

# Formalizing the Future: Lean's Impact on Mathematics, Programming, and AI

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

May 6, 2025



## **Breaking the Cycle of Uncertainty: Math, Software, and AI You Can Trust**

Math, software, and AI often rely on **manual review** or **partial testing**.

An error in a theorem or critical software system can have massive consequences.

**Progress dies where fear of mistakes lives.**



## Breaking the Cycle of Uncertainty: Math, Software, and AI You Can Trust

Math, software, and AI often rely on **manual review** or **partial testing**.

An error in a theorem or critical software system can have massive consequences.

**Progress dies where fear of mistakes lives.**

Lean: **machine-checkable proofs eliminate guesswork and create trust.**

If every step is formally verified, we unlock unprecedented confidence and collaboration.



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

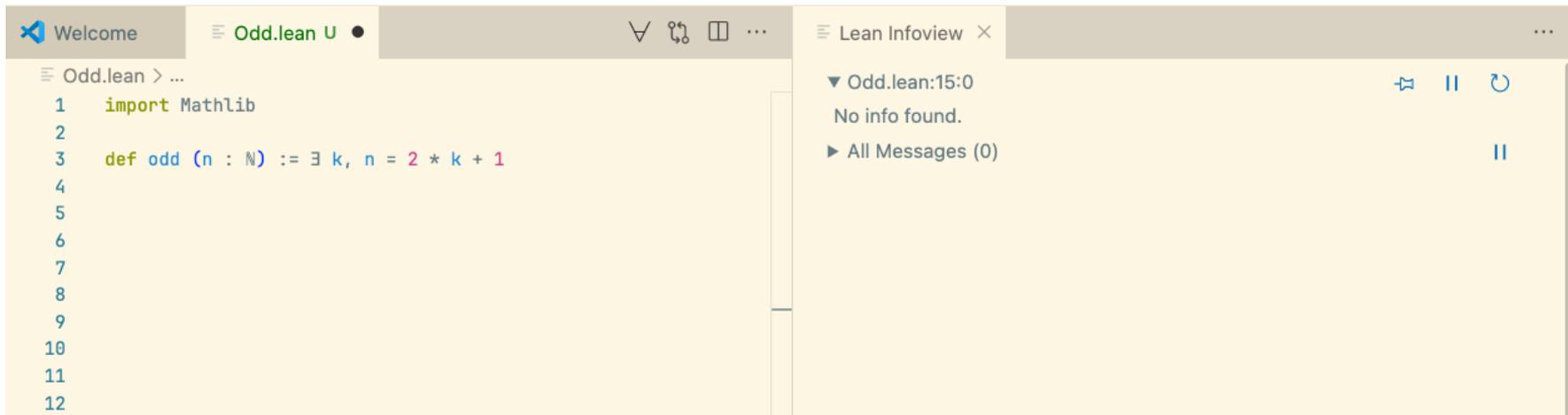
The Lean project, started in 2013, aimed at merging interactive and automated theorem proving.

Lean provides **machine-checkable proofs**.

Lean addresses the “trust bottleneck”.

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

## A small example



The screenshot shows the Lean IDE interface. The top bar contains a 'Welcome' tab and a file named 'Odd.lean U'. The code editor displays the following code:

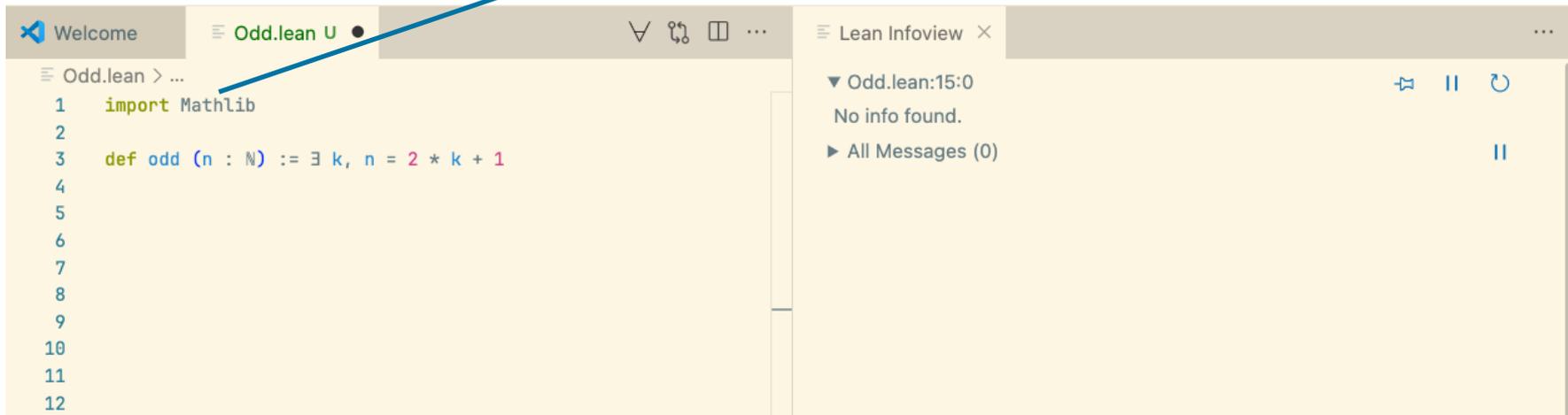
```
Odd.lean > ...  
1 import Mathlib  
2  
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

The right-hand side of the IDE shows the 'Lean Infoview' panel. It contains the following text:

▼ Odd.lean:15:0  
No info found.  
► All Messages (0)

## A small example

Mathlib is the Lean Mathematical library



The screenshot shows the Lean IDE interface. The code editor on the left contains the following code:

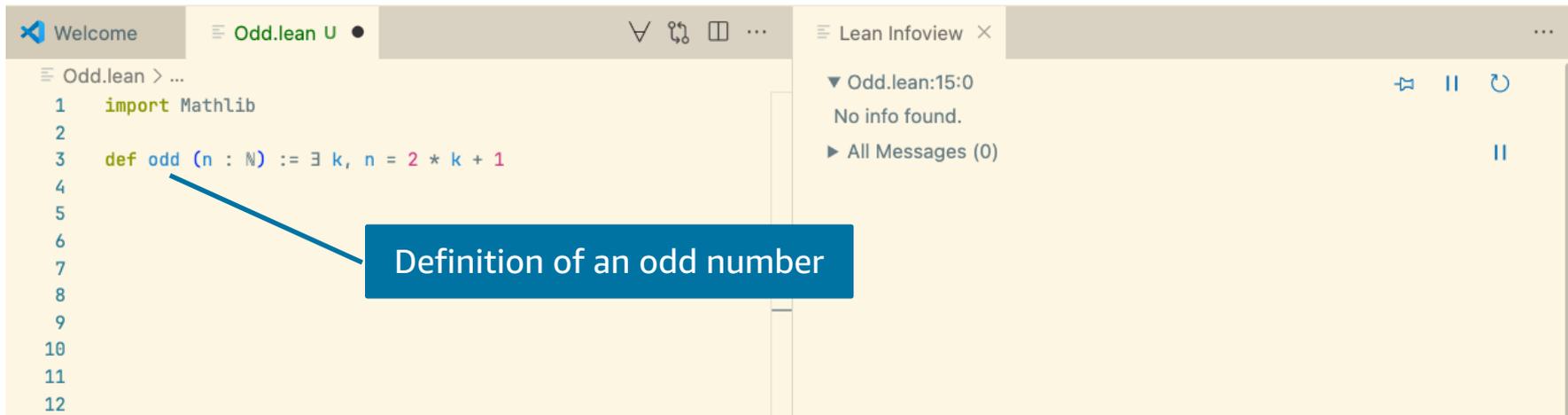
```
1 import Mathlib
2
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5
6
7
8
9
10
11
12
```

The infoview panel on the right shows the following content:

