Lean: Machine-Checked Mathematics and Verified Programming, Past and Future

Leo de Moura
Senior Principal Applied Scientist, AWS
Chief Architect, Lean FRO

June 19, 2025

How can we ensure that our most critical software and hardware systems behave exactly as intended, and that every proof of correctness is independently verifiable?

Before Lean, there was Z3

Z3 is a state-of-the-art SMT solver.

Z3 powered **bug-finding** pipelines like Microsoft's SAGE fuzzing tool: caught thousands of defects.

But for whole-program verification, proofs were brittle: minor code edits broke solver traces.

The Lean project, started in 2013, aimed at merging interactive and automated theorem proving.

Lean is an open-source programming language and proof assistant that is transforming how we approach mathematics, software verification, and AI.

Lean opens up new possibilities for **collaboration** in mathematics.

Lean and its tooling are implemented in Lean. Lean is very **extensible**.

LSP, Parser, Macro System, Elaborator, Type Checker, Tactic Framework, Proof automation, Compiler, Build System, Documentation Authoring Tool.

Lean has a **small trusted core**, proofs can be exported and independently checked.

The **Lean FRO** is a nonprofit dedicated to developing Lean.

Lean is based on dependent type theory from the beginning

*"**Algebraic reasoning is fundamental to modern mathematics**. We calculate with **abstract structures** the same way we calculate with numbers; for example, we take sums, products, powers, and limits of structures just as we take sums, products, powers, and limits of numbers. Then, in the same breath, we talk about elements of those structures and operations on those elements. To formalize this kind of reasoning, **we need a language in which types and structures are first-class objects**, and we need tools that can interpret ambiguous notation and fill in the information that is left implicit in informal mathematics. **There is no way around using dependent type theory for all that.**" Jeremy Avigad*

Users focus on mathematics, not encoding tricks.

Lean is based on dependent type theory

An example *by Kim Morrison*:

```
structure IndexMap (α : Type u) (β : Type v) [BEq α] [Hashable α] where
  private indices : HashMap α Nat
  private keys : Array α
  private values : Array β
  private size_keys' : keys.size = values.size := by grind
  private WF : ∀ (i : Nat) (a : α), keys[i]? = some a ↔ indices[a]? = some i := by grind
```

Full example here.

An example *by Kim Morrison*:

```
structure IndexMap (α : Type u) (β : Type v) [BEq α] [Hashable α] where
  private indices : HashMap α Nat
  private keys : Array α
  private values : Array β
  private size_keys' : keys.size = values.size := by grind
  private WF : ∀ (i : Nat) (a : α), keys[i]? = some a ↔ indices[a]? = some i := by grind
```

```
def insert [LawfulBEq α] (m : IndexMap α β) (a : α) (b : β) : IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    { indices := m.indices
      keys := m.keys.set i a
      values := m.values.set i b }
  | none =>
    { indices := m.indices.insert a m.size
      keys := m.keys.push a
      values := m.values.push b }
```

An example *by Kim Morrison*:

```
/-! ### Verification theorems -/

attribute [local grind] getIdx findIdx insert

@[grind] theorem getIdx_findIdx (m : IndexMap α β) (a : α) (h : a ∈ m) :
    m.getIdx (m.findIdx a h) = m[a] := by grind

@[grind] theorem mem_insert (m : IndexMap α β) (a a' : α) (b : β) :
    a' ∈ m.insert a b ↔ a' = a ∨ a' ∈ m := by
  grind

@[grind] theorem getElem_insert (m : IndexMap α β) (a a' : α) (b : β) (h : a' ∈ m.insert a b) :
    (m.insert a b)[a']'h = if h' : a' == a then b else m[a'] := by
  grind

@[grind] theorem findIdx_insert_self (m : IndexMap α β) (a : α) (b : β) :
    (m.insert a b).findIdx a (by grind) = if h : a ∈ m then m.findIdx a h else m.size := by
  grind
```

We Listen to Our Users: Classical Mathematics from Day 1

User-driven design philosophy: Classical logic and mathematics as defaults

Our first user was a mathematician: Jeremy Avigad

The math community using Lean is growing rapidly. They love the system

Lean is also a programming language, you can be constructive when it matters.

Practical focus: **Verification engineers prioritize getting proofs done** over foundational concerns

Mathlib

The Lean Mathematical Library supports a wide range of projects.

It is an open-source **collaborative project** with over 500 contributors and 1.8M LoC.

"I'm investing time now so that somebody in the future can have that amazing experience",

Heather Macbeth

Quantamagazine | Physics | Mathematics | Biology | Computer Science | Topics | Archive

FOUNDATIONS OF MATHEMATICS

Building the Mathematical Library of the Future

Lean is a Development Environment for formal verification

Rich user interface and integrated tooling

Build system, LSP server, and VS Code plugin work seamlessly together

Lake, Lean make, is our cargo

Reservoir (reservoir.lean-lang.org): Our package ecosystem, think crates.io

Real-time feedback: Errors, goals, and hints as you type

“Great tooling is essential” Jared Roesch

Mathlib > RingTheory > Finiteness.lean

```lean
theorem FG.stabilizes_of_iSup_eq {M' : Submodule R M} (hM' : M'.FG) (N : ℕ →o Submodule R M)
    (H : iSup N = M') : ∃ n, M' = N n := by
  obtain ⟨S, hS⟩ := hM'
  have : ∀ s : S, ∃ n, (s : M) ∈ N n := fun s =>
    (Submodule.mem_iSup_of_chain N s).mp
      (by
        rw [H, ← hS]
        exact Submodule.subset_span s.2)
  choose f hf using this
  use S.attach.sup f
  apply le_antisymm
  · conv_lhs => rw [← hS]
    rw [Submodule.span_le]
    intro s hs
    exact N.2 (Finset.le_sup <| S.mem_attach ⟨s, hs⟩) (hf _)
  · rw [← H]
    exact le_iSup _ _
```

▼ Finiteness.lean:365:2

▼ Tactic state

1 goal

▼ **case** intro

```
R : Type u_1
M : Type u_2
inst✝² : Semiring R
inst✝¹ : AddCommMonoid M
inst✝ : Module R M
M' : Submodule R M
N : ℕ →o Submodule R M
H : iSup ⇑N = M'
S : Finset M
hS : span R ↑S = M'
f : { x // x ∈ S } → ℕ
hf : ∀ (s : { x // x ∈ S }), ↑s ∈ N (f s)
⊢ ∃ n, M' = N n
```

Mathlib > RingTheory > Finiteness.lean

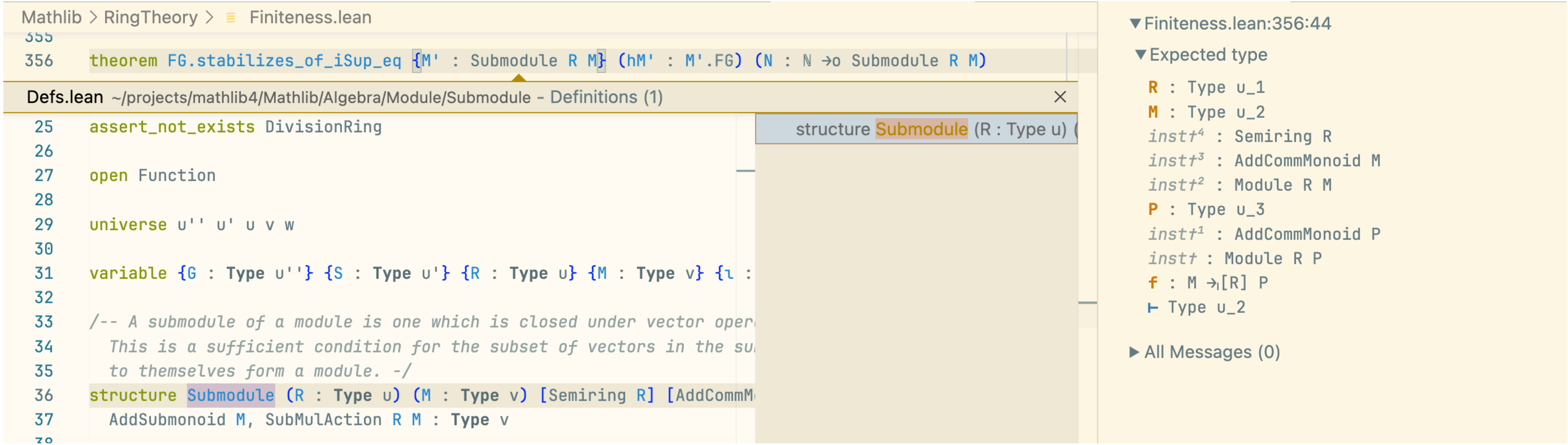
```lean
theorem FG.stabilizes_of_iSup_eq {M' : Submodule R M} (hM' : M'.FG) (N : ℕ →o Submodule R M)
    (H : iSup N = M') : ∃ n, M' = N n := by
  obtain ⟨S, hS⟩ := hM'
  have : ∀ s : S, ∃ n, (s : M) ∈ N n := fun s =>
    (Submodule.mem_iSup_of_chain N s).mp
      (by
        rw [H, ← hS]
        exact Submodule.subset_span s.2)
  choose f hf using this
  use S.attach.sup f
  apply le_antisymm
  · conv_lhs => rw [← hS]
    rw [Submodule.span_le]
    intro s hs
    exact N.2 (Finset.le_sup <| S.mem_attach ⟨s, hs⟩) (hf _)
  · rw [← H]
    exact le_iSup _ _
```

