Lean 4: an extensible proof assistant and programming language

Leonardo de Moura
Senior Principal Applied Scientist - AWS
Chief Architect - Lean FRO

LEAN Proof Assistant & Programming Language

Based on dependent type theory

Goals

- **Extensibility**, Expressivity, Scalability, Efficiency

A platform for

- Formalized mathematics
- Software development and verification
- Developing custom automation and Domain Specific Languages

Small trusted kernel, external type/proof checkers

http://lean-lang.org

LEAN is and IDE for automated reasoning

Lean is a development environment for automated reasoning.

Proofs and definitions are machine checkable.

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

A compiler: high-level language ⇒ kernel code

```
theorem euclid_exists_infinite_primes (n : ℕ) : ∃ p, n ≤ p ∧ Prime p :=
  let p := minFac (factorial n + 1)
  have f1 : (factorial n + 1) ≠ 1 :=
    ne_of_gt $ succ_lt_succ' $ factorial_pos _
  have pp : Prime p :=
    min_fac_prime f1
  have np : n ≤ p := le_of_not_ge fun h =>
    have h₁ : p | factorial n := dvd_factorial (min_fac_pos _) h
    have h₂ : p | 1 := (Nat.dvd_add_iff_right h₁).2 (min_fac_dvd _)
    pp.not_dvd_one h₂
  Exists.intro p
```

Lean 4 is an efficient programming language

We want proof automation written by users to be very efficient.

Lean memory manager is **now** the Bing memory manager (Daan Leijen – RiSE).

"Functional but in Place" (FBIP) distinguished paper award at PLDI'21.

Proofs are used to optimize code too.

It is a fully extensible programming language.

There are many more surprises coming...

Lean is a language for "programming your proofs and proving your programs"

LEAN enables decentralized collaboration

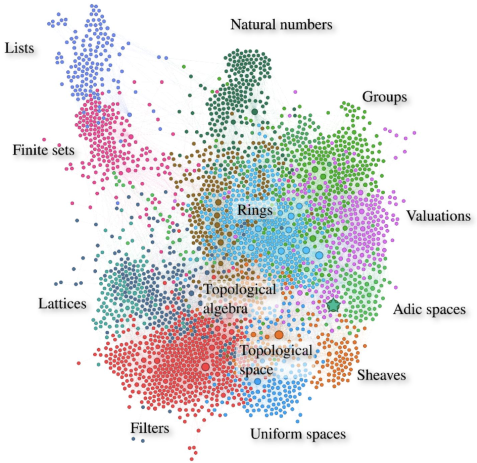
Meta-programming

Users extend Lean using Lean itself.

Proof automation.

Visualization tools.

Custom notation.



Formal Proofs

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

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

Hack without fear.

LEAN Community develops Mathlib

mathlib documentation   algebraic_geometry.Scheme   Google site search

style guide
documentation style guide
naming conventions

Library

core
- data
- init
- system

mathlib
- algebra
- algebraic_geometry
  - presheafed_space
  - EllipticCurve
  - Scheme
  - Spec
  - is_open_comap_C
  - locally_ringed_space
  - presheafed_space
  - prime_spectrum

```
theorem algebraic_geometry.Scheme.Γ_obj_op                                source
    (X : algebraic_geometry.Scheme) :
  algebraic_geometry.Scheme.Γ.obj (opposite.op X) =
   X.X.to_SheafedSpace.to_PresheafedSpace.presheaf.obj (opposite.op ⊤)

@[simp]                                                                   source
theorem algebraic_geometry.Scheme.Γ_map {X Y : algebraic_geometry.Schemeᵒᵖ}
    (f : X ⟶ Y) :
  algebraic_geometry.Scheme.Γ.map f =
   f.unop.val.c.app (opposite.op ⊤) ≫
    (opposite.unop Y).X.to_SheafedSpace.to_PresheafedSpace.presheaf
     .map   algebraic_geometry.LocallyRingedSpace.to_SheafedSpace
     (topological_space.opens.le_map_top f.unop.val.base ⊤).op

theorem algebraic_geometry.Scheme.Γ_map_op                                source
    {X Y : algebraic_geometry.Scheme} (f : X ⟶ Y) :
  algebraic_geometry.Scheme.Γ.map f.op =
   f.val.c.app (opposite.op ⊤) ≫
    X.X.to_SheafedSpace.to_PresheafedSpace.presheaf.map
     (topological_space.opens.le_map_top f.val.base ⊤).op
```

