Formal Proof Verification

An Introduction to Computer-Assisted Mathematics

Christoph Spiegel

Block Course 2026

Freie Universität Berlin — Zuse Institute Berlin

P01_Introduction

Introduction

What is a Formal System?

A **formal system** is a game on symbols: an alphabet, rules for valid strings, and rules for deriving new strings from old ones. Nothing is assumed — only rule application counts.

What is a Formal System?

A **formal system** is a game on symbols: an alphabet, rules for valid strings, and rules for deriving new strings from old ones. Nothing is assumed — only rule application counts.

The MIU System (Hofstadter, *Gödel, Escher, Bach*, 1979)

- **Alphabet**: `M`, `I`, `U`
- **Starting string**: `MI`
- **Rules** (`x`, `y` denote arbitrary strings):
  1. you can turn `xI` into `xIU`
  2. you can turn `Mx` into `Mxx`
  3. you can turn `xIIIy` into `xUy`
  4. you can turn `xUUy` into `xy`

What is a Formal System?

A **formal system** is a game on symbols: an alphabet, rules for valid strings, and rules for deriving new strings from old ones. Nothing is assumed — only rule application counts.

The MIU System (Hofstadter, *Gödel, Escher, Bach*, 1979)

- **Alphabet**: `M`, `I`, `U`
- **Starting string**: `MI`
- **Rules** (`x`, `y` denote arbitrary strings):
  1. you can turn `xI` into `xIU`
  2. you can turn `Mx` into `Mxx`
  3. you can turn `xIIIy` into `xUy`
  4. you can turn `xUUy` into `xy`

Challenge: Can you derive `MIIIII` from `MI`? What about `MU`?

What is a Formal System?

String	Rule
MI	start
MII	Rule 2
MIIII	Rule 2
MIIIIIIII	Rule 2
MIIIIIIIIU	Rule 1
MIIIIIUU	Rule 3
MIIIII	Rule 4

What is a Formal System?

String	Rule	I-count
MI	start	1
MII	Rule 2	$\times 2 = 2$
MIIII	Rule 2	$\times 2 = 4$
MIIIIIIII	Rule 2	$\times 2 = 8$
MIIIIIIIIU	Rule 1	$= 8$
MIIIIIUU	Rule 3	$-3 = 5$
MIIIII	Rule 4	$= 5$

Why can MU not appear? The I-count starts at 1 and can only be doubled or decreased by 3, keeping it $\not\equiv 0 \pmod 3$ — so MU is not derivable.

What is a Formal System?

String	Rule	I-count
MI	start	1
MII	Rule 2	$\times 2 = 2$
MIIII	Rule 2	$\times 2 = 4$
MIIIIIIII	Rule 2	$\times 2 = 8$
MIIIIIIIIU	Rule 1	$= 8$
MIIIIIUU	Rule 3	$-3 = 5$
MIIIII	Rule 4	$= 5$

Why can MU not appear? The I-count starts at 1 and can only be doubled or decreased by 3, keeping it $\not\equiv 0 \pmod 3$ — so MU is not derivable.

Remark. The argument never applied an MIU rule — it reasoned *about* the system from outside. In fact, derivability is decidable: a string is derivable iff it starts with M, contains no other M, and its I-count is $\not\equiv 0 \pmod 3$.

What is Formal Verification?

In **formal verification**, a program and its desired correctness property are encoded in a formal system. Checking whether the property holds reduces to asking whether a certain derivation exists — checked mechanically, modulo errors in the checker.

What is Formal Verification?

In **formal verification**, a program and its desired correctness property are encoded in a formal system. Checking whether the property holds reduces to asking whether a certain derivation exists — checked mechanically, modulo errors in the checker.

This is not hypothetical — formal verification has been deployed in safety-critical and security-critical systems for decades:

System	Property verified
CompCert	Compiled C code matches source semantics
ProVerif	Protocol secrecy/authentication properties (symbolic model)
DAIDALUS (NASA)	UAV collision avoidance algorithm is safe
FM 8501	Verified microprocessor: design model matches ISA specification
Mondex	Electronic purse (ITSEC Level E6 certification)

