreases* every step.

```
-- Gcd.lean: Euclid's algorithm terminates because the first argument strictly drops
def gcd (a b : Nat) : Nat :=
  match a, b with
  | 0,   b => b
  | a+1, b => gcd (b % (a+1)) (a+1)          -- recurse on (b mod a+1, a+1)
  termination_by a                          -- the measure that must shrink
  decreasing_by exact Nat.mod_lt b (Nat.succ_pos a) -- b % (a+1) < a+1
```

A measure bounded below by 0 can't decrease forever $\Rightarrow$ the loop must stop. (This is the discrete cousin of a Lyapunov function.)

A second worked invariant: GCD correctness

Euclid's algorithm replaces the pair (a, b) with $(b \bmod a, a)$ until the first hits 0:

```
gcd a b = if a = 0 then b else gcd (b mod a) a
```

What makes it *correct*? The **conserved quantity**: the gcd of the pair never changes across a step.

$$\text{Inv}(a, b) \equiv \text{gcd}(a, b) = \text{gcd}(m, n)$$

This is an **inductive** invariant: $\text{gcd}(b \bmod a, a) = \text{gcd}(a, b)$, so each step preserves it. When the first argument hits 0, $\text{gcd}(0, b) = b$ reads off the answer. Same recipe as the counter — find the relationship the program *maintains*, prove it's preserved.

Live demo: Binomial expected value

For $X \sim \text{Binomial}(n, \theta)$ — n trials, each succeeding with probability θ — the expected number of successes is $n\theta$:

$$\mathbb{E}[X] = \sum_{k=0}^n k \binom{n}{k} \theta^k (1 - \theta)^{n-k} = n\theta$$

The same statement in Lean (Mathlib):

```
theorem binomial_expectation (n : ℕ) (θ : ℝ) :  
  ∑ k ∈ Finset.range (n + 1),  
    (k : ℝ) * (n.choose k : ℝ) * θ^k * (1 - θ)^(n - k)  
  = n * θ
```

Proof in three moves: drop the $k = 0$ term, absorb $k \binom{n}{k} = n \binom{n-1}{k-1}$ to factor $n\theta$ out, finish via the binomial theorem $(\theta + (1 - \theta))^{n-1} = 1$. Full proof:

`MathlibDemos/Binomial.lean`.

The Pythagorean theorem, three views

The classical $a^2 + b^2 = c^2$ has three natural Lean renderings — concrete, abstract, and coordinate — in `MathlibDemos/Pythagorean.lean`:

```
-- 1. Primitive triples – closed by `decide`, no Mathlib needed.
example : (3 : ℕ)^2 + 4^2 = 5^2 := by decide
example : (5 : ℕ)^2 + 12^2 = 13^2 := by decide
example : (8 : ℕ)^2 + 15^2 = 17^2 := by decide

-- 2. Inner-product form:  $\|u + v\|^2 = \|u\|^2 + \|v\|^2$  when  $u \perp v$ .
theorem pythagorean_inner {E} [NormedAddCommGroup E] [InnerProductSpace ℝ E]
  (u v : E) (h : ⟨u, v⟩_ℝ = 0) : \|u + v\|^2 = \|u\|^2 + \|v\|^2

-- 3. 2-D coordinates:  $\|v\|^2 = v_0^2 + v_1^2$  in  $\text{EuclideanSpace } \mathbb{R} (\text{Fin } 2)$ .
theorem pythagorean_2d (v : EuclideanSpace ℝ (Fin 2)) :
  \|v\|^2 = v 0 ^ 2 + v 1 ^ 2
```

$\sqrt{2}$ is irrational

Classical Euclid: assume $\sqrt{2} = p/q$ in lowest terms. Then $p^2 = 2q^2$, so $2 \mid p$, so $p = 2k$, so $q^2 = 2k^2$, so $2 \mid q$. But $\gcd(p, q) = 1$. $\perp$

Mathlib gives the *general* fact for any prime square root:

```
theorem sqrt_two_irrational : Irrational (Real.sqrt 2) :=  
  Nat.Prime.irrational_sqrt (by decide)
```