algebraic_geometry.Scheme

source

- Imports
- Imported by

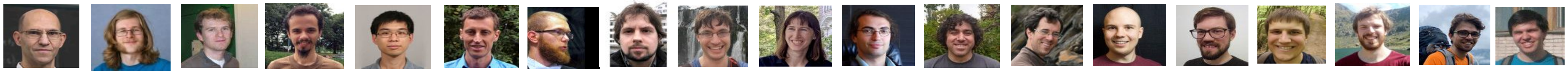
algebraic_geometry.Scheme
algebraic_geometry.Scheme.Spec
algebraic_geometry.Scheme.Spec_map
algebraic_geometry.Scheme.Spec_map_2
algebraic_geometry.Scheme.Spec_map_comp
algebraic_geometry.Scheme.Spec_map_id
algebraic_geometry.Scheme.Spec_obj
algebraic_geometry.Scheme.Spec_obj_2
algebraic_geometry.Scheme.

The Lean Mathematical Library

The mathlib Community*

Abstract

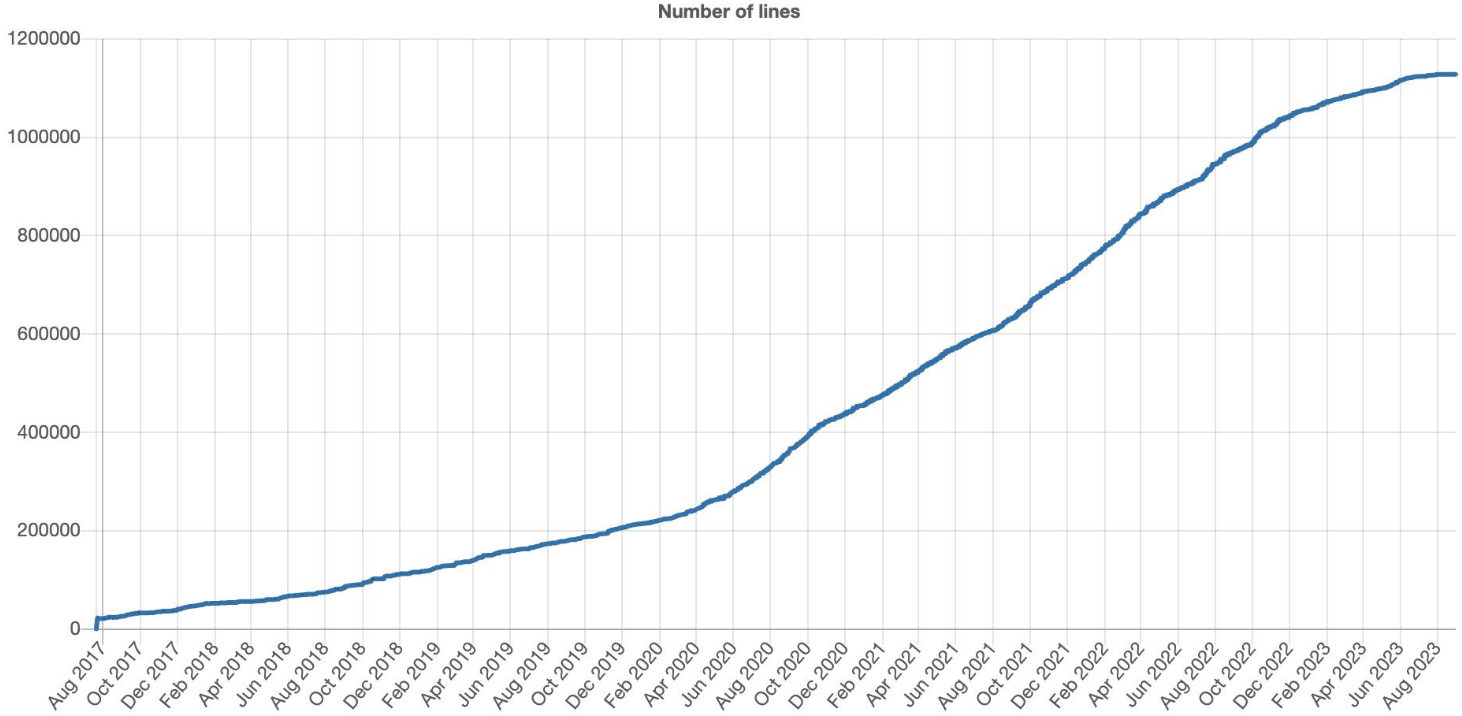
This paper describes mathlib, a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant. Among proof assistant libraries, it is distinguished by its dependently typed foundations, focus on classical mathematics, extensive hierarchy of structures, use of large- and small-scale automation, and distributed organization. We explain the architecture and design decisions of the library and the social organization that has led to its development.