```
Lean Infoview ×
▼ Odd.lean:15:0
No info found.
► All Messages (0)
```

A blue arrow points from the text box above to the `import Mathlib` line in the code editor.

## A small example



The screenshot shows the Lean IDE interface. The main editor displays the following code:

```
Odd.lean > ...  
1 import Mathlib  
2  
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

A blue callout box with the text "Definition of an odd number" has an arrow pointing to the definition on line 3.

The right-hand pane shows the "Lean Infoview" for the definition. It contains the following text:

```
▼ Odd.lean:15:0  
No info found.  
► All Messages (0)
```

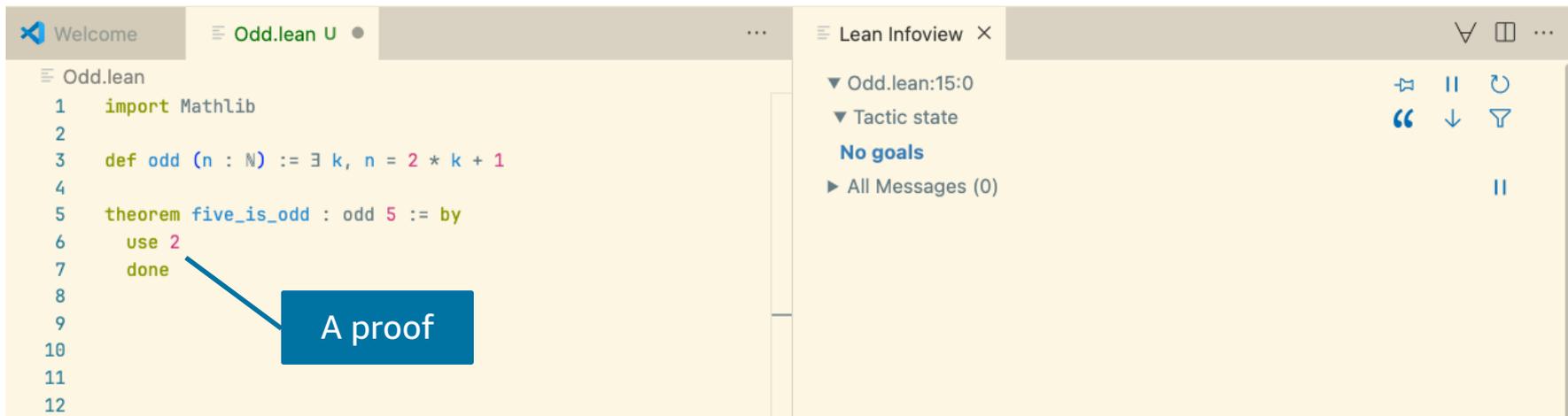
# Our first theorem

The screenshot shows the Lean IDE interface. The main editor displays the following code in `Odd.lean`:

```
1 import Mathlib
2
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5 theorem five_is_odd : odd 5 := by
6   use 2
7   done
```

The right-hand pane shows the `Lean Infoview` for the current theorem, indicating `No goals` and `All Messages (0)`. A blue callout box points to the theorem statement `theorem five_is_odd : odd 5 := by` with the text: **Theorem statement, i.e., the claim being made**

# Our first theorem



The screenshot shows the Lean IDE interface. The main editor displays the following code in `Odd.lean`:

```
1 import Mathlib
2
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5 theorem five_is_odd : odd 5 := by
6   use 2
7   done
```

A blue callout box with the text "A proof" has an arrow pointing to the `done` keyword on line 7.

The right-hand pane, titled "Lean Infoview", shows the current state of the proof:

- ▼ Odd.lean:15:0
- ▼ Tactic state
- No goals**
- All Messages (0)

Control icons for the infoview include a pin, a pause, a refresh, a quote, a down arrow, a filter, and a vertical bar.



# Our first theorem

```
Odd.lean > five_is_odd
1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  theorem five_is_odd : odd 5 := by
6    use 3
7    done
8
9
10
11
12
```

An incorrect proof

Lean Infoview

- Odd.lean:7:2
- Tactic state
- 1 goal
  - case h
    - $5 = 2 * 3 + 1$
- Messages (1)
- All Messages (1)

# Theorem proving in Lean is an interactive game

The screenshot shows the Lean IDE interface. On the left, the source file `Odd.lean` contains the following code:

```

1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    done
8
9
10
11
12

```

On the right, the `Lean Infoview` panel displays the current state of the proof:

- `Odd.lean:7:2`
- `Tactic state`
- `1 goal`
- `n : ℕ`
- `⊢ odd n → odd (n * n)`
- `Messages (1)`
- `All Messages (2)`

A blue callout box with the text "The 'game board'" has an arrow pointing to the goal statement `⊢ odd n → odd (n * n)` in the `Lean Infoview` panel.

*"You have written my favorite computer game", Kevin Buzzard*

# Theorem proving in Lean is an interactive game

```
Odd.lean > square_of_odd_is_odd
1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    intro ⟨k₁, e₁⟩
8    done
9
10
11
12
```