▼ Finiteness.lean:365:2

▼ Tactic state

1 goal

```
▼ case intro
R : Type u_1
M : Type u_2
inst✝² : Semiring R
inst✝¹ : AddCommMonoid M
inst✝ : Module R M
M' : Submodule R M
N : ℕ →o Submodule R M
H : iSup ⇑N = M'
S : Finset M
hS : span R ↑S = M'
f : { x // x ∈ S } → ℕ
hf : ∀ (s : { x // x ∈ S }), ↑s ∈ N (f s)
⊢ ∃ n, M'   M' : Submodule R M
```



Mathlib > RingTheory > Finiteness.lean
356 theorem FG.stabilizes_of_iSup_eq {M' : Submodule R M} (hM' : M'.FG) (N : ℕ →o Submodule R M)
Defs.lean ~/projects/mathlib4/Mathlib/Algebra/Module/Submodule - Definitions (1)
25 assert_not_exists DivisionRing
27 open Function
29 universe u'' u' u v w
31 variable {G : Type u''} {S : Type u'} {R : Type u} {M : Type v} {ι :
33 /-- A submodule of a module is one which is closed under vector oper
34 This is a sufficient condition for the subset of vectors in the su
35 to themselves form a module. -/
36 structure Submodule (R : Type u) (M : Type v) [Semiring R] [AddCommM
37 AddSubmonoid M, SubMulAction R M : Type v
structure Submodule (R : Type u)
▼Finiteness.lean:356:44
▼Expected type
R : Type u_1
M : Type u_2
inst✝⁴ : Semiring R
inst✝³ : AddCommMonoid M
inst✝² : Module R M
P : Type u_3
inst✝¹ : AddCommMonoid P
inst✝ : Module R P
f : M →ₗ[R] P
⊢ Type u_2
▶ All Messages (0)

Mathlib > Algebra > Module > Submodule > Defs.lean > Submodule

```lean
  This is a sufficient condition for the subset of vectors in the submodule
  to themselves form a module. -/
structure Submodule (R : Type u) (M : Type v) [Semiring R] [AddCommMonoid M] [Module R M] extends
  AddSubmonoid M, SubMulAction R M : Type v
```

Defs.lean ~/projects/mathlib4/Mathlib/Algebra/Group/Submonoid - Definitions (1)

```lean
add_decl_doc Submonoid.toSubsemigroup

/-- `SubmonoidClass S M` says `S` is a type of subsets `s ≤ M` that
and are closed under `(*)` -/
class SubmonoidClass (S : Type*) (M : outParam Type*) [MulOneClass M
  MulMemClass S M, OneMemClass S M : Prop

section

/-- An additive submonoid of an additive monoid `M` is a subset cont
  closed under addition. -/
structure AddSubmonoid (M : Type*) [AddZeroClass M] extends AddSubse
  /-- An additive submonoid contains `0`. -/
  zero_mem' : (0 : M) ∈ carrier
```

structure AddSubmonoid (M : Type

▼ Defs.lean:37:8

▼ Expected type

```
G : Type u''
S : Type u'
R† : Type u
M† : Type v
ι : Type w
R : Type u
M : Type v
inst†² : Semiring R
inst†¹ : AddCommMonoid M
inst† : Module R M
⊢ Type v
```

▶ All Messages (0)

▼ *Example: Using* `simp?`

The non-terminal `simp?` in this proof suggests a smaller `simp` with `only`:

```
example (xs : Array Unit) : xs.size = 2 → xs = #[(), ()] := by
  intros
  ext
  simp?
  assumption
```

The suggested rewrite is:

```
Try this: simp only [List.size_toArray, List.length_cons, List.length_n
```

```
▼ case h₁
xs : Array Unit
a✝ : xs.size = 2
⊢ xs.size = #[(), ()].size
```

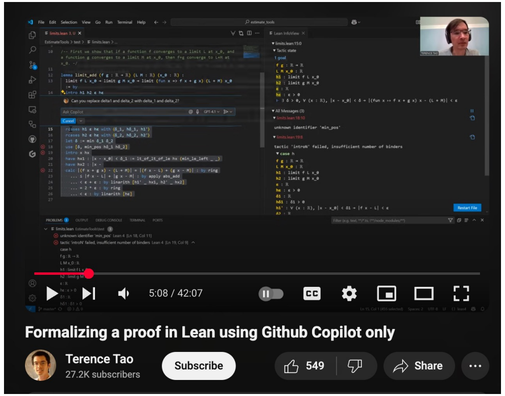
This results in the more maintainable proof:

```
example (xs : Array Unit) : xs.size = 2 → xs = #[(), ()] := by
  intros
  ext
  simp only [List.size_toArray, List.length_cons, List.length_nil, Nat.zer
  assumption
```

Mathematics

Lean is Taking Mathematics by Storm

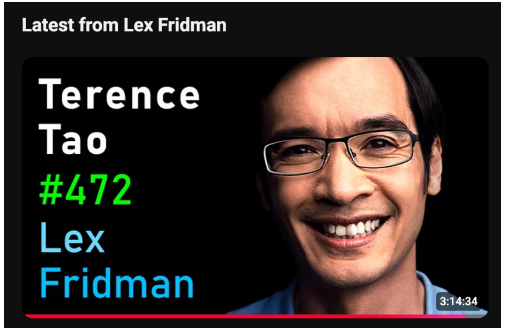
*"**Lean enables large-scale collaboration** by allowing mathematicians to break down complex proofs into smaller, verifiable components. This formalization process ensures the correctness of proofs and facilitates contributions from a broader community. **With Lean, we are beginning to see how AI can accelerate the formalization of mathematics, opening up new possibilities for research.**" — Terence Tao*



Fermat's Last Theorem – Kevin Buzzard

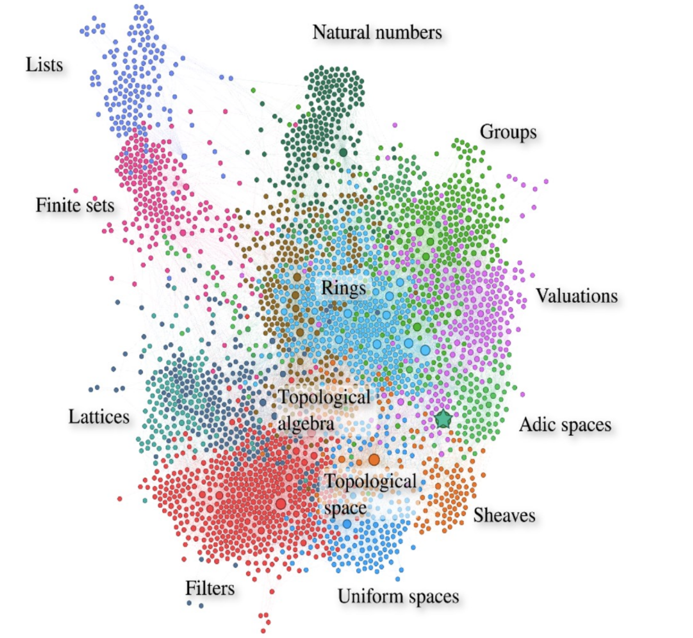
Carleson's Theorem – Floris van Doorn

How did we get here?