Mathlib statistics

Counts

Definitions	Theorems	Contributors
66599	122987	310



The Lean Zulip Channel – https://leanprover.zulipchat.com

condensed mathematics > Condensed R-modules — Oct 07

Peter Scholze (EDITED)

My math understanding is that `Condensed Ab.{u+1}` ought to be functors from `Profinite.{u}` to `Ab.{u+1}`, and then the index set `J` that appears will be, for a presheaf F, the disjoint union over all isomorphism classes of objects S of `Profinite.{u}` of $F(S)$. Now in ZFC universes, this disjoint union still lies in the `u+1` universe.

But what you say above indicates that this is also true, as long as the index set of S's is still in universe `u`. Well, it isn't quite -- it's a bit larger, but still much smaller than `u+1` in terms of ZFC universes.

So maybe that it helps to take instead functors from `Profinite.{u}` to `Ab.{u+2}`? Then I'm pretty sure `Profinite.{u}` lies in `Type.{u+1}`, so that disjoint union of $F(S)$'s above should lie in `Type.{u+2}`, and this should be good enough.

lean-gptf > OpenAI gpt-f key — Oct 08

Stanislas Polu — 6:03 AM

@Ayush Agrawal let me check 👍

👍 1

6:33 AM

We had a bit of a backlog
Good think you reached out. Invites are out.

6:34 AM

But! Note that the model is quite stale. We're working on updating it, but don't be surprised if it's not super useful as it was trained on a rather old snaphost of mathlib

👍 1

FLT regular > Cyclotomic field defn — Oct 25

Eric Rodriguez — 10:09 AM

I noticed this project so far is working with `adjoin_root cyclotomic`. I wonder if instead, `X^n-1.splitting_field` is a better option. I think the second option is better suited to Galois theory (as then the `.gal` has good defeq) and also easier to generalise to other fields. (it works for all fields with n ≠ 0, whilst I think this one may not)

new members > ✓ ∀ x y z : A, x ≠ y → (x ≠ z ∨ y ≠ z) :=

Jia Xuan Ng (EDITED)

Hi everyone, I'm trying to prove ∀ x y z : A, x ≠ y → (x ≠ z ∨ y ≠ z) :=, which I believe to be provable.
Reason why this is is because I use implication logical equivalences e.g. P -> Q === !P V Q such that I derived:
x ≠ y → ¬ (x ≠ z) → y ≠ z ==> x ≠ y → x = z → y ≠ z which is essentially stating:
"If x isn't equivalent to y, if x is equivalent to z, then y isn't equivalent to z", which is a tautology.

However, I just can't seem to do anything... thank you very much.

general > Bachelor thesis accomplished🎉 — Today

Giacomo Maletto — 9:52 AM

Hello, I'm a math student at University of Turin and I've been using proof assistants for about a year, with the objective of formalizing a computer science paper written by my advisor (about a class of functions similar in spirit to primitive recursive functions, but which are all invertible).

After a lot of work here's my thesis! https://github.com/GiacomoMaletto/RPP/blob/main/Tesi/main.pdf (Lean code in the same repo).
It's written in an informal, colloquial manner and I tried to turn it into an introduction/invitation to Lean.

Actually I've used Coq for 90% of the duration of the project, completed it, and then switched to Lean - doing basically the same thing in about 750 LOC instead of >3000. I'm not turning back.

Looking forward to start using Lean for something more involved!

🎉 17 1

Focused Research Organization (FRO)

A new type of nonprofit startup for science developed by Convergent Research.

convergentresearch.org

The Lean FRO

Mission: address **scalability**, **usability**, and **proof automation** in Lean

We want to popularize formal mathematics and verification.