Proof Assistants and Libraries

System	Year	Foundation	Notable for
Dependent Type Theory			
`Lean`	**2013**	**DTT (constructive core)**	`Mathlib`**; broad community math library**
`Rocq`	1989	CIC (DTT) / Intuit.	`MathComp`; CompCert, Feit-Thompson
`Agda`	2007	UDTT / Intuit.	Standard Library; HoTT
Higher-Order Logic			
`Isabelle`	1986	HOL	`AFP`; seL4 kernel
`HOL Light`	1996	HOL	`Flyspeck`; Kepler conjecture
`PVS`	1992	HOL+DTT	`NASA Libraries`; safety-critical
Other			
`Mizar`	1973	TG / Classical	`MML`; oldest active system
`ACL2`	1990	FORL / Classical	`ACL2 Books`; hardware verification
`Metamath`	1992	ZFC / Classical	`set.mm`; ZFC-based

Proofs exceed what humans can verify

1 Computation-heavy proofs

- **Kepler conjecture (Hales, 1998):** reduced to checking thousands of nonlinear inequalities and tens of thousands of linear programs by computer. After four years of review, the referees reported being "99% certain" of correctness but unable to certify the computer calculations.

Proofs exceed what humans can verify

1 Computation-heavy proofs

- **Kepler conjecture (Hales, 1998):** reduced to checking thousands of nonlinear inequalities and tens of thousands of linear programs by computer. After four years of review, the referees reported being "99% certain" of correctness but unable to certify the computer calculations.

2 Proofs no one can read end-to-end

- **Classification of finite simple groups (1955–2004)**: tens of thousands of pages across around 100 authors; no single person has read it in full.
- **Feit-Thompson theorem (1963)**: a 255-page paper; a full revision spans two books.

Peer review is not keeping up

3 Undetected errors

- **Smooth toric varieties (2004–2023)**: *Annals* published contradictory results in 2004 and 2006; retracted in 2023.
- **Adam optimizer (ICLR 2015)**: over 250,000 citations; shown in 2018 to not converge on certain convex objectives.

Peer review is not keeping up

3 Undetected errors

- **Smooth toric varieties (2004–2023)**: *Annals* published contradictory results in 2004 and 2006; retracted in 2023.
- **Adam optimizer (ICLR 2015)**: over 250,000 citations; shown in 2018 to not converge on certain convex objectives.

4 Slow decisions

- A typical submission waits over a year for a decision; top journals regularly exceed 18 months. [AMS Backlog 2024]

Peer review is not keeping up

3 Undetected errors

- **Smooth toric varieties (2004–2023)**: *Annals* published contradictory results in 2004 and 2006; retracted in 2023.
- **Adam optimizer (ICLR 2015)**: over 250,000 citations; shown in 2018 to not converge on certain convex objectives.

4 Slow decisions

- A typical submission waits over a year for a decision; top journals regularly exceed 18 months. [AMS Backlog 2024]

5 Disputes it cannot resolve

- **Mochizuki's ABC conjecture (2012):** 500 pages; 13 years later, no consensus. Scholze–Stix identified a gap (2018); Mochizuki disputes it; the community has no mechanism to arbitrate. A $1M prize has been announced. [Quanta]

The authority trap

- **Kapranov–Voevodsky, ∞-groupoids (1991)**: Simpson shows the main theorem is false in 1998; the exact error takes another 15 years to locate. Co-authored by a future Fields medalist; studied for over two decades.
- **Voevodsky, presheaves with transfers (1993)**: error in a key lemma found by Deligne during a 1999 lecture series — six years of active use without detection.

> *"A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail."*
>
> — Voevodsky, IAS essay (2014)

Formalization adds insight, not just certainty

The case for formal verification is not only **defensive**. Formalization forces **explicit reasoning** and can reveal structure invisible in an informal proof — even to its own author.

Formalization adds insight, not just certainty

The case for formal verification is not only **defensive**. Formalization forces **explicit reasoning** and can reveal structure invisible in an informal proof — even to its own author.

Peter Scholze posted a public challenge in 2020, uncertain whether his own proof of the **liquid tensor experiment** was correct, and invited the community to formalize and check it. When the **Lean** formalization succeeded in 2022:

> *"[I learned] what actually makes the proof work! [. . .] I did not understand why the argument worked [. . .] but [the formalization] made me realize that actually the key thing happening is a reduction from a non-convex problem over the reals to a convex problem over the integers."*
>
> — Peter Scholze, 2021

Four Color Theorem (1976 / 2005)

Theorem

Every planar map can be colored with at most 4 colors such that no two adjacent regions receive the same color.

Challenge. Appel and Haken (1976) gave the first proof using a computer — and the first that human referees could not independently check. Two earlier proposed proofs had been accepted for over a decade before errors were found.