Preamble: the Perfectoid Spaces Project (cont.)

Kevin Buzzard, Patrick Massot, Johan Commelin



mathoverflow

Home

What are "perfectoid spaces"?

72

Here is a completely different kind of answer to this question.

A *perfectoid space* is a term of type `PerfectoidSpace` in the Lean theorem prover.

Here's a quote from the source code:

```
structure perfectoid_ring (R : Type) [Huber_ring R] extends Tate_ring R : Prop :=
(complete  : is_complete_hausdorff R)
(uniform   : is_uniform R)
(ramified  : ∃ ϖ : pseudo_uniformizer R, ϖ^p ∣ p in R°)
(Frobenius : surjective (Frob R°/p))
```

The Challenge

In November of 2020, Peter Scholze posits the Liquid Tensor Experiment (LTE) challenge.

> *“I spent much of 2019* ***obsessed*** *with the proof of this theorem,* ***almost getting crazy over it****. In the end, we were able to get an argument pinned down on paper, but I think nobody else has dared to look at the details of this, and so I still have some small lingering doubts”*,
>
> Peter Scholze

The First Victory

Johan Commelin led a team with several members of the **Lean community and announced the formalization of the crucial intermediate lemma** that Scholze was unsure about, with only minor corrections, in **May 2021**.

> *"[T]his was precisely the kind of oversight I was worried about when I asked for the formal verification. […] The proof walks a fine line, so if some argument needs constants that are quite a bit different from what I claimed, it might have collapsed"*, Peter Scholze


nature

Explore content ⌄ Journal information ⌄ Publish with us ⌄ Subscribe

nature > news > article

NEWS | 18 June 2021

Mathematicians welcome computer-assisted proof in 'grand unification' theory

Achieving the Unthinkable

The full challenge was completed in July 2022.

The team not only verified the proof but also simplified it.
Moreover, they did this without fully understanding the entire proof.

Johan, the project lead, reported that he could only see two steps ahead. **Lean was a guide**.

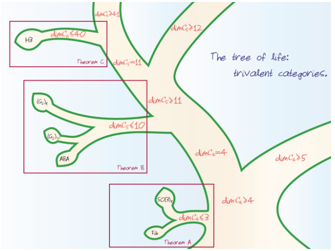
> *"The Lean Proof Assistant was really that: an assistant in navigating through the thick jungle that this proof is. Really, one key problem I had when I was trying to find this proof was that I was essentially unable to keep all the objects in my RAM, and I think the same problem occurs when trying to read the proof"*, Peter Scholze

Automating Quantum Algebra

Here is a concrete example from quantum algebra. It comes from a classification result involving quantum SO(3) categories. Specifically, the condition that certain relations among trivalent graphs imply a constraint on the parameters d, t, and c:

```
example {α} [CommRing α] [IsCharP α 0] (d t c : α) (d_inv PSO3_inv : α)
  (Δ40 : d^2 * (d + t - d * t - 2) *
    (d + t + d * t) = 0)
  (Δ41 : -d^4 * (d + t - d * t - 2) *
    (2 * d + 2 * d * t - 4 * d * t^2 + 2 * d * t^4 + 2 * d^2 * t^4 - c * (d + t + d * t)) = 0)
  (_ : d * d_inv = 1)
  (_ : (d + t - d * t - 2) * PSO3_inv = 1) :
  t^2 = t + 1 := by grind
```

From: “Categories generated by a trivalent vertex”, Morrison, Peters, and Snyder



Automating Quantum Algebra

```
example {α} [CommRing α] [IsCharP α 0] (d t c : α) (d_inv PSO3_inv : α)
  (Δ40 : d^2 * (d + t - d * t - 2) *
    (d + t + d * t) = 0)
  (Δ41 : -d^4 * (d + t - d * t - 2) *
    (2 * d + 2 * d * t - 4 * d * t^2 + 2 * d * t^4 + 2 * d^2 * t^4 - c * (d + t + d * t)) = 0)
  (_ : d * d_inv = 1)
  (_ : (d + t - d * t - 2) * PSO3_inv = 1) :
  t^2 = t + 1 := by grind
```

This is not a toy: it encodes a real algebraic constraint derived from relations among diagrams in a pivotal tensor category.

Lean can handle this kind of reasoning automatically, in milliseconds.

Automating Quantum Algebra

We can explore new mathematical and physical structures, from topological quantum fields theories to fusion categories.

Lean is helping researchers reason reliably about complex symbolic systems that were previously handled only by hand or with unverified computer algebra.

Reasoning at the right level of abstraction

“I'm interested in developing some API for linearly ordered vector spaces, in order to easily handle manipulations of asymptotic orders” – Terence Tao on the Lean Zulip

```
example {R} [OrderedVectorSpace R] (x y z : R)
        : x ≤ 2•y → y < z → x < 2•z := by
 grind -- 🎉
```

OrderedVectorSpace implements IntModule, LinearOrder, IntModule.IsOrdered.

Software

Lean in Software Verification

Lean is a programming language, and is used in **many software verification projects.**

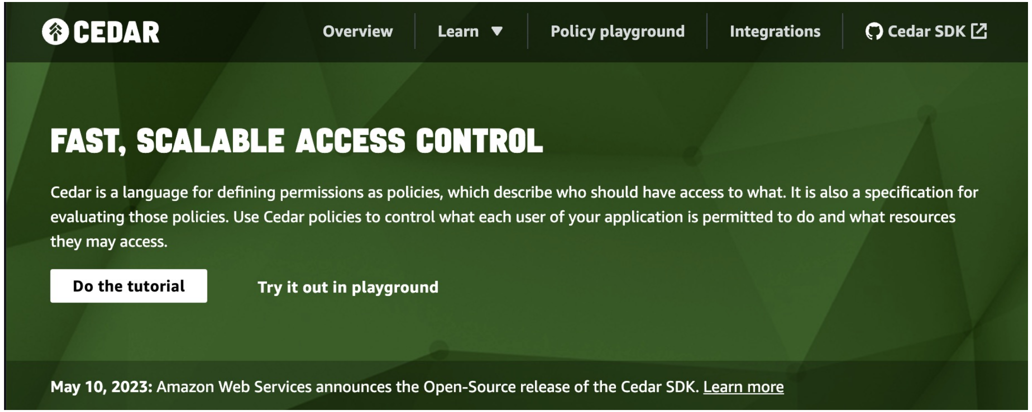
You can write code and reason about it simultaneously.

You can prove that your code has the properties you expect.

"Testing can show the presence of bugs, but not their absence", E. Dijkstra

Cedar

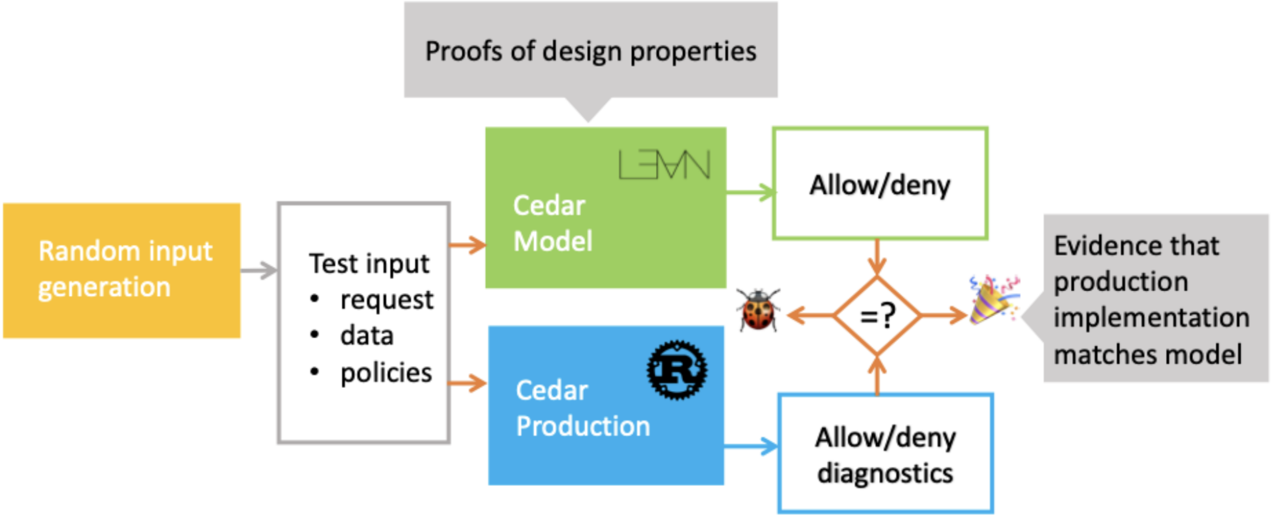
https://www.cedarpolicy.com/



https://github.com/cedar-policy/cedar-spec

```
def isAuthorized (req : Request) (entities : Entities) (policies : Policies) : Response :=
  let forbids := satisfiedPolicies .forbid policies req entities
  let permits := satisfiedPolicies .permit policies req entities
  let erroringPolicies := errorPolicies policies req entities
  if forbids.isEmpty && !permits.isEmpty
  then { decision := .allow, determiningPolicies := permits, erroringPolicies }
  else { decision := .deny,  determiningPolicies := forbids, erroringPolicies }
```