Lean Infoview

Odd.lean:8:2

Tactic state

1 goal

$n$   $k_1$  :  $\mathbb{N}$

$e_1$  :  $n = 2 * k_1 + 1$

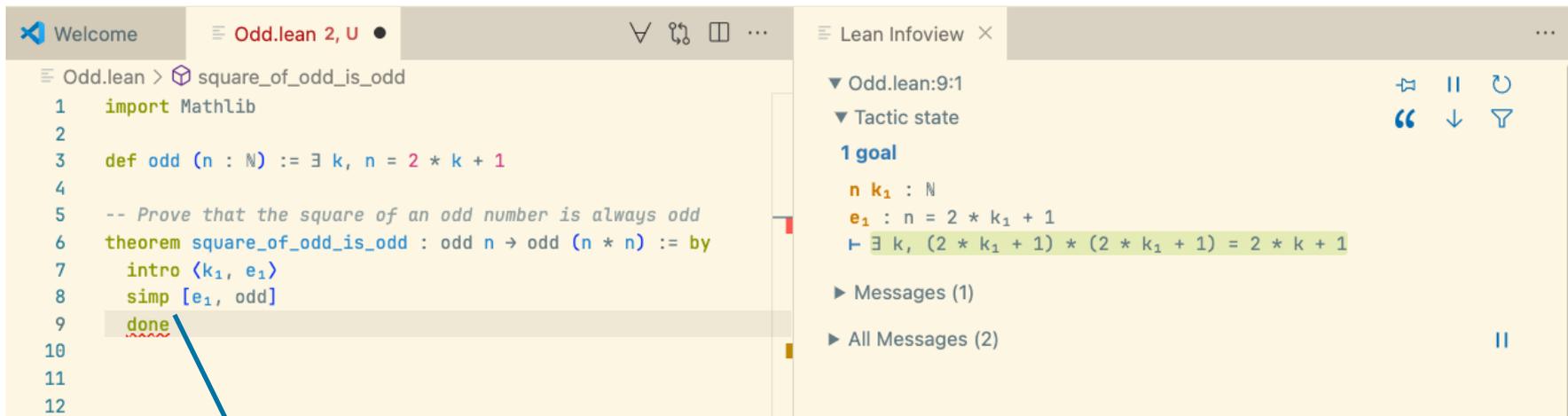
⊢  $\text{odd } (n * n)$

Messages (1)

All Messages (2)

A "game move", aka "tactic"

# Theorem proving in Lean is an interactive game



The screenshot shows the Lean IDE interface. On the left, a code editor displays a Lean script for proving that the square of an odd number is odd. The script includes an import, a definition of an odd number, a theorem statement, and a proof using the `simp` tactic. A blue arrow points from the `simp` line to a callout box. On the right, the 'Lean Infoview' panel shows the current tactic state, including the goal and the hypotheses used by the `simp` tactic.

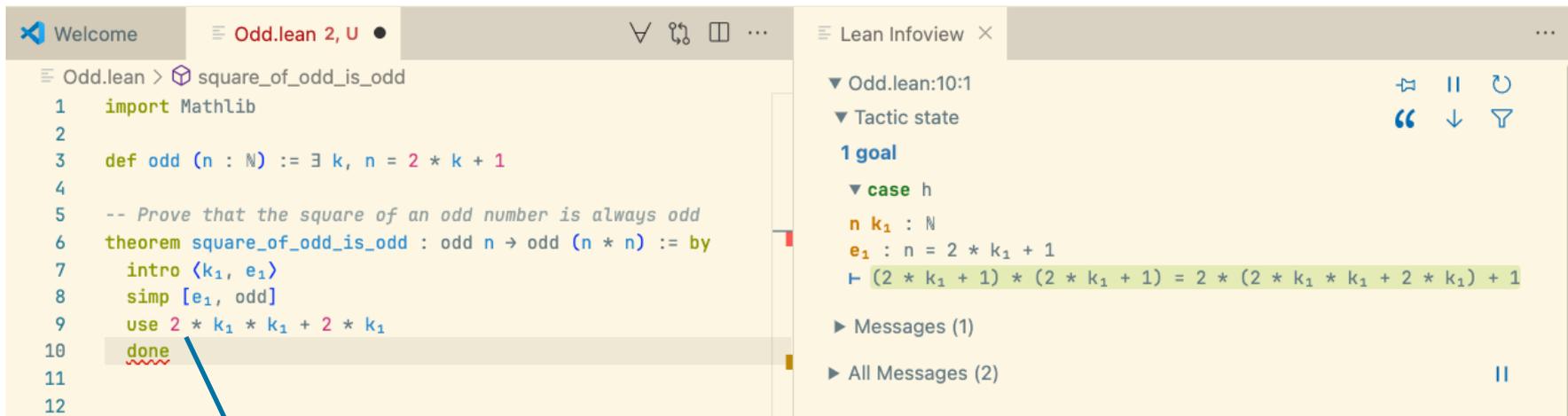
```
Odd.lean > square_of_odd_is_odd
1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    intro (k1, e1)
8    simp [e1, odd]
9    done
10
11
12
```

Lean Infoview

- ▼ Odd.lean:9:1
- ▼ Tactic state
- 1 goal
- n k<sub>1</sub> : ℕ
- e<sub>1</sub> : n = 2 \* k<sub>1</sub> + 1
- ┆ ∃ k, (2 \* k<sub>1</sub> + 1) \* (2 \* k<sub>1</sub> + 1) = 2 \* k + 1
- Messages (1)
- All Messages (2)

The “game move” `simp`, the simplifier, is one of the most popular moves in our game

# Theorem proving in Lean is an interactive game



The screenshot shows the Lean IDE interface. On the left, the source code for a theorem is displayed:

```

1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    intro (k1, e1)
8    simp [e1, odd]
9    use 2 * k1 * k1 + 2 * k1
10   done
11
12

```

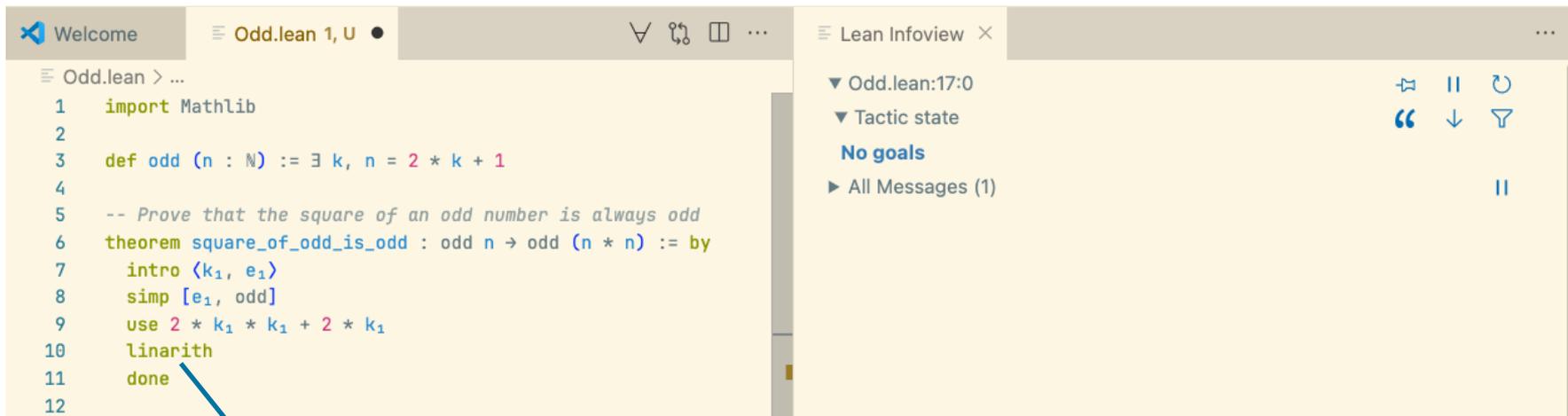
On the right, the 'Lean Infview' panel shows the current state of the proof:

- Odd.lean:10:1
- Tactic state
- 1 goal
- case h
- n k<sub>1</sub> : ℕ
- e<sub>1</sub> : n = 2 \* k<sub>1</sub> + 1
- ┆ (2 \* k<sub>1</sub> + 1) \* (2 \* k<sub>1</sub> + 1) = 2 \* (2 \* k<sub>1</sub> \* k<sub>1</sub> + 2 \* k<sub>1</sub>) + 1
- Messages (1)
- All Messages (2)

A blue arrow points from the `done` keyword in the code to the text box below.