7 FTEs by end of year

Supported by Simons Foundation International, Alfred P. Sloan Foundation, and Richard Merkin

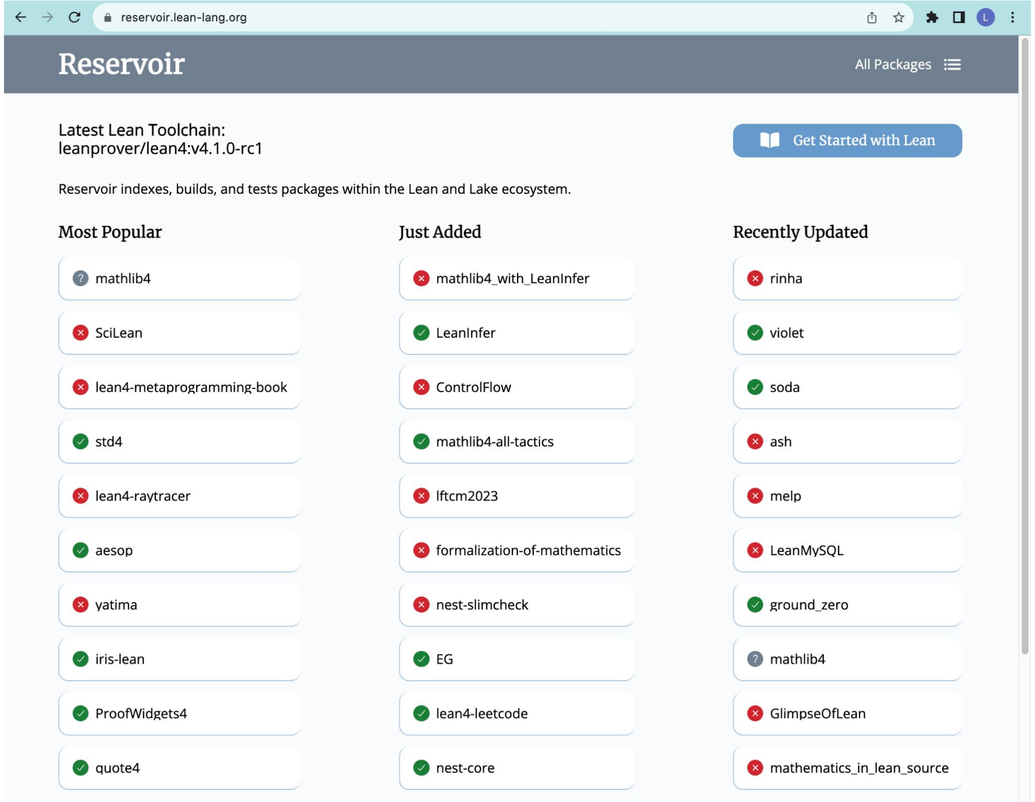
lean-fro.org

Questions of Scale

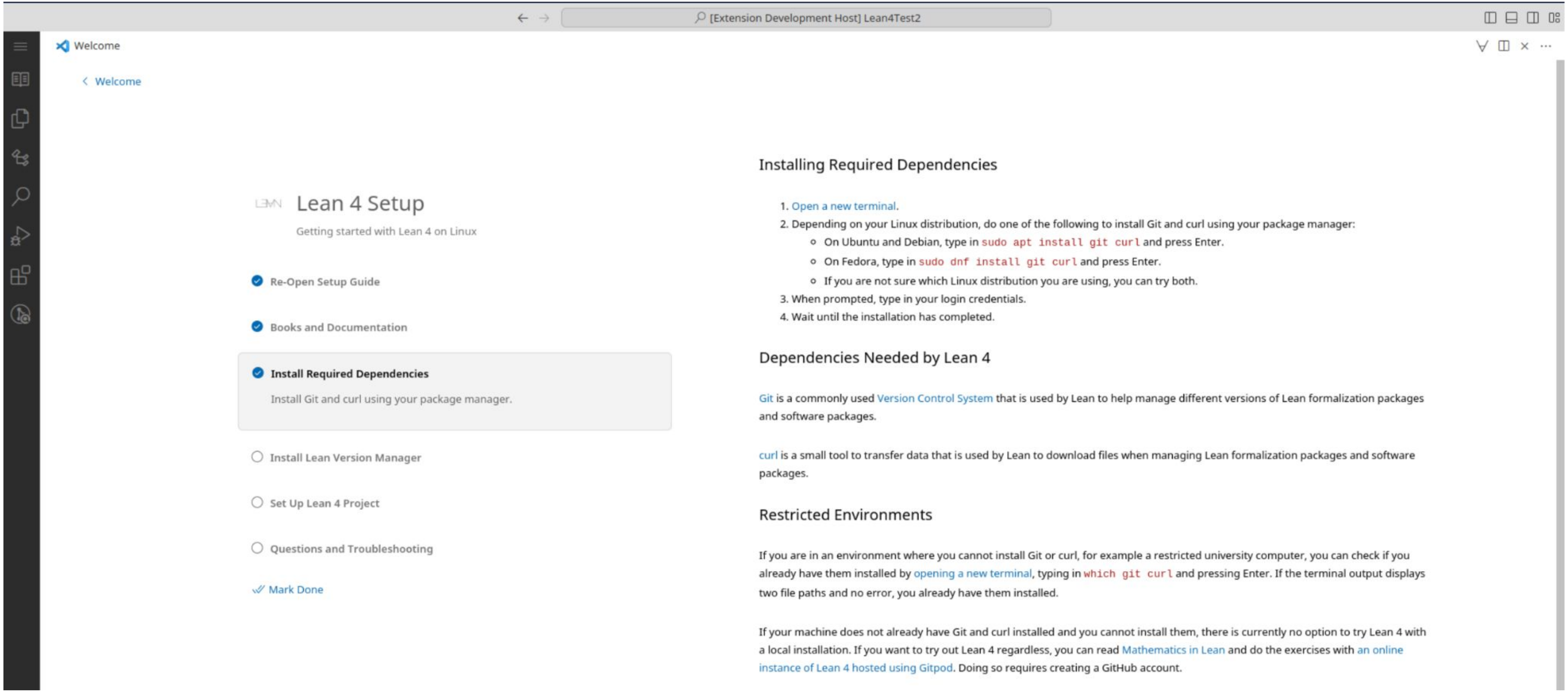
“Can mathlib scale to 100 times its present size, with a community 100 times its present size and commits going in at 100 times the present rate? [...] Will the proofs be maintained afterwards [...]?”

– Joseph Myers on Lean Zulip

So many new features in the "oven"



So many new features in the “oven”



Loogle

loogle.lean-fro.org/?q=Real.sin%2C+Real.cos%2C+%28_+%5E+2%29+%2B+%28_+%5E+2%29

Loogle!

Real.sin, Real.cos, (_ ^ 2) + (_ ^ 2)

#find

Result

Found 13 definitions mentioning Real.cos, HPow.hPow, HAdd.hAdd, OfNat.ofNat and Real.sin. Of these, 2 match your patterns.

- Real.cos_sq_add_sin_sq Mathlib.Data.Complex.Exponential