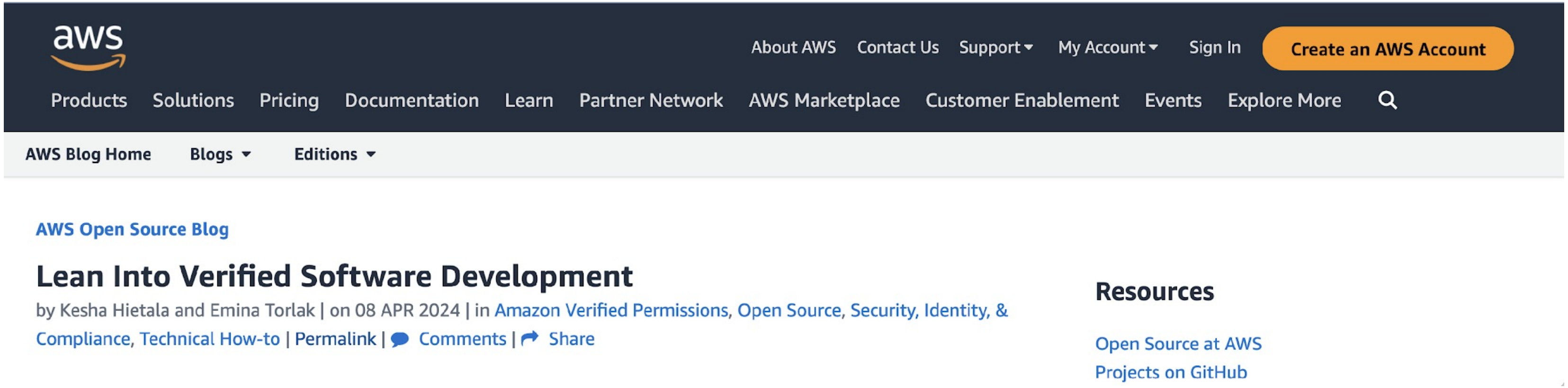
Cedar



*"**Lean is the core verification technology behind Cedar**, the open-source authorization language that powers cloud services like Amazon Verified Permissions and AWS Verified Access. Our team rigorously formalizes and verifies core components of Cedar using Lean's proof assistant, and we leverage **Lean's lightning-fast runtime** to continuously test our production Rust code against the Lean formalization. Lean's efficiency, extensive libraries, and vibrant community **enable us to develop and maintain Cedar at scale**, while ensuring the key correctness and security properties that our users depend on." — Emina Torlak, Senior Principal Applied Scientist, AWS*

Cedar

To learn more about Cedar:
https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/



SampCert

An **open-source** Lean library of formally **verified differential privacy primitives**.

The implementation is not only verified, but it is also **twice as fast as the previous one**.

Paper at this PLDI.

SampCert would not exist without Mathlib

SampCert is software, but its verification relies heavily on Mathlib.

The verification of code addressing practical problems in data privacy depends on the formalization of mathematical concepts, from **Fourier analysis** to **number theory** and **topology**.

“For SampCert, I started using Lean because of Mathlib, but I realized that Lean isn't just an excellent proof assistant, it's also a very pleasant and efficient programming language with a great ecosystem. As a result, we continued using Lean for TenCert.” Jean-Baptiste Tristan

Verifying Cryptography with Aeneas at Microsoft

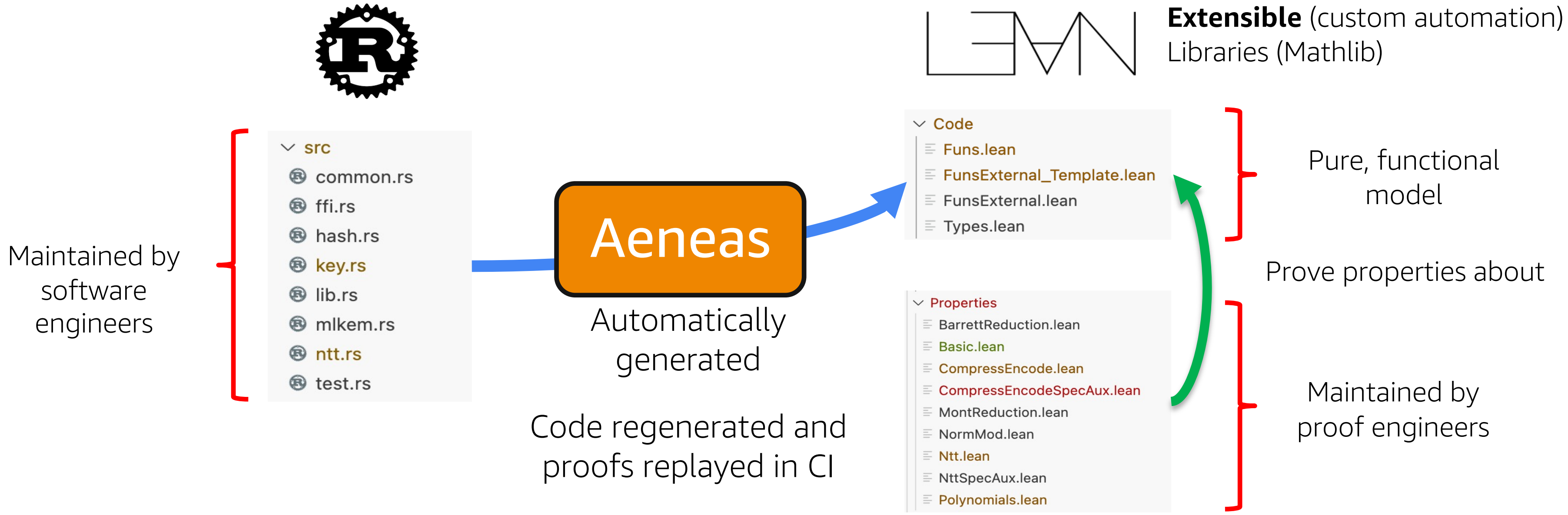
They verify (and fix/improve) the Rust code as written by software engineers.

Code is evolving (new optimizations for specific hardware): They must adapt to rewrites.

Rewriting SymCrypt in Rust to modernize Microsoft's cryptographic library.

*“**The verification crucially relies on the Lean** interactive theorem prover, whose **extensibility** has been **key** in developing custom automation to make verification amenable in an industrial setting. Lean FRO” – Son Ho*

Verifying Cryptography with Aeneas at Microsoft



Lean Extensions in Aeneas

```
syntax (name := zmodify) "zmodify" ("to" term)? ("[" (term<|>"*"),* "]")? (location)? : tactic

def parseZModify : TSyntax ``zmodify -> TacticM (Option Expr × ScalarTac.CondSimpPartialArgs × Utils.Location)
  | `(tactic| zmodify $[to $n]? $[[$args,*]]?) => do
    let n ← Utils.optElabTerm n
    let args := args.map (·.getElems) |>.getD #[]
    let args ← ScalarTac.condSimpParseArgs "zmodify" args
    pure (n, args, Utils.Location.targets #[] true)
  | `(tactic| zmodify $[to $n]? $[[$args,*]]? $[$loc:location]?) => do
    let n ← Utils.optElabTerm n
    let args := args.map (·.getElems) |>.getD #[]
    let args ← ScalarTac.condSimpParseArgs "zmodify" args
    let loc ← Utils.parseOptLocation loc
    pure (n, args, loc)
  | _ => Lean.Elab.throwUnsupportedSyntax
```