The “game move” `use` is the standard way of proving statements about existentials

# Theorem proving in Lean is an interactive game



```
1 import Mathlib
2
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5 -- Prove that the square of an odd number is always odd
6 theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7   intro (k₁, e₁)
8   simp [e₁, odd]
9   use 2 * k₁ * k₁ + 2 * k₁
10  linarith
11  done
12
```

Lean Infview ×

- Odd.lean:17:0
- Tactic state
- No goals
- All Messages (1)

We complete this level using `linarith`, the linear arithmetic, move



# Theorem proving in Lean is an interactive **and addictive** game

```
Welcome | Odd.lean 1, U • | Lean Infoview ×
```

```
Odd.lean > ...
1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    intro (k₁, e₁)
8    simp [e₁, odd]
9    use 2 * k₁ * k₁ + 2 * k₁
10   linarith
11   done
12
```

▼ Odd.lean:17:0  
▼ Tactic state  
**No goals**  
► All Messages (1)

*"You can do 14 hours a day in it and not get tired and feel kind of high the whole day. You're constantly getting positive reinforcement", Amelia Livingston*



## 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



Quanta magazine

Physics

Mathematics

Biology

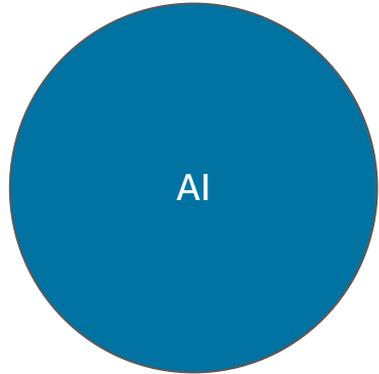
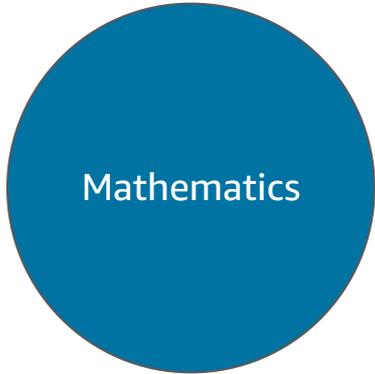
Computer Science

Topics

Archive

FOUNDATIONS OF MATHEMATICS

# Building the Mathematical Library of the Future



# Mathematics



# Preamble: the Perfectoid Spaces Project

*Kevin Buzzard, Patrick Massot, Johan Commelin*

Goal: Demonstrate that we can **define complex mathematical objects** in Lean.

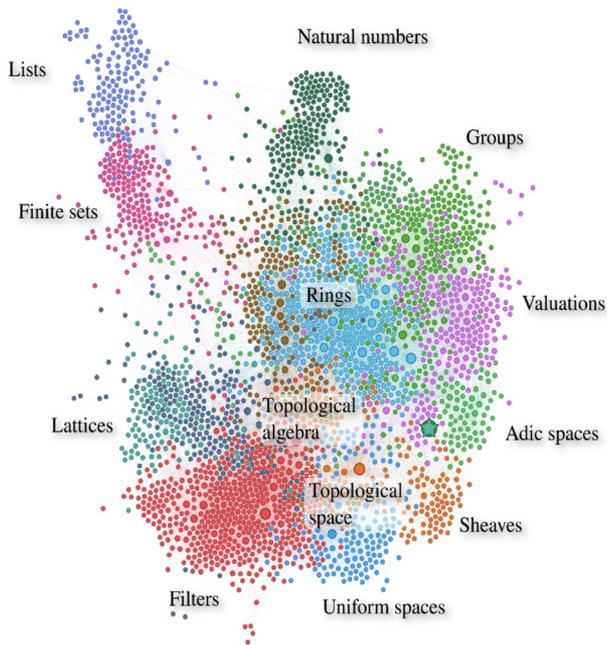
They translated Peter Scholze's definition into a form a computer can understand.

It not only achieved its goals but also demonstrated to the math community that **formal objects can be visualized and inspected with computer assistance.**

**Math** is now **data** that can be **processed, transformed,** and **inspected** in various ways.

# Preamble: the Perfectoid Spaces Project (cont.)

Kevin Buzzard, Patrick Massot, Johan Commelin



🏠 Home

What are "perfectoid spaces"?

▲

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

**72** 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 : ∃ ω : pseudo_uniformizer R, ω^p | p in R^o)
  (Frobenius : surjective (Frob R^o/p))
            
```

🔖
🔄



Mathlib > RingTheory > Finiteness.lean

```
355
356 theorem F6.stabilizes_of_iSup_eq {M' : Submodule R M} (hM' : M'.FG) (N : ℕ → Submodule R M)
357   (H : iSup N = M') : ∃ n, M' = N n := by
358   obtain ⟨S, hS⟩ := hM'
359   have : ∀ s : S, ∃ n, (s : M) ∈ N n := fun s =>
360     (Submodule.mem_iSup_of_chain N s).mp
361     (by
362       rw [H, ← hS]
363       exact Submodule.subset_span s.2)
364   choose f hf using this
365   use S.attach.sup f
366   apply le_antisymm
367   · conv_lhs => rw [← hS]
368     rw [Submodule.span_le]
369     intro s hs
370     exact N.2 (Finset.le_sup <| S.mem_attach ⟨s, hs⟩) (hf _)
371   · rw [← H]
372     exact le_iSup _ _
```

▼ Finiteness.lean:365:2

▼ Tactic state

1 goal

▼ case intro

R : Type u\_1

M : Type u\_2

inst<sup>2</sup> : Semiring R

inst<sup>1</sup> : AddCommMonoid M

inst : Module R M

M' : Submodule R M

N : ℕ → 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