Four Color Theorem (1976 / 2005)

Theorem

Every planar map can be colored with at most 4 colors such that no two adjacent regions receive the same color.

Challenge. Appel and Haken (1976) gave the first proof using a computer — and the first that human referees could not independently check. Two earlier proposed proofs had been accepted for over a decade before errors were found.

Formalization. Gonthier (2005) in `Rocq`; the checker replaced the external computer enumeration entirely.

Feit-Thompson Theorem (1963 / 2012)

Theorem

Every finite group of odd order is solvable: it can be built up from abelian groups by iterated extensions.

Challenge. The proof is a 255-page paper; a full revision spans two books. No single person has read it all.

Feit-Thompson Theorem (1963 / 2012)

Theorem

Every finite group of odd order is solvable: it can be built up from abelian groups by iterated extensions.

Challenge. The proof is a 255-page paper; a full revision spans two books. No single person has read it all.

Formalization. Gonthier et al. (2012) in `Rocq`, over several years, with a large team at Microsoft Research and Inria.

No individual has read this proof in full. The formalization replaced accumulated trust with unconditional certainty.

Liquid Tensor Experiment (2020 / 2022)

Theorem

For $0 < p' < p \leq 1$, certain Ext groups vanish: $\mathrm{Ext}^i(\mathcal{M}_{p'}(S),\, V) = 0$ for all $i \geq 1$, where S is a profinite set and V a p-Banach space. (simplified; Theorem 9.1 of Scholze's *Analytic*)

Challenge. Scholze posted a public challenge: no one else had checked the details of this foundational result, and for a theorem of this importance, near-certainty was not enough.

Liquid Tensor Experiment (2020 / 2022)

Theorem

For $0 < p' < p \leq 1$, certain Ext groups vanish: $\mathrm{Ext}^i(\mathcal{M}_{p'}(S),\, V) = 0$ for all $i \geq 1$, where S is a profinite set and V a p-Banach space. (simplified; Theorem 9.1 of Scholze's *Analytic*)

Challenge. Scholze posted a public challenge: no one else had checked the details of this foundational result, and for a theorem of this importance, near-certainty was not enough.

Formalization. Community project in `Lean` (2020–2022), initiated by Scholze. Mid-formalization, Scholze discovered the proof's key mechanism: a reduction from a non-convex problem over $\mathbb{R}$ to a convex one over $\mathbb{Z}$.

> *"complex proofs about complex objects"* — Buzzard (ICM 2022)

Polynomial Freiman-Ruzsa Conjecture (2023)

Theorem

If $A \subseteq \mathbb{F}_2^n$ satisfies $|A+A| \leq K|A|$, then A is covered by at most $2K^{12}$ cosets of a subspace $H \leq \mathbb{F}_2^n$ with $|H| \leq |A|$.

Challenge. A major breakthrough in additive combinatorics by Gowers, Green, Manners, and Tao, resolving a decades-old conjecture.

Polynomial Freiman-Ruzsa Conjecture (2023)

Theorem

If $A \subseteq \mathbb{F}_2^n$ satisfies $|A+A| \leq K|A|$, then A is covered by at most $2K^{12}$ cosets of a subspace $H \leq \mathbb{F}_2^n$ with $|H| \leq |A|$.

Challenge. A major breakthrough in additive combinatorics by Gowers, Green, Manners, and Tao, resolving a decades-old conjecture.

Formalization. Tao, Dillies, and Mehta in `Lean 4` + `Mathlib`, within three weeks of the preprint.

Formalized within weeks of the preprint: formal proof can now operate at the research frontier, not just on classical results decades old.

What Formalization Changes

Traditional Mathematics	Formalized Mathematics
Natural language Lemma, Theorem, Proof, ...	**Formal language** functional programming style
Variable detail specialist papers to student textbooks	**Uniform detail** every step justified, axioms explicit
Implicit foundations axioms rarely stated or agreed upon	**Explicit foundations** classical or constructive; DTT vs. ZFC
Verified by humans trust, authority, and shared background	**Machine-checked** certain, modulo errors in the checker
+ Easy to write and read	– (Currently) hard to write and read
– Can contain errors	+ Guaranteed; traceable to axioms

"You don't understand things. You just get used to them." —Von Neumann

Where the Field Is Heading

"When integrated with tools such as formal proof verifiers, internet search, and symbolic math packages, I expect, say, 2026-level AI, when used properly, will be a trustworthy co-author in mathematical research."
— Tao (2023)