You don’t need to learn a new programming language to extend Lean

Lean Extensions in Aeneas

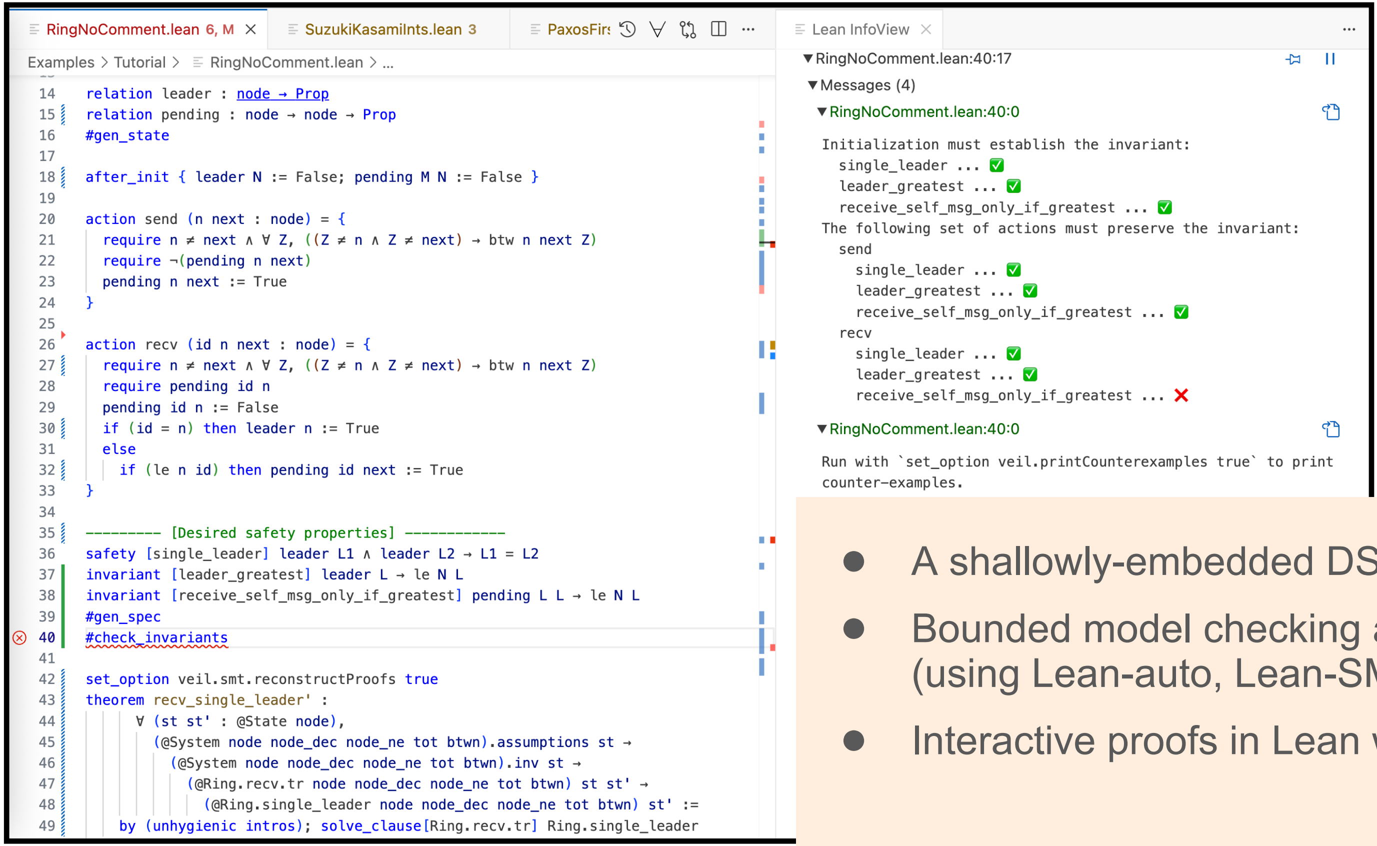
```
/-- The `scalar_tac_simps` simp attribute. -/
initialize scalarTacSimpExt : SimpExtension ←
  registerSimpAttr `scalar_tac_simps "\
    The `scalar_tac_simps` attribute registers simp lemmas to be used by `scalar_tac`
    during its preprocessing phase."
```

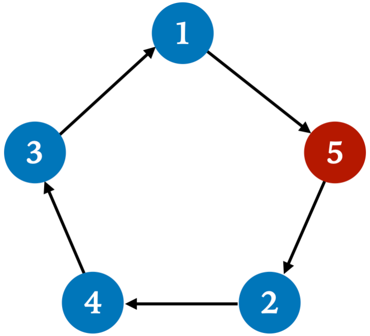
```
/-- A simpproc to reduce expressions of the shape: `Fin.val (6 : Fin 7)` -/
simproc reduceFinOfNatVal (@Fin.val _ _) := fun e => do
  trace[ReduceFin] "- e: {e}\n"
  match e.consumeMData.getAppFnArgs with
  | (``Fin.val, #[finTy, value]) =>
    trace[ReduceFin] "- finTy: {finTy}\n- value: {value}\n"
    -- Small helper
    let extractOfNatValue (e : Expr) : Option Nat :=
      match e.consumeMData.getAppFnArgs with
      | (``OfNat.ofNat, #[_, value, _]) =>
        match exprToNat? value with
        | none => none
        | some value => some value
      | _ => none
```

Parallel Tactic Execution: Critical for Users Like Those on Aeneas

```
theorem t1 : True := by
  sleep 1000
  sleep 1000
  trivial

theorem t2 : True := by
  sleep 1100
  sleep 1100
  trivial

theorem t3 : True := by
  sleep 1200
  sleep 1200
  trivial

theorem t4 : True := by
  sleep 1200
  sleep 1200
  trivial
```

Veil: Multi-Modal Verifier for Distributed Protocols





There is *at most one leader.*

- A shallowly-embedded DSL in Lean
- Bounded model checking and automation via SMT (using Lean-auto, Lean-SMT)
- Interactive proofs in Lean when automation fails

github.com/verse-lab/veil

Pîrlea et al., CAV'25

Splean: a Simple Separation Logic in Lean

Misc.lean ↓M, M | Basic.lean | HProp.lean | SepLog.lean ↓M

SPLean > Experiments > Misc.lean > {} find_index > {} Lang

```
section find_index
namespace Lang

lang_def add_pointer :=
  fun p q =>
    let m := !q in
    for i in [0:m] {
      ++p
    }; !p

lemma add_pointer_spec (p q : loc) (n m : Int) (_ : m >= 0) :
  { p ~~> n ∗ q ~~> m }
  [ add_pointer p q ]
  { v, ⌜v = n+m⌝ ∗ ⊤  } := by
    xstep
    -- Tactic for a [for]-loop. This tactic should be supplied with a loop
    -- invariant [I]. In this case [I] only captures a piece of the state,
    -- relevant to the loop body. The rest of the state would be framed
    -- automatically
    xfor (fun i => p ~~> n + i)
    { move=>*;
      xapp; -- Here rather than symbolically executing the
            -- top-most instruction in [incr], we apply its
            -- specification lemma directly via [xapp] tactic
      xsimp; omega }
    move=> ? /=; xapp; xsimp
```

Lean InfoView

▼ Misc.lean:65:9

▼ Tactic state

1 goal

```
▼ case xsimp_goal.a.a.a
p q : loc
n m : ℤ
x✝ : m ≥ 0
⊢ { q ~~> m ∗ p ~~> n }
  [for i in [0 : m] {
        ++p };
     !p]
  { v, ⌜v = (n + m)⌝ ∗ ⊤ }
```

▶ All Messages (1)

Restart File

github.com/verse-lab/splean

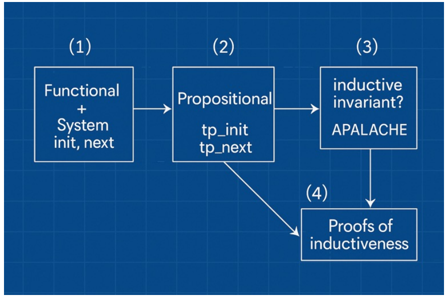
More Protocol Verification in Lean

"No other interactive theorem prover captured my attention for so long." Igor Konnov

[Specifying and simulating two-phase commit in Lean4]

[Proving consistency of two-phase commit in Lean4]

[Proving completeness of an eventually perfect failure detector in Lean4]



"Of course, this is all done through "monads", but they are relatively easy to use in Lean — even if you are not quite ready to buy into the FP propaganda. As a bonus point, this simulator is really fast." Igor Konnov

KLR: a language and elaborators for machine learning kernels

Define a common representation for kernel functions with a precise formal semantics along with translations from common kernel languages to the KLR core language.