```

355
356 theorem F6.stabilizes_of_iSup_eq {M' : Submodule R M} (hM' : M'.FG) (N : N → Submodule R M)
357   (H : iSup N = M') : ∃ n, M' = N n := by
358   obtain ⟨S, hS⟩ := hM'
359   have : ∀ s : S, ∃ n, (s : M) ∈ N n := fun s =>
360     (Submodule.mem_iSup_of_chain N s).mp
361     (by
362       rw [H, ← hS]
363       exact Submodule.subset_span s.2)
364   choose f hf using this
365   use S.attach.sup f
366   apply le_antisymm
367   · conv_lhs => rw [← hS]
368     rw [Submodule.span_le]
369     intro s hs
370     exact N.2 (Finset.le_sup <| S.mem_attach ⟨s, hs⟩) (hf _)
371   · rw [← H]
372     exact le_iSup _ _
---
```

▼ Finiteness.lean:365:2

▼ Tactic state

1 goal

▼ case intro

R : Type u\_1

M : Type u\_2

inst<sup>2</sup> : Semiring R

inst<sup>1</sup> : AddCommMonoid M

inst : Module R M

M' : Submodule R M

N : N → 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

555

356 theorem FG.stabilizes\_of\_iSup\_eq {M' : Submodule R M} (hM' : M'.FG) (N : N → Submodule R M)

Defs.lean ~/projects/mathlib4/Mathlib/Algebra/Module/Submodule - Definitions (1)

25 assert\_not\_exists DivisionRing

26

27 open Function

28

29 universe u' u' u v w

30

31 variable {G : Type u''} {S : Type u'} {R : Type u} {M : Type v} {u :

32

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

38

structure Submodule (R : Type u) (

▼ Finiteness.lean:356:44

▼ Expected type

R : Type u\_1

M : Type u\_2

inst<sup>4</sup> : Semiring R

inst<sup>3</sup> : AddCommMonoid M

inst<sup>2</sup> : Module R M

P : Type u\_3

inst<sup>1</sup> : AddCommMonoid P

inst : Module R P

f : M →<sub>[R]</sub> P

↳ Type u\_2

► All Messages (0)



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

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

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

```
84   add_decl_doc Submonoid.toSubsemigroup
85
86   /- `SubmonoidClass S M` says `S` is a type of subsets `s ≤ M` that
87   and are closed under `(*)` -/
88   class SubmonoidClass (S : Type*) (M : outParam Type*) [MulOneClass M]
89     MulMemClass S M, OneMemClass S M : Prop
90
91   section
92
93   /- An additive submonoid of an additive monoid `M` is a subset cont
94   closed under addition. -/
95   structure AddSubmonoid (M : Type*) [AddZeroClass M] extends AddSubse
96     /- An additive submonoid contains `0`. -/
97     zero_mem' : (0 : M) ∈ carrier
98
```

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)



## 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.**

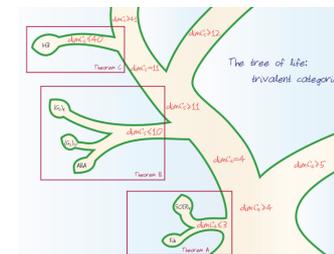
*“The Lean Proof Assistant was really that: an assistant in navigating through the thick jungle that this proof is. Really, one key problem I had when I was trying to find this proof was that I was essentially unable to keep all the objects in my RAM, and I think the same problem occurs when trying to 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 {a} [CommRing a] [IsCharP a 0] (d t c : a) (d_inv PS03_inv : a) :
  d^2 * (d + t - d * t - 2) * (d + t + d * t) = 0 →
  -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) * PS03_inv = 1 →
  t^2 = t + 1 := by
  grind +ring
```

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 PS03_inv : α) :
  d^2 * (d + t - d * t - 2) * (d + t + d * t) = 0 →
  -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) * PS03_inv = 1 →
  t^2 = t + 1 := by
  grind +ring
```

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.

`grind +ring` **is just another move in our interactive game.**