Where the Field Is Heading

"When integrated with tools such as formal proof verifiers, internet search, and symbolic math packages, I expect, say, 2026-level AI, when used properly, will be a trustworthy co-author in mathematical research."
— Tao (2023)

1 Signals of progress

- **AlphaProof + AlphaGeometry 2** (2024–2025, Google DeepMind): Lean-based RL; 4/6 IMO 2024 problems. [Nature 2025]
- **Gemini Deep Think** (2025): IMO 2025 gold; 5/6 problems in natural language (not machine-checked). [DeepMind 2025]

Where the Field Is Heading

"When integrated with tools such as formal proof verifiers, internet search, and symbolic math packages, I expect, say, 2026-level AI, when used properly, will be a trustworthy co-author in mathematical research."
— Tao (2023)

1 Signals of progress

- **AlphaProof + AlphaGeometry 2** (2024–2025, Google DeepMind): Lean-based RL; 4/6 IMO 2024 problems. [Nature 2025]
- **Gemini Deep Think** (2025): IMO 2025 gold; 5/6 problems in natural language (not machine-checked). [DeepMind 2025]

2 What remains hard

- Hairer et al. (FirstProof, 2026): 10 unpublished research problems; one-shot remains hard. [FAQ]
- Gowers (2025): proof *style* does not guarantee proof *reasoning*.

A history of computational mathematics

Year	Milestone
1976	Four color theorem: proof by exhaustion; first major proof referees could not check end-to-end.
1996	McCune proves Robbins algebras are boolean using automated theorem proving (ATP).
1998	Hales proves the Kepler conjecture using interval arithmetic on thousands of nonlinear inequalities and over 40,000 linear programs.
2016	Boolean Pythagorean triples resolved using SAT; the certificate is 200 TB.
2020	Keller's conjecture (dimension 7, 90 years open) resolved using SAT; 200 GB certificate.
2024	Empty hexagon number $h(6) = 30$ (Happy Ending problem): SAT (17,300 CPU hours) plus `Lean` formal verification.
2026	Erdős Problem #728 claimed resolved by an AI system, producing a `Lean` proof (translated to informal in the paper).

Introduction — Contents

Continuity

In point-set topology, continuity is defined entirely via open sets:

Definition

$f : X \to Y$ is **continuous** if for every open $U \subseteq Y$, the preimage $f^{-1}(U)$ is open in X.

Continuity

In point-set topology, continuity is defined entirely via open sets:

Definition

$f : X \to Y$ is **continuous** if for every open $U \subseteq Y$, the preimage $f^{-1}(U)$ is open in X.

Theorem

If $f : X \to Y$ and $g : Y \to Z$ are continuous, then $g \circ f$ is continuous.

Continuity

In point-set topology, continuity is defined entirely via open sets:

Definition

$f : X \to Y$ is **continuous** if for every open $U \subseteq Y$, the preimage $f^{-1}(U)$ is open in X.

Theorem

If $f : X \to Y$ and $g : Y \to Z$ are continuous, then $g \circ f$ is continuous.

Proof. Let $U \subseteq Z$ be open. Set $V := g^{-1}(U)$ and $W := f^{-1}(V)$. Then V is open (since g is continuous), W is open (since f is continuous), and $W = (g \circ f)^{-1}(U)$. □

Let's look at this in Lean.

Infinitely many primes

Theorem

For every $n \in \mathbb{N}$, there exists a prime $p \geq n$.

Infinitely many primes

Theorem

For every $n \in \mathbb{N}$, there exists a prime $p \geq n$.

Proof (Euclid). Let p be the smallest prime factor of $n! + 1$. Note that this is well-defined as $n! \geq 1$ and hence $n! + 1 \geq 2$, so that in particular $n! + 1 \neq 1$.

Infinitely many primes

Theorem

For every $n \in \mathbb{N}$, there exists a prime $p \geq n$.

Proof (Euclid). Let p be the smallest prime factor of $n! + 1$. Note that this is well-defined as $n! \geq 1$ and hence $n! + 1 \geq 2$, so that in particular $n! + 1 \neq 1$.

To show that $p \geq n$, suppose for contradiction that $p < n$. Since p is a positive integer less than n, it appears as a factor in the product $n! = 1 \cdot 2 \cdots n$, so $p \mid n!$. But p also divides $n! + 1$, so $p \mid (n! + 1) - n! = 1$, contradicting that p is prime. $\square$

Infinitely many primes

Theorem

For every $n \in \mathbb{N}$, there exists a prime $p \geq n$.