"The lean meta programming is amazing. Have managed to delete hundreds of lines of boilerplate in the last couple days." Sean McLaughlin

KLR is also open source.

```
private def evalTensorScalar (ts : TensorScalar) (t: ByteArray) : Err ByteArray := do
  match ts with
  | TensorScalar.mk op0 c0 rev0 op1 c1 rev1 =>
  let f0 <- evalAluOp op0
  let f1 <- evalAluOp op1
  let c0 := c0.toLEByteArray
  let c1 := c1.toLEByteArray
  apply2 f0 rev0 c0 f1 rev1 c1 t
```

KLR: a language and elaborators for machine learning kernels

KLR uses bit-vectors, fixed integers, etc.

```
private def decBV64 : DecodeM (BitVec 64) :=
  let u8_64 : DecodeM UInt64 := next >>= fun x => return x.toUInt64
  return ((<- u8_64) <<< 0  |||
          (<- u8_64) <<< 8  |||
          (<- u8_64) <<< 16 |||
          (<- u8_64) <<< 24 |||
          (<- u8_64) <<< 32 |||
          (<- u8_64) <<< 40 |||
          (<- u8_64) <<< 48 |||
          (<- u8_64) <<< 56).toBitVec
```

bv_decide: another powerful move

A verified bit-blaster by **Henrik Boving**, Josh Clune, Siddharth Bhat, and Alex Keizer

Uses LRAT proof producing SAT solvers: **Cadical**

```
/-
Close a goal by:
1. Turning it into a BitVec problem.
2. Using bitblasting to turn that into a SAT problem.
3. Running an external SAT solver on it and obtaining an LRAT proof from it.
4. Verifying the LRAT proof using proof by reflection.
-/
syntax (name := bvDecideSyntax) "bv_decide" : tactic
```

“Blasting” popcount with bv_decide

```
def popcount : Stmt := imp {
  x := x - ((x >>> 1) &&& 0x55555555);
  x := (x &&& 0x33333333) + ((x >>> 2) &&& 0x33333333);
  x := (x + (x >>> 4)) &&& 0x0F0F0F0F;
  x := x + (x >>> 8);
  x := x + (x >>> 16);
  x := x &&& 0x0000003F;
}
```

```
def pop_spec (x : BitVec 32) : BitVec 32 :=
  go x 0 32
where
  go (x : BitVec 32) (pop : BitVec 32) (i : Nat) : BitVec 32 :=
    match i with
    | 0 => pop
    | i + 1 =>
      let pop := pop + (x &&& 1#32)
      go (x >>> 1#32) pop i
```

```
theorem popcount_correct :
    ∃ ρ, (run (Env.init x) popcount 8) = some ρ ∧ ρ "x" = pop_spec x := by
  simp [run, popcount, Expr.eval, Expr.BinOp.apply, Env.set, Value, pop_spec, pop_spec.go]
  bv_decide
```

“Blasting” popcount with bv_decide

Imp.lean > { } Imp.Stmt > popcount_correct

```
theorem popcount_correct :
    ∃ p, (run (Env.init x) popcount 8) = some p
  simp [run, popcount, Expr.eval, Expr.BinOp.app
  bv_decide
```

▼ Tactic state

1 goal

```
x : Value
⊢ ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) + ((x - (x >>> 1 &&&
1431655765#32)) >>> 2 &&& 858993459#32) +
            ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32)) >>>
              4 &&&
          252645135#32) +
        ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32) +
              ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                  ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32)) >>>
                4 &&&
            252645135#32) >>>
          8 +
      (((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32) +
              ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                  ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32)) >>>
                4 &&&
            252645135#32) +
          ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                  ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32) +
                ((x - (x >>> 1 &&& 1431655765#32) &&& 858993459#32) +
                    ((x - (x >>> 1 &&& 1431655765#32)) >>> 2 &&& 858993459#32)) >>>
                  4 &&&
              252645135#32) >>>
            8) >>>
        16 &&&
```

Does Lean Have Hammers?

The Lean community is also actively developing automation.

LeanHammer: an automated reasoning tool for Lean which brings together multiple proof search and reconstruction techniques and combine them into one tool.

Lean-SMT: An SMT tactic for discharging proof goals in Lean

"Improving automation for proofs in Lean is an exciting research direction. ***Lean-SMT aims to improve automation by enabling the automatic replay in Lean of proof certificates produced*** *by SMT solvers."*
Clark Barrett

grind (again)

```
example (x : BitVec 16) : (x + 256)*(x - 256) = x^2 := by
  grind
```

```
def siftDown (a : Array Int) (root : Nat) (e : Nat) (h : e ≤ a.size := by grind) : Array Int :=
  if _ : leftChild root < e then
    let child := leftChild root
    let child := if _ : child+1 < e then
      if a[child] < a[child + 1] then child + 1 else child
    else child
    if a[root] < a[child] then
      let a := a.swap root child
      siftDown a child e
    else a
  else a
termination_by e - root

theorem siftDown_size {a root e h} : (siftDown a root e h).size = a.size := by
   fun_induction siftDown <;> grind [siftDown]
```

grind diagnostics at your fingertips

```
example {α} (as bs cs : Array α) (v₁ v₂ : α)
        (i₁ i₂ j : Nat)
        (h₁ : i₁ < as.size)
        (h₂ : bs = as.set i₁ v₁)
        (h₃ : i₂ < bs.size)
        (h₃ : cs = bs.set i₂ v₂)
        (h₄ : i₁ ≠ j)
        (h₅ : j < cs.size)
        (h₆ : j < as.size)
        : cs[j] = as[j] := by
  grind
```

```
`grind` failed
▼case grind
α : Type u_1
as bs cs : Array α
v₁ v₂ : α
i₁ i₂ j : Nat
h₁ : i₁ + 1 ≤ as.size
h₂ : bs = as.set i₁ v₁ ⋯
h₃ : i₂ + 1 ≤ bs.size
h₃_1 : cs = bs.set i₂ v₂ ⋯
h₄ : ¬i₁ = j
h₅ : j + 1 ≤ cs.size
h₆ : j + 1 ≤ as.size
h : ¬cs[j] = as[j]
⊢ False

[grind] Goal diagnostics ▼
  [facts] Asserted facts ▶
  [eqc] True propositions ▶
  [eqc] False propositions ▶
  [eqc] Equivalence classes ▶
  [ematch] E-matching patterns ▶
  [cutsat] Assignment satisfying linear constraints ▼
    [assign] i₁ := 0
    [assign] i₂ := 1
    [assign] j := 1
    [assign] as.size := 2
    [assign] bs.size := 2
    [assign] cs.size := 2
```

grind diagnostics at your fingertips

```
example {α} (as bs cs : Array α) (v₁ v₂ : α)
        (i₁ i₂ j : Nat)
        (h₁ : i₁ < as.size)
        (h₂ : bs = as.set i₁ v₁)
        (h₃ : i₂ < bs.size)
        (h₃ : cs = bs.set i₂ v₂)
        (h₄ : i₁ ≠ j)
        (h₅ : j < cs.size)
        (h₆ : j < as.size)
        : cs[j] = as[j] := by
  grind
```

```
[grind] Goal diagnostics ▼
  [facts] Asserted facts ▶
  [eqc] True propositions ▶
  [eqc] False propositions ▼
    [prop] i₁ = j
    [prop] cs[j] = as[j]
    [prop] ¬i₂ = j
    [prop] (bs.set i₂ v₂ ⋯)[j] = bs[j]
  [eqc] Equivalence classes ▶
  [ematch] E-matching patterns ▶
  [cutsat] Assignment satisfying linear constraints ▶
  [limits] Thresholds reached ▶

[grind] Issues ▶

[grind] Diagnostics ▼
  [thm] E-Matching instances ▼
    [] Array.getElem_set_ne ↦ 2
    [] Array.size_set ↦ 2
    [] Array.getElem_set_self ↦ 1
```

grind diagnostics at your fingertips

```
example {α} (as bs cs : Array α) (v₁ v₂ : α)
        (i₁ i₂ j : Nat)
        (h₁ : i₁ < as.size)
        (h₂ : bs = as.set i₁ v₁)
        (h₃ : i₂ < bs.size)
        (h₃ : cs = bs.set i₂ v₂)
        (h₄ : i₁ ≠ j)
        (h₅ : j < cs.size)
        (h₆ : j < as.size)
        : cs[j] = as[j] := by
  grind
```

```
[grind] Goal diagnostics ▼
  [facts] Asserted facts ▶
  [eqc] True propositions ▶
  [eqc] False propositions ▼
    [prop] i₁ = j
    [prop] cs[j] = as[j]
    [prop] ¬i₂ = j
    [prop] (bs.set i₂ v₂ ⋯)[j] = bs[j]
  [eqc] Equivalence classes ▶
  [ematch] E-matching patterns ▶
  [cutsat] Assignment satisfying linear constraints ▶
  [limits] Thresholds reached ▶
```

```
@Array.getElem_set_ne : ∀ {α : Type u_1} {xs : Array α} {i : Nat} (h'
: i < xs.size) {v : α} {j : Nat} (pj : j < xs.size),
  i ≠ j → (xs.set i v h')[j] = xs[j]
```

```
    [] Array.getElem_set_ne ↦ 2
    [] Array.size_set ↦ 2
    [] Array.getElem_set_self ↦ 1
```