- Real.sin_sq_add_cos_sq Mathlib.Data.Complex.Exponential

```
@[simp]                                                   source
theorem Real.sin_sq_add_cos_sq
      (x : ℝ) :
  Real.sin x ^ 2 + Real.cos x ^ 2 = 1
```

Loogle

mathlib4 > toFinsetFactors SEP 14

Arend Mellendijk 12:20 PM

@loogle Nat.factors, List.toFinset

loogle 12:20 PM

🔍 Nat.prime_divisors_eq_to_filter_divisors_prime, Nat.factors_mul_toFinset, and 13 more

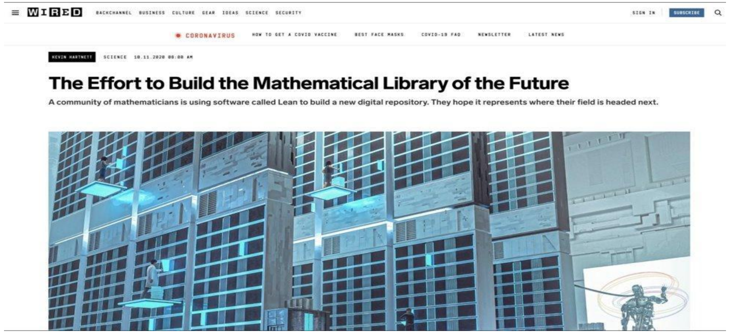
Antoine Chambert-Loir 12:32 PM

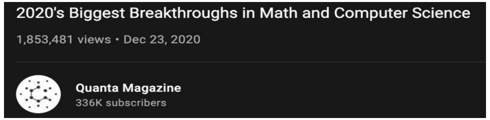
FactorsFinset

Eric Wieser EDITED 12:39 PM

I'd be tempted to make this an `abbrev` so that it doesn't cost anything

The Lean Mathematical Library goes viral – 2020





“You can do 14 hours a day in it and not get tired and feel kind of high the whole day,” Livingston said. “You’re constantly getting positive reinforcement.”