Proof (Euclid). Let p be the smallest prime factor of $n! + 1$. Note that this is well-defined as $n! \geq 1$ and hence $n! + 1 \geq 2$, so that in particular $n! + 1 \neq 1$.

To show that $p \geq n$, suppose for contradiction that $p < n$. Since p is a positive integer less than n, it appears as a factor in the product $n! = 1 \cdot 2 \cdots n$, so $p \mid n!$. But p also divides $n! + 1$, so $p \mid (n! + 1) - n! = 1$, contradicting that p is prime. □

We revisit this in Part 6 and formalize several genuinely different proofs, from Euclid to Furstenberg's topological proof.

Let's look at this in Lean.

Two examples, one system

1. **More code than proof.** A two-line informal argument becomes dozens of lines in Lean. The overhead is not wasted: every implicit step is now explicit and machine-checked.
2. **One library, many domains.** Topology and number theory are unrelated, yet both proofs drew on the same Mathlib infrastructure.

Two examples, one system

1. **More code than proof.** A two-line informal argument becomes dozens of lines in Lean. The overhead is not wasted: every implicit step is now explicit and machine-checked.
2. **One library, many domains.** Topology and number theory are unrelated, yet both proofs drew on the same Mathlib infrastructure.

Habit. When Lean forces precision, ask which assumption your informal proof left implicit.

Introduction — Contents

Lean is a programming language

Lean is not just a proof assistant. It is a **functional programming language**, closer to Haskell or OCaml than to Python or Java.

Lean is a programming language

Lean is not just a proof assistant. It is a **functional programming language**, closer to Haskell or OCaml than to Python or Java.

1 Programming model

- **Functional**: expressions instead of statements, immutable data.
- **Pattern matching**: replaces `if`/`else` chains and `switch`. Each case binds the components it matches.
- Structures and namespaces replace classes. Dot notation (`c.area`) works, but without inheritance or mutable state.
- `#eval` runs code; `#check` inspects types.

Lean is a programming language

Lean is not just a proof assistant. It is a **functional programming language**, closer to Haskell or OCaml than to Python or Java.

1 Programming model

- **Functional**: expressions instead of statements, immutable data.
- **Pattern matching**: replaces `if`/`else` chains and `switch`. Each case binds the components it matches.
- Structures and namespaces replace classes. Dot notation (`c.area`) works, but without inheritance or mutable state.
- `#eval` runs code; `#check` inspects types.

2 Type discipline

- Types enforced at compile time, not optional hints.
- Compiler infers types when unambiguous.

Let's look at this in Lean.

What we just saw in Lean

1 From programs to proofs

- A `theorem` is a `def` whose return type is a proposition. A proof is the function body.
- A type system rich enough to state mathematical properties can check proofs by type-checking programs.

What we just saw in Lean

1 From programs to proofs

- A `theorem` is a `def` whose return type is a proposition. A proof is the function body.
- A type system rich enough to state mathematical properties can check proofs by type-checking programs.

2 Practical details

- `sorry` and `axiom` create **explicit** proof gaps; Lean tracks them. No gaps means a fully checked result.
- Definitions matter: in $\mathbb{N}$, subtraction truncates, so `3 - 5 = 0`.
- Lean's kernel logic is constructive, but Mathlib routinely uses classical axioms (often enabled implicitly by tactics).

What we just saw in Lean

1 From programs to proofs

- A `theorem` is a `def` whose return type is a proposition. A proof is the function body.
- A type system rich enough to state mathematical properties can check proofs by type-checking programs.

2 Practical details

- `sorry` and `axiom` create **explicit** proof gaps; Lean tracks them. No gaps means a fully checked result.
- Definitions matter: in $\mathbb{N}$, subtraction truncates, so `3 - 5 = 0`.
- Lean's kernel logic is constructive, but Mathlib routinely uses classical axioms (often enabled implicitly by tactics).

Habit. Think of a theorem as a function: hypotheses in, conclusion out. A proof is the body that transforms one into the other.

Course Outline

The course has two phases: guided lectures with exercises, followed by independent project work.

Course Outline

The course has two phases: guided lectures with exercises, followed by independent project work.

Part	Focus
P01	**Introduction:** two Lean proofs; Lean as a programming language
P02	**Logic:** propositional reasoning, quantifiers, and core tactics
P03	**Set theory:** subsets, intersections, unions, and set families
P04	**Type theory:** dependent types, inductive types, and type classes
P05	**Natural numbers:** build arithmetic from scratch; inequalities
P06+	**Project phase:** independent formalization projects using Mathlib

What You Will Walk Away With

Mindset. Formalization changes how you reason.