The arithmetic skeleton — formalised line-for-line with the proof above — is also available, with no real numbers involved ([MathlibDemos/SqrtTwoIrrational.lean](#)):

```
theorem no_coprime_sq_eq_two_mul_sq :  
  ¬ ∃ p q : ℕ, 0 < q ∧ Nat.Coprime p q ∧ p^2 = 2 * q^2
```

L2 recap

- A handful of tactics (`intro` , `simp` , `omega` , `cases` , `constructor`) close most goals.
- Mathlib is searchable via `exact?` / `apply?` and naming conventions.
- The counter is a Lean `TransitionSystem` ; prove invariants by **induction**.
- **Strengthen** to an inductive Φ , then read off each spec as a corollary.
- Verify with `#print axioms` — a build is not a proof.



Break

We resume after the break with AI in the proof loop, and Lean in industry.

L3 — AI in the loop + Lean for real

What's actually new

LLMs can now **draft tactic scripts** from a goal:

- Claude Code / Copilot in the editor — suggest the next tactic.
- AlphaProof, DeepSeek-Prover — generate whole Lean proofs.

The pattern, again: **AI proposes, the kernel disposes**. A wrong suggestion fails to type-check — it cannot corrupt a proof.

Two kinds of prover: automated vs. interactive

The lecture draws the line we've been living on both sides of:

	Automated (ATP / SMT)	Interactive (ITP)
Input	a formula	a theorem + a proof script
You do	press the button	guide the proof
Output	yes / no / <i>time-out</i>	a checked proof (or failure)
Logic	mostly first-order, decidable fragments	higher-order, full dependent types
Examples	nuXmv, Z3 (Days 1–2)	Lean, Coq, Isabelle, PVS

- **Automation buys ease; expressiveness costs interaction.** Some specs (e.g. CompCert's compiler-correctness) simply *can't be stated* in the first-order language an SMT solver decides — you need the higher-order logic of an ITP.
- The deep asymmetry behind all of it: **finding a proof is hard; checking one is easy.**

The AI-proving landscape

Because checking is cheap, the field has thrown search and learning at the *hard* half — finding proofs:

- **Classic ATP / benchmarks** — TPTP problem sets; “Formalizing 100 Theorems”; competitions long predate LLMs.
- **LLMs for math** — Minerva (Google, quantitative reasoning), tool-augmented models (LLM + Wolfram/SMT), and Gowers’ “automatic theorem proving project” all asked: can models *find* proofs?
- **Neural provers in Lean** — AlphaProof (IMO-silver level), DeepSeek-Prover-V2 ($\approx 89\%$ on miniF2F) — generate Lean scripts, then the **kernel checks every one**.
- **Programs, not just papers** — DARPA’s **PROVERS** program targets affordable formally-verified software at scale.

The unifying bet: let AI *search* for the proof, let the *kernel* certify it. Hard-to-find, easy-to-check is the ideal shape for an AI + verifier loop.

Live demo: AI-assisted repair

1. Open `Counter.lean`; weaken `counterInv` to drop the second conjunct.
2. `lake build` → the existing `counterInv_step` script **breaks** (its off-case used that conjunct).
3. Ask Claude Code to repair it.
4. Read what it proposes — does it re-add the conjunct, or find the simpler argument that `$x \leq 10$` needs on its own?

Where AI is reliable — and where it is not

Reliable	Unreliable
explaining what <code>omega</code> / <code>simp</code> did	"prove this" with no context
naming a Mathlib lemma	long multi-file proofs
drafting a routine tactic block	inventing the right <i>strengthening</i>
generating a counterexample	"looks right" but <code>omega</code> won't close

Treat suggestions as **drafts to read**, not finished proofs.

How to drive the AI well

1. **Give it context** — show your `TransitionSystem` and state definitions first.
2. **Ask for structure, not a finished proof** — “suggest a strengthening that makes this inductive, with one-line justification.”
3. **Paste the elaborator error back** — Lean’s error says exactly where it failed.
4. **Iterate** — the AI usually fixes its own mistakes given the error; sometimes you step in.