```
example {α} [CommRing α] [IsCharP α 0] [NoNatZeroDivisors α]
  (d t : α)
  (Q_3_4 : d^3*(t^4+3*t^3-t^2-3*t-1)+d^2*(2*t^4+t^2+2*t+1)+d*(t^4-3*t^3+3*t^2+6*t+1)-t^2+2*t+2 = 0)
  (Q_4_5 : d^4*t^5+d^3*(3*t^5-3*t^4-3*t^3+7*t^2+5*t+1)
    +d^2*(3*t^5-5*t^4-5*t^3+10*t^2+12*t+2)+d*(t^5-t^4-5*t^3+3*t^2+9*t+5)+t^4-3*t^3+4*t+1 = 0) :
  243+2268*t+7371*t^2+6192*t^3-16071*t^4-31161*t^5+11784*t^6+51173*t^7
  -4565*t^8-48060*t^9+7055*t^10+26569*t^11-9795*t^12-5753*t^13+4514*t^14-1020*t^15+72*t^16 = 0 := by
  grind +ring
```



## Should we trust Lean?

Lean has a small trusted proof checker.

Do I need to trust the checker?

No, **you can export your proof**, and use external checkers. There are checkers implemented in C/C++, Rust, Lean, etc.

**You can implement your own checker.**



## What did we learn?

Machine-checkable proofs enable a new level of **collaboration** in mathematics.

The power of the **community**.

We don't need to trust our automation/moves.

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**.

## What did we learn?

Another unexpected benefit of formal mathematics: **auto refactoring** and **generalization**.

general An example of why formalization is useful

Mar 31



**Riccardo Brasca** EDITED

7:53 AM

I really like what is going on with #12777. @Sebastian Monnet proved that if  $E$ ,  $F$  and  $K$  are fields such that `finite_dimensional F E`, then `fintype (E →a [F] K)`. We already have `docs#field.alg_hom.fintype`, that is exactly the same statement with the additional assumption `is_separable F E`.

The interesting part of the PR is that, with the new theorem, the linter will automatically flag all the theorem that can be generalized (for free!), removing the separability assumption. I think in normal math this is very difficult to achieve, if I generalize a 50 years old paper that assumes `p ≠ 2` to all primes, there is no way I can manually check and maybe generalize all the papers that use the old one.



3



5

# 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.

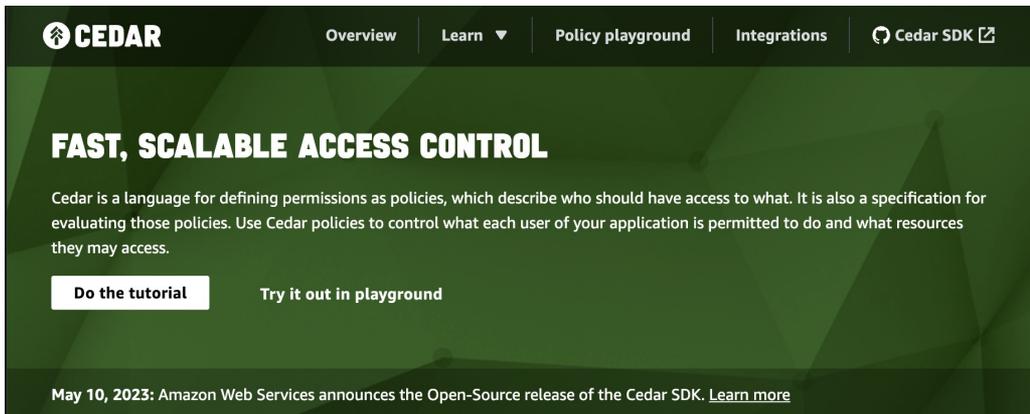
You can prove that your code has the properties you expect.

*“Testing can show the presence of bugs, but not their absence”, E. Dijkstra*



# Cedar

<https://www.cedarpolicy.com/>

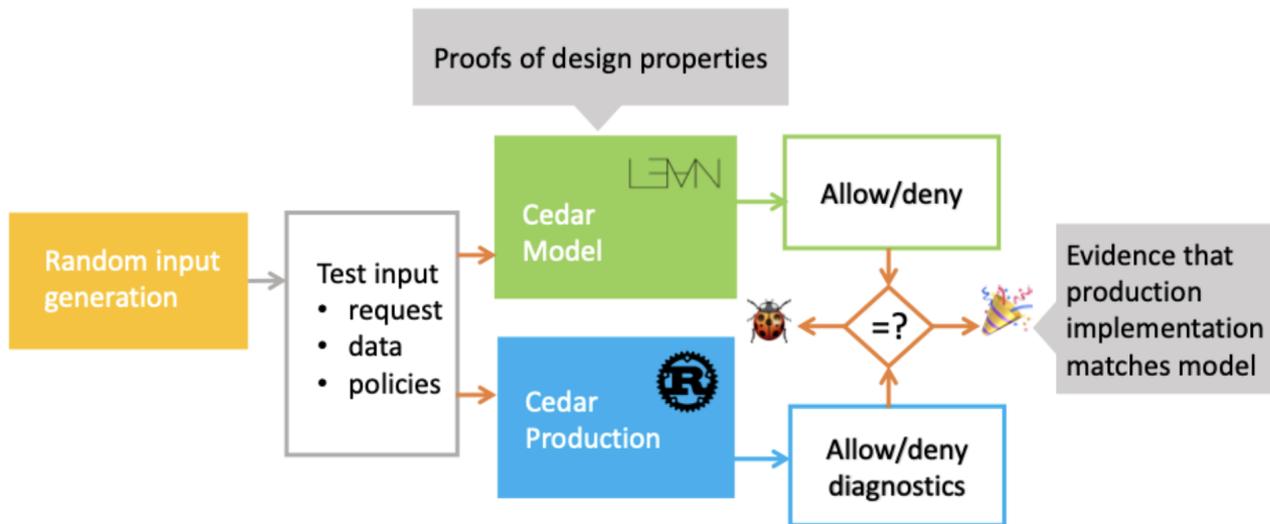


The screenshot shows the Cedar website homepage. At the top, there is a navigation bar with the Cedar logo on the left and links for 'Overview', 'Learn', 'Policy playground', 'Integrations', and 'Cedar SDK'. The main content area has a dark green background with the heading 'FAST, SCALABLE ACCESS CONTROL'. Below this, a paragraph explains that Cedar is a language for defining permissions as policies. Two buttons are present: 'Do the tutorial' and 'Try it out in playground'. At the bottom, a news snippet from May 10, 2023, mentions Amazon Web Services' announcement of the Open-Source release of the Cedar SDK.

<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



**Takeaway:** “We’ve found Lean to be a great tool for verified software development. You get a full-featured programming language, fast proof checker and runtime, and a familiar way to build both models and proofs”



# Cedar

To learn more about Cedar:

<https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/>

The screenshot shows the top navigation bar of the AWS website. On the left is the AWS logo. To its right are links for 'About AWS', 'Contact Us', 'Support' (with a dropdown arrow), 'My Account' (with a dropdown arrow), and 'Sign In'. A prominent orange button labeled 'Create an AWS Account' is on the far right. Below these are links for 'Products', 'Solutions', 'Pricing', 'Documentation', 'Learn', 'Partner Network', 'AWS Marketplace', 'Customer Enablement', 'Events', and 'Explore More', followed by a search icon. A secondary bar below contains 'AWS Blog Home', 'Blogs' (with a dropdown arrow), and 'Editions' (with a dropdown arrow).

[AWS Open Source Blog](#)

## Lean Into Verified Software Development

by Kesha Hietala and Emina Torlak | on 08 APR 2024 | in [Amazon Verified Permissions](#), [Open Source](#), [Security](#), [Identity](#), & [Compliance](#), [Technical How-to](#) | [Permalink](#) | [Comments](#) | [Share](#)

### Resources

[Open Source at AWS](#)  
[Projects on GitHub](#)

# Differential Privacy

A mathematical framework that ensures the **privacy of individuals** in a dataset by adding controlled **random noise** to the data.

Discrete sampling algorithms, like the **Discrete Gaussian Sampler**, are used to add carefully calibrated noise to data.

What may go wrong if a buggy sampler is used?

**Privacy Violations:** leakage of sensitive information

**Incorrect Results:** distorted analysis results



# SampCert

A project led by **Jean-Baptiste Tristan** at AWS.

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

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

He managed to implement **aggressive optimizations** because Lean served as a guide, ensuring that **no bugs** were introduced.



## 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**.



## 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.

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 :**

```
  ∃ p, (run (Env.init x) popcount 8) = some p ∧ p "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
50 theorem popcount_correct :
51   ∃ p, (run (Env.init x) popcount 8) = some p
52   simp [run, popcount, Expr.eval, Expr.BinOp.app
53     bv_decide
54
```

▼Tactic state

1 goal

x : Value

$$\begin{aligned} & \vdash ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + ((x - (x \ggg 1 \&\& \\ & 1431655765\#32)) \ggg 2 \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32)) \ggg \\ & 4 \&\& \\ & 252645135\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32)) \ggg \\ & 4 \&\& \\ & 252645135\#32) \ggg \\ & 8 + \\ & (((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32)) \ggg \\ & 4 \&\& \\ & 252645135\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \&\& 858993459\#32) + \\ & ((x - (x \ggg 1 \&\& 1431655765\#32)) \ggg 2 \&\& 858993459\#32)) \ggg \\ & 4 \&\& \\ & 252645135\#32) \ggg \\ & 8) \ggg \\ & 16 \&\& \\ & 63\#32 = \\ & (x \&\& 1\#32) + (x \ggg 1 \&\& 1\#32) + (x \ggg 2 \&\& 1\#32) + (x \ggg 3 \&\& 1\#32) + (x \ggg \\ & 4 \&\& 1\#32) + \end{aligned}$$

## grind in Software Verification