What You Will Walk Away With

Mindset. Formalization changes how you reason.

Lean.

- Read and write tactic proofs in **Lean 4**.
- Use **Mathlib**: find lemmas, read notation, and reuse results.
- Understand practical type theory: `Prop` vs. `Type`, dependent types, and type classes.
- Structure projects and contribute to **Mathlib**.

What You Will Walk Away With

Mindset. Formalization changes how you reason.

Lean.

- Read and write tactic proofs in **Lean 4**.
- Use **Mathlib**: find lemmas, read notation, and reuse results.
- Understand practical type theory: `Prop` vs. `Type`, dependent types, and type classes.
- Structure projects and contribute to **Mathlib**.

Workflow.

- Use **git**: branches, pull requests, and code review.
- An outlook on using **LLMs** for coding and research.
- Complete a small, polished formalization as a course project.

Setup checklist

1. **Developer tools**
   - **macOS:** `xcode-select --install` (includes git)
   - **Linux:** `sudo apt install git` (or equivalent)
   - **Windows:** install WSL2 with Ubuntu; work inside WSL

Setup checklist

1. **Developer tools**
   - **macOS:** `xcode-select --install` (includes git)
   - **Linux:** `sudo apt install git` (or equivalent)
   - **Windows:** install WSL2 with Ubuntu; work inside WSL

2. **VS Code** Install from code.visualstudio.com. Windows/WSL users: add the *Remote - WSL* extension.

Setup checklist

1. **Developer tools**
   - **macOS:** `xcode-select --install` (includes git)
   - **Linux:** `sudo apt install git` (or equivalent)
   - **Windows:** install WSL2 with Ubuntu; work inside WSL

2. **VS Code** Install from code.visualstudio.com. Windows/WSL users: add the *Remote - WSL* extension.

3. **Lean 4 + VS Code extension** Install the `lean4` extension in VS Code; it installs elan and Lean automatically. The *Lean Infoview* shows goals and context as you type — read it constantly.

Setup checklist

1. **Developer tools**
   - **macOS:** `xcode-select --install` (includes git)
   - **Linux:** `sudo apt install git` (or equivalent)
   - **Windows:** install WSL2 with Ubuntu; work inside WSL

2. **VS Code** Install from code.visualstudio.com. Windows/WSL users: add the *Remote - WSL* extension.

3. **Lean 4 + VS Code extension** Install the `lean4` extension in VS Code; it installs elan and Lean automatically. The *Lean Infoview* shows goals and context as you type — read it constantly.

4. **GitHub + SSH** Create an account, generate a key (`ssh-keygen -t ed25519`), add it at github.com/settings/ssh/new — or log in to your GitHub account directly in VS Code.

Creating your first project

Path A — New project:

1. Create a new repository on github.com/new
2. Initialize with Mathlib: `lake init . math`

Path B — Course repo: Fork the course repository on GitHub.

Both paths: Clone via VS Code's *Clone Git Repository* command.

Creating your first project

Path A — New project:

1. Create a new repository on github.com/new
2. Initialize with Mathlib: `lake init . math`

Path B — Course repo: Fork the course repository on GitHub.

Both paths: Clone via VS Code's *Clone Git Repository* command.

File	Purpose
`lean-toolchain`	Pins the Lean version; elan reads this automatically.
`lakefile.toml`	Project config: name, dependencies, build targets.
`*.lean`	Root import file for your library.
`.git/`	Git repository data.
`.gitignore`	Files excluded from version control.
`.github/`	CI workflows.
`.lake/`	Build cache (do not commit).

Building and troubleshooting

Building

1. `lake exe cache get` — download pre-built Mathlib (avoids hours of compilation)
2. `lake build` — compile the project
3. Open a `.lean` file in VS Code; the Infoview should show no errors

Building and troubleshooting

Building

1. `lake exe cache get` — download pre-built Mathlib (avoids hours of compilation)
2. `lake build` — compile the project
3. Open a `.lean` file in VS Code; the Infoview should show no errors

When Lean acts up (escalating steps)

1. Restart server: `Ctrl+Shift+P` → *Lean 4: Restart Server*
2. `pkill -f lean && lake build`
3. `pkill -f lean && rm -rf .lake && lake exe cache get && lake build`