Lean in industry and research

- **PFR** (Tao et al.) and **Liquid Tensor Experiment** (Scholze/Commelin) — research math, formalized.
- **AWS Cedar** — the IAM policy language is **specified in Lean** (public repo); the spec is the source of truth.
- **DeepMind AlphaProof**; **Lean FRO** (funded nonprofit maintaining Lean).

Where Coq (now Rocq) and Isabelle still lead

Lean is ascendant in mathematics, but:

- **Iris / separation logic** (Coq) — concurrent program logics.
- **CompCert** (Coq) — the verified C compiler.
- **seL4** (Isabelle) — the verified microkernel.

Different tools, same kernel-checked guarantee. Pick by ecosystem.

L3 recap

- AI can draft tactic scripts; the **kernel still arbitrates**.
- AI is reliable on local tasks, unreliable on global/creative ones — read every suggestion.
- Drive it with **context + structure + the elaborator error**.
- Lean is real: PFR, Mathlib, AWS Cedar, AlphaProof; Coq/Isabelle lead in verified systems.

Wrap + work

Day 3 in one slide

- A proof assistant checks every step against a tiny **kernel**.
- Prove invariants by **induction** — works for infinite/parametric systems model checking can't touch.
- **Strengthen** to an inductive Φ ; read off each spec as a corollary; verify with `#print axioms`.
- AI **drafts**, the kernel **decides**, you **judge** the creative steps.

Day 4 preview

- Leave the abstract counter; bring everything to bear on **actual code**.
- **CBMC** — bounded model checking of C source (Day 1's idea, on real programs).
- **Cryptol + SAW** — bit-precise specs, proved equivalent to C implementations.
- Then the frontier: **neural-network verification** and industrial deployments.

In-session exercise

In `examples/CounterDemo` :

1. `lake build` — the solution files compile clean (no `sorry`); open a `*Starter.lean` to see the fill-in-the-blank `sorry` holes you'd complete.
2. In `Counter.lean`, find the three pieces of the inductive-invariant pattern (`counterInv_init`, `counterInv_step`, `counterInv_inductive`).
3. Prove **one** new corollary from the strengthened Φ — e.g. `mode = off  $\rightarrow$  x < 10` — via `invariant_strengthening`, and confirm with `#print axioms` (no `sorryAx`).

Homework (ungraded, for depth)

New to Lean? Warm up first on the `DiscreteMath` set-theory starter and the `CounterLadder` tactic ladder.

Pick **one** (see `assignments/day03.md`):

- Prove the **combined** invariant $(x \leq 10) \wedge (\text{mode} = \text{off} \rightarrow x = 0) \wedge (x > 0 \rightarrow \text{mode} = \text{on})$ is inductive.
- Change the bound `10` to `25` (it appears in the `next` guards and in `counterInv`) and re-prove `CounterTS_inv1` (use Claude Code for the edits).
- Translate a Day-2 model into Lean — `scripts/smv2lean/to_lean.sh` `day02/examples/<model>.smv` (or by hand) — and prove one invariant. (`Peterson`, `Prodcons` are pre-translated; like the Day-2 samplers, some INVARSPECs hold and some are deliberately false.)

Use Claude Code as a partner; note one thing it got right and one it got wrong.

References for Day 3

- **Avigad, de Moura, Kong, Carneiro.** *Theorem Proving in Lean 4* — https://leanprover.github.io/theorem_proving_in_lean4/
- **Avigad.** *Mathematics and the formal turn*, Bull. AMS 2024.
- **Alper.** *Embracing AI and formalization*, Bull. AMS 2026.
- **Bayer et al.** *Mathematical Proof Between Generations*, Notices AMS 2024 (Buzzard on Lean).
- **AlphaProof** (DeepMind, 2024); **DeepSeek-Prover-V2** (2025).
- **Lean community** — <https://leanprover-community.github.io/> · Lean Game Server (Natural Number Game).

Full list: repo [README.md](#).

Next session: Day 4

Repo: github.com/ttj/fmaiv — the CounterDemo Lake project, slides, and the assignment.