```
example (x : BitVec 8) : (x + 16)*(x - 16) = x^2 := by
  grind +ring
```

```
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]
```



## What did we learn?

Machine-checkable proofs enable you to **code without fear**.

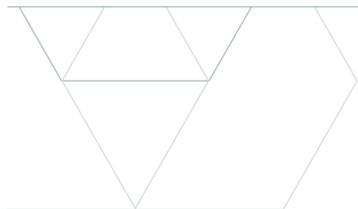
Powerful proof automation.

Industrial projects: Verified compilers, policy languages, cryptographic libraries, etc.

Many more at the **Lean Project Registry**: <https://reservoir.lean-lang.org/>

 | science

[Research areas](#) ▾ [Blog](#) [Publications](#) [Conferences](#) [Code and datasets](#) [Academia](#) ▾ [Careers](#)



AUTOMATED REASONING

How the Lean language  
brings math to coding  
and coding to math

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.**



## AI Proof Assistants

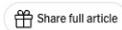
Several groups are developing AI that suggests the **next move(s)** in Lean's interactive proof game.

[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.

## Move Over, Mathematicians, Here Comes AlphaProof

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



Ring 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



## What did we learn?

Machine-checkable proofs enable **AI that does not hallucinate**.

LLMs are getting better and better at explaining Lean code.

In an era of big data and LLMs, machine-checkable proofs ensure trust in results.

AI systems that prove rather than guess.

**Before we wrap up...**



# Lean Enables Decentralized Collaboration

## Lean is Extensible

Users extend Lean using Lean itself.

### **Lean is implemented in Lean.**

You can make it your own.

You can create your own moves.

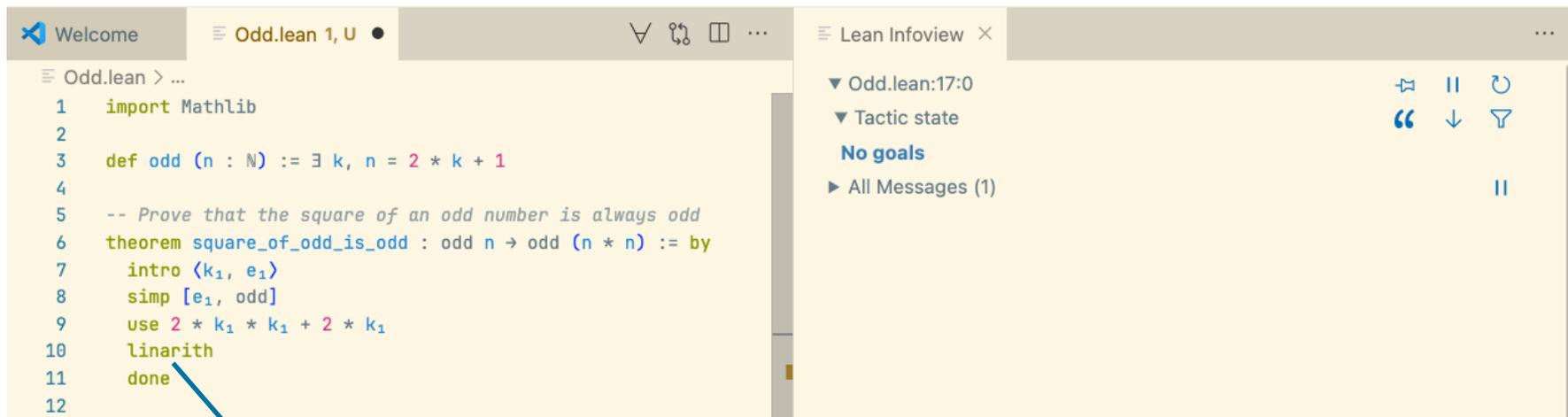
## Machine-Checkable Proofs

You don't need to trust me to use my proofs.

You don't need to trust my automation to use it.

### **Code without fear.**

# Lean is a game where we can implement your own moves



The screenshot shows the Lean IDE interface. The main editor displays a Lean script for proving that the square of an odd number is odd. The script includes an import, a definition of an odd number, and a theorem with a proof using several tactics. A blue arrow points from the `linarith` tactic on line 10 to a callout box. The right-hand pane shows the 'Lean Infoview' with the current tactic state, indicating 'No goals'.

```
1 import Mathlib
2
3 def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5 -- Prove that the square of an odd number is always odd
6 theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7   intro ⟨k₁, e₁⟩
8   simp [e₁, odd]
9   use 2 * k₁ * k₁ + 2 * k₁
10  linarith
11  done
12
```

Lean Infoview

- Odd.lean:17:0
- Tactic state
- No goals
- All Messages (1)

The `linarith` “move” was implemented by the Mathlib community in Lean!

# Lean is a game where we can implement your own moves

The screenshot shows the Lean IDE interface. On the left, a code editor displays a proof script for the theorem `square_of_odd_is_odd`. The script includes an `import Mathlib` statement, a definition of `odd`, and a proof using `linarith`. A blue arrow points from the `linarith` tactic on line 10 to a callout box. On the right, the 'Lean Infoview' panel shows the current tactic state, which is 'No goals', and a list of messages.