What is grind?

A proof-automation tactic **inspired by modern SMT solvers**. Think of it as a **virtual whiteboard**:

Discovers new equalities, inequalities, etc.

Writes facts on the board and merges equivalent terms

Multiple engines cooperate on the same workspace

Cooperating Engines:

Congruence closure; E-matching; Constraint propagation; Guided case analysis

Satellite theory solvers (linear integer arithmetic, commutative rings, linear arithmetic)

Supports dependent types, type-class system, and dependent pattern matching

Produces ordinary Lean proof terms for every fact

What grind is NOT

Not designed for combinatorially explosive search spaces:

- Large-n pigeonhole instances
- Graph-coloring reductions
- High-order N-queens boards
- 200-variable Sudoku with Boolean constraints

Why? These require thousands/millions of case-splits that overwhelm grind's branching search

Key takeaway: grind excels at cooperative reasoning with multiple engines, but struggles with brute-force combinatorial problems.

For massive case-analysis, use bv_decide

"if-normalization" challenge by Leino, Merz, and Shankar

```
def normalize (assign : Std.HashMap Nat Bool) : IfExpr → IfExpr
  | lit b => lit b
  | var v =>
    match assign[v]? with
    | none => var v
    | some b => lit b
  | ite (lit true)  t _ => normalize assign t
  | ite (lit false) _ e => normalize assign e
  | ite (ite a b c) t e => normalize assign (ite a (ite b t e) (ite c t e))
  | ite (var v)     t e =>
    match assign[v]? with
    | none =>
      let t' := normalize (assign.insert v true) t
      let e' := normalize (assign.insert v false) e
      if t' = e' then t' else ite (var v) t' e'
    | some b => normalize assign (ite (lit b) t e)
  termination_by e => e.normSize

-- We tell `grind` to unfold our definitions above.
attribute [local grind] normalized hasNestedIf hasConstantIf hasRedundantIf disjoint vars eval List.disjoint

theorem normalize_spec (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
    ∧ (∀ f, (normalize assign e).eval f = e.eval fun w => assign[w]?.getD (f w))
    ∧ ∀ (v : Nat), v ∈ vars (normalize assign e) → ¬ v ∈ assign := by
  fun_induction normalize with grind
```

"if-normalization" challenge by Leino, Merz, and Shankar

Interactive tactic suggestion tool: the `try?` tactic

It tries many different tactics, guesses induction principle, and is **extensible**

```
theorem normalize_spec (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
    ∧ (∀ f, (normalize assign e).eval f = e.eval fun w => assign[w]?.getD (f w))
    ∧ ∀ (v : Nat), v ∈ vars (normalize assign e) → ¬ v ∈ assign := by
  try?
```

Lean Infoview

▼Suggestions

```
Try these:

• fun_induction normalize <;> grind
• fun_induction normalize <;>
    grind only [vars, normalized, disjoint, =_ Std.HashMap.contains_iff_mem, =_
      List.contains_iff_mem, List.contains_eq_mem, hasNestedIf, hasConstantIf, hasRedundantIf,
      List.elem_nil, eval, cases Or, List.contains_cons, List.eq_or_mem_of_mem_cons,
      Option.getD_none, List.mem_cons_of_mem, getElem?_pos, getElem?_neg, Option.getD_some, =
      Std.HashMap.mem_insert, = Std.HashMap.getElem?_insert, = Std.HashMap.getElem_insert, =
      Std.HashMap.contains_insert, =_ List.cons_append, = List.append_assoc, = List.contains_append,
      List.nil_append, List.disjoint, List.append_nil, = List.cons_append, =_ List.append_assoc, →
      List.eq_nil_of_append_eq_nil, List.mem_append]
```

AI

Lean Enables Verified AI for Mathematics and Code

LLMs are powerful tools, but they are prone to **hallucinations**.

In Math, a **small mistake can invalidate the whole proof**.

Imagine manually checking an AI-generated proof with the size and complexity of FLT.

The informal proof is **over 200 pages**.

Buzzard estimates a formal proof will require more than **1M LoC** on top of Mathlib.

Machine-checkable proofs are the antidote to hallucinations.

Synthetic Data Generation

LLMs require **vast amounts of data** for training.

Lean mathematical libraries provide valuable, **correct-by-construction training data**.

AILean, a project led by **Soonho Kong** at AWS, uses Lean to generate **new synthetic theorems** that are correct by construction.

Pantograph by Leni Aniva (Stanford) is also getting very popular in the Lean community.

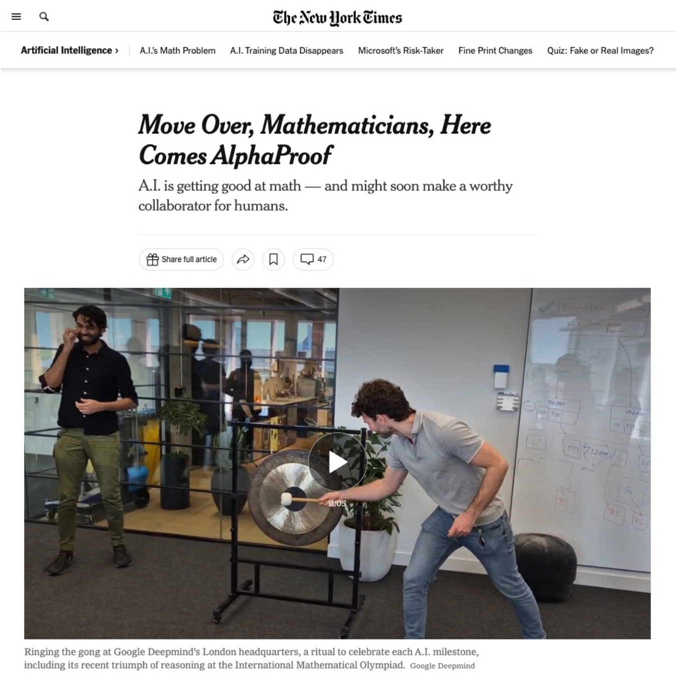
AI Proof Assistants

Several groups are developing AI that suggests the **next move**(s) in Lean's interactive proof game.

LeanDojo is an open-source project from Caltech, and everything (model, datasets, code) is open.

OpenAI and Meta AI have also developed AI assistants for Lean.

Claude 4 is fantastic on Lean code. Their System Card contains a Lean example.