“It will be so cool that it’s worth a big-time investment now,” Macbeth said. “I’m investing time now so that somebody in the future can have that amazing experience.”

The Liquid Tensor Experiment (LTE) – 2021

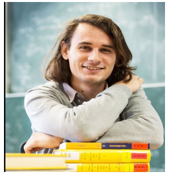
Peter Scholze (Fields Medal 2018) was unsure about one of his latest results in Analytic Geometry.

The Lean community and Scholze formalized the result he was unsure about.

We thought it would take years (Scholze included).

Trust agnostic collaboration allowed us to achieve it in months. (Math Hive in action).

"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*



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

2023 has been a great year for LEAN

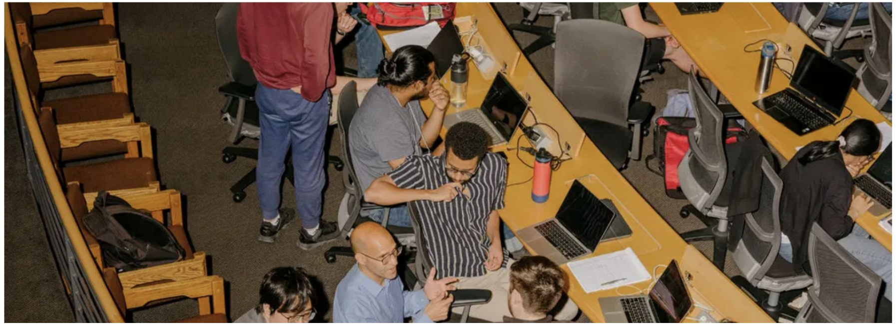
The New York Times

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

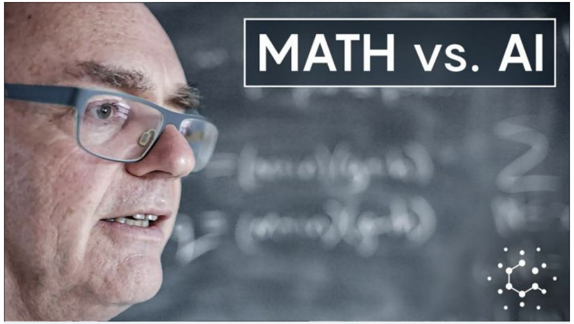
For thousands of years, mathematicians have adapted to the latest advances in logic and reasoning. Are they ready for artificial intelligence?

Give this article



Terence Tao
@tao@mathstodon.xyz

Leo de Moura surveyed the features and use cases for Lean 4. I knew it primarily as a formal proof assistant, but it also allows for less intuitive applications, such as truly massive mathematical collaborations on which individual contributions do not need to be reviewed or trusted because they are all verified by Lean. Or to give a precise definition of an extremely complex mathematical object, such as a perfectoid space.

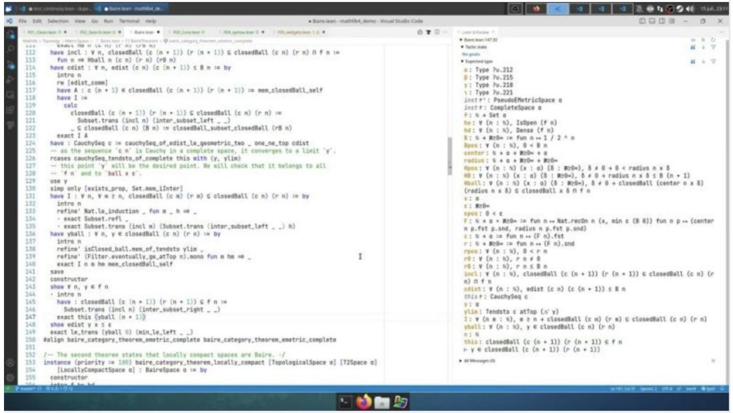


When Computers Write Proofs, What's the Point of Mathematicians?
youtube.com

2023 has been a great year for LEAN

Leonardo de Moura (He/Him) • You
Senior