```

1  import Mathlib
2
3  def odd (n : ℕ) := ∃ k, n = 2 * k + 1
4
5  -- Prove that the square of an odd number is always odd
6  theorem square_of_odd_is_odd : odd n → odd (n * n) := by
7    intro ⟨k₁, e₁⟩
8    simp [e₁, odd]
9    use 2 * k₁ * k₁ + 2 * k₁
10   linarith
11   done
12

```

Lean Infoview

- Odd.lean:17:0
- Tactic state
- No goals
- All Messages (1)

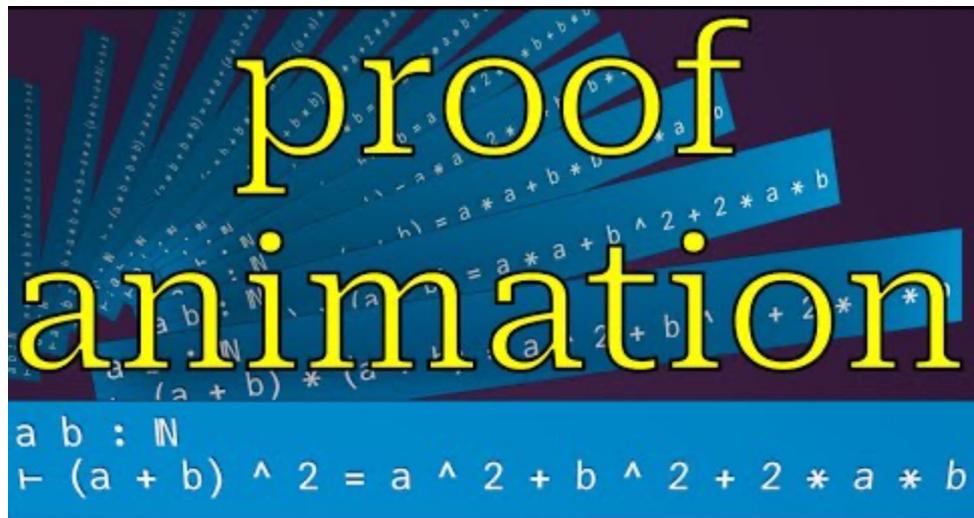
The `linarith` “move” was implemented by the Mathlib community in Lean!

The `by_decide` and `grind` “moves” are also implemented in Lean!

## You can use Lean to introspect its internal data

The tool [lean-training-data](#) is implemented in Lean itself. **It is a Lean package.**

A similar approach can be used to automatically generate proof animations.





# 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**.

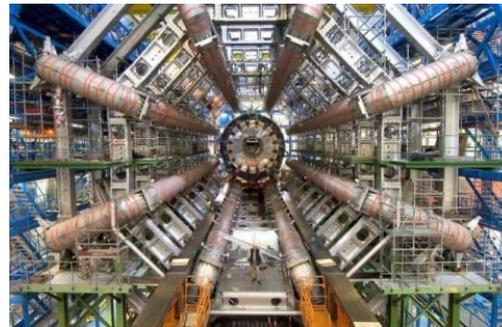
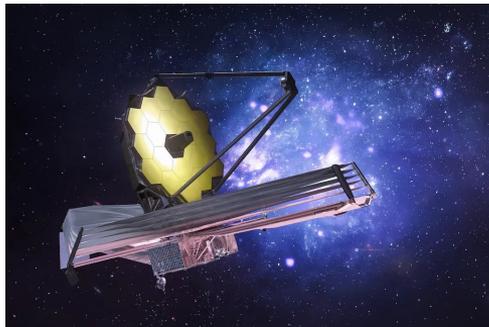
It must reach **self-sustainability in August 2028** and become the **Lean Foundation**.

Philanthropic support is gratefully acknowledged from the **Simons Foundation**, the **Alfred P. Sloan Foundation**, **Richard Merkin**, and **Alex Gerko**.

## FROs accelerate scientific progress / Lean as a Catalyst

James Webb Telescope and CERN illustrate a common pattern in science: a need for projects that are bigger than an academic lab can undertake, more coordinated than a loose consortium or themed department, and not directly profitable enough to be a venture-backed startup or industrial R&D project.

<https://www.convergentresearch.org/about-fros>



# Lean FRO: by numbers

**19 releases** and **4,047 pull requests** merged in the main repository only since its launch in July 2023.

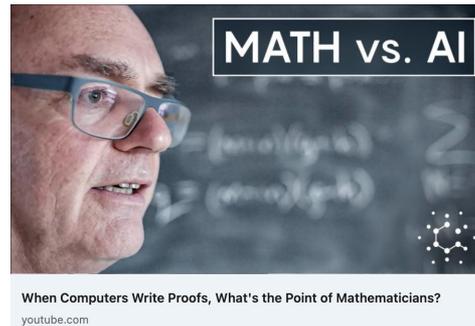
Public roadmaps: <https://lean-fro.org/about/roadmap-y2/>

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



## *A.I. Is Coming for Mathematics, Too*

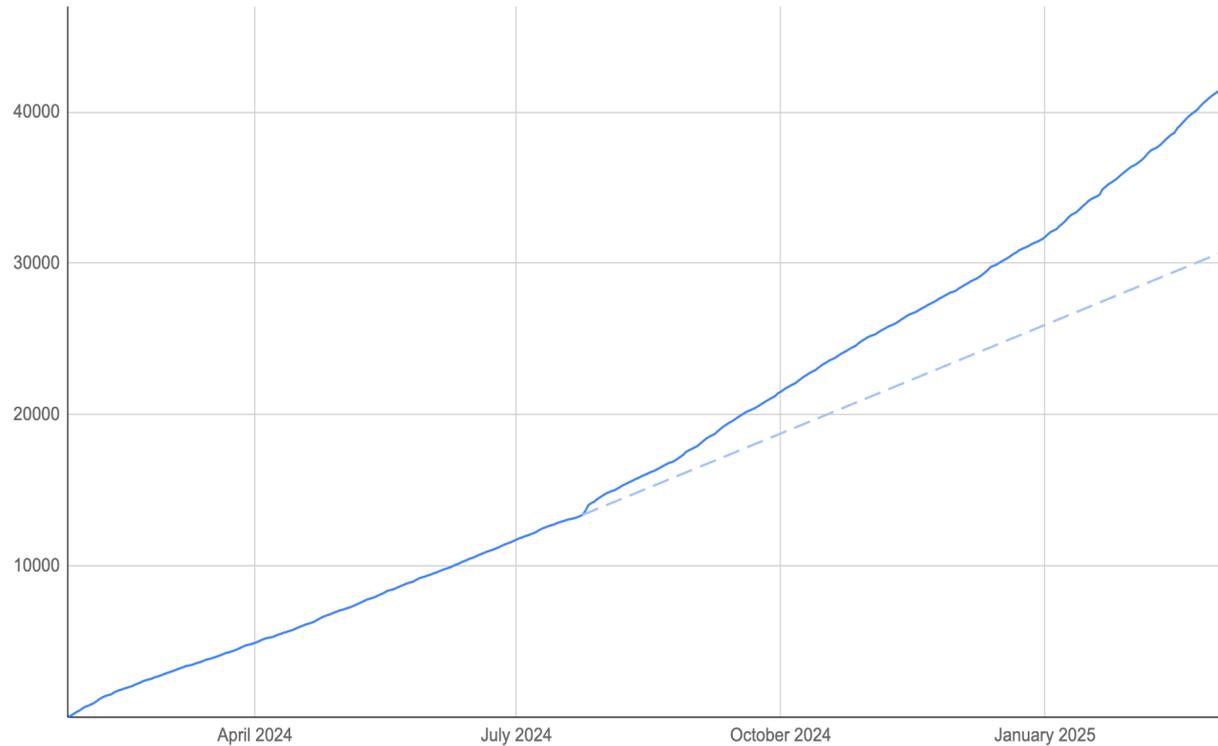
For thousands of years, mathematicians have adapted to the latest advances in logic and reasoning. Are they ready for artificial intelligence?



# Growth of Lean projects on GitHub



## New installations of Lean Development Environment (2024 to present)





## 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](https://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**.

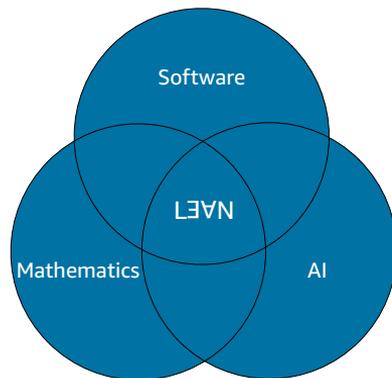
The Mathlib community is changing how math is done.

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**.

Lean tracks details, so humans focus on big ideas.

Decentralized collaboration with Lean: Large teams can collectively tackle huge proofs without losing track.

The entire discipline thrives when no one has to “take it on faith.”



# Thank You

<https://leanprover.zulipchat.com/>

x: @leanprover

LinkedIn: Lean FRO

Mastodon: @leanprover@functional.cafe

#leanlang, #leanprover

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