The New York Times

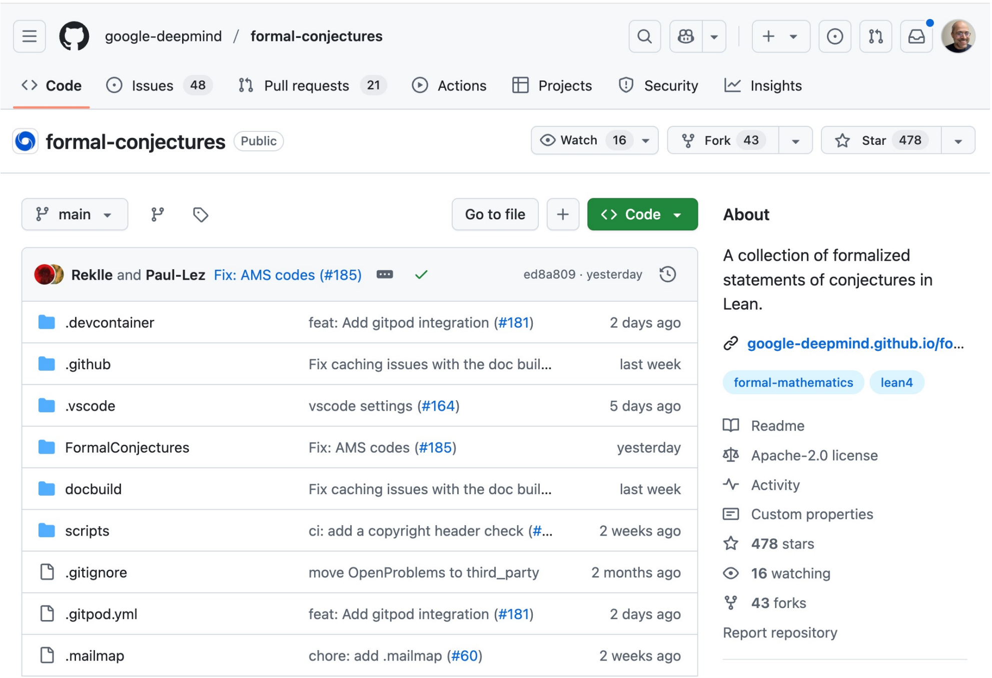
Artificial Intelligence › | A.I.'s Math Problem | A.I. Training Data Disappears | Microsoft's Risk-Taker | Fine Print Changes | Quiz: Fake or Real Images?

Move Over, Mathematicians, Here Comes AlphaProof

A.I. is getting good at math — and might soon make a worthy collaborator for humans.

Share full article | 47

Ringing the gong at Google Deepmind's London headquarters, a ritual to celebrate each A.I. milestone, including its recent triumph of reasoning at the International Mathematical Olympiad. Google Deepmind


google-deepmind / formal-conjectures

Code | Issues 48 | Pull requests 21 | Actions | Projects | Security | Insights

formal-conjectures Public

Watch 16 | Fork 43 | Star 478

main | Go to file | + | Code

Reklle and Paul-Lez Fix: AMS codes (#185) ✓ ed8a809 · yesterday

.devcontainer	feat: Add gitpod integration (#181)	2 days ago
.github	Fix caching issues with the doc buil...	last week
.vscode	vscode settings (#164)	5 days ago
FormalConjectures	Fix: AMS codes (#185)	yesterday
docbuild	Fix caching issues with the doc buil...	last week
scripts	ci: add a copyright header check (#...	2 weeks ago
.gitignore	move OpenProblems to third_party	2 months ago
.gitpod.yml	feat: Add gitpod integration (#181)	2 days ago
.mailmap	chore: add .mailmap (#60)	2 weeks ago

About

A collection of formalized statements of conjectures in Lean.

google-deepmind.github.io/fo...

formal-mathematics | lean4

Readme
Apache-2.0 license
Activity
Custom properties
478 stars
16 watching
43 forks
Report repository

"At Google DeepMind, we used Lean to build AlphaProof, a new reinforcement-learning based system for formal math reasoning. ***Lean's extensibility and verification capabilities were key in enabling the development of AlphaProof****." — Pushmeet Kohli, Vice President, Research Google DeepMind*

Auto-formalization

The process of converting natural language into a formal language like Lean.

Bhavik Mehta • 1st
Chapman Fellow in Mathematics at Imperial College Lo...
4d • Edited •

Thrilled to share a major milestone from Big Proof in Cambridge! 🚀 It was an immense honour to present alongside some of the most prestigious mathematicians of our time.

A highlight? Introducing Trinity, a revolutionary auto-formalisation agent. This innovative tool is part of Christian Szegedy's verified superintelligence program with Morph Labs.

Morph Labs has used Trinity to auto-formalise a proof that the famous abc conjecture is true almost always, producing over 3500 lines of Lean.

Want to learn more about my work and see Jared and me discuss Trinity's incredible capabilities? Check out the session recording: https://lnkd.in/eifg42Z5 The section 45:00 - 59:00 is unmissable, make sure to watch it all!

#FormalMathematics #AI #ProofAutomation #BigProof #Math #Lean

You and 71 others   3 comments • 5 reposts

Before we wrap up…

Lean FRO: Shaping the Future of Lean Development

The Lean Focused Research Organization (FRO) is a non-profit dedicated to Lean's development.

Founded in **August 2023**, the organization has 19 members.

Its mission is to enhance critical areas: **scalability**, **usability**, **documentation**, and **proof automation**.

It must reach **self-sustainability in August 2028** and become the **Lean Foundation**.

We are very grateful for all philanthropic support we have received.

Lean FRO: by numbers

20 releases and **4,383 pull requests** merged in the main repository only since its launch in July 2023.

Public roadmaps: https://lean-fro.org/about/roadmap-y2/

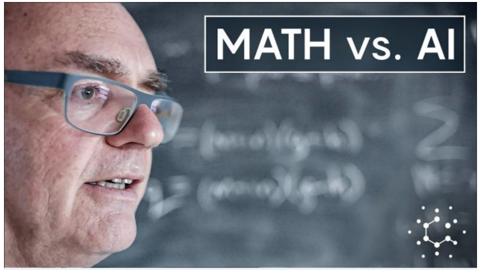
Lean project was featured in multiple venues NY Times, Quanta, Scientific American, etc.


The New York Times

A.I. and Chatbots › | Can A.I Be Fooled? | Testing a Tutorbot | Chatbot Prompts to Try | A.I.'s Literary Skills | What Are the Dangers of A.I.?

A.I. Is Coming for Mathematics, Too

For thousands of years, mathematicians have adapted to the latest advances in logic and reasoning. Are they ready for artificial intelligence?



When Computers Write Proofs, What's the Point of Mathematicians?

youtube.com

Lean FRO: Roadmap

Lean v4.22’s release will celebrate the Lean FRO’s second anniversary

Many new features coming in the Lean FRO year 3.

New Compiler - Enhanced performance and optimization

New Module System - Faster recompilation and better dependency management

Improved do-notation - better support for reasoning about it

Enhanced Proof Automation - Continue improving bv_decide, grind, simp

Scalability Improvements - Handle larger codebases efficiently

Literate Programming System - Seamless documentation integration

New Website - Modern interface and better resources

CSLib

A Mathlib for computer science.

Steering committee of CSLib:

Swarat Chaudhuri (Google DeepMind and UT Austin)

Clark Barrett (Stanford University and Amazon)

Jason Gross (Theorama)

Leo de Moura (Amazon and Lean FRO)

CSLib aims to be a foundation for **teaching**, **research**, and new **verification** efforts, including AI-assisted.

How can I contribute?

Help building Mathlib.

Want to engage with the vibrant Lean community? Join our Zulip channel.

Interested in ML kernels? Contribute to the KLR project.

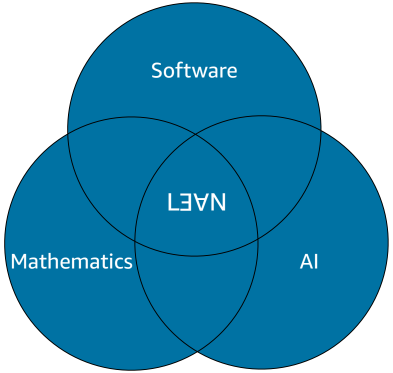
Want to contribute to a large formalization project? Join the FLT formalization project.

Start your own open-source Lean project! Your package will be available on our registry Reservoir.

Start using Lean online: live.lean-lang.org

Support the Lean FRO: Funding, partnerships, or simply advocating the project.

Conclusion



Lean is an **efficient programming language** and **proof assistant**.

Lean is very **extensible** and is implemented in Lean.

Lean proofs are maintainable, stable, and transparent.

Progress is accelerating with the Lean FRO: module system, new compiler, new proof automation, etc.

The Mathlib community is changing how math is done.

It is not just about proving but also understanding complex objects and proofs, getting new insights, and navigating through the “thick jungles” that are **beyond our cognitive abilities**.

Thank You

https://leanprover.zulipchat.com/
x: @leanprover
LinkedIn: Lean FRO
Mastodon: @leanprover@functional.cafe
#leanlang, #leanprover

https://www.lean-lang.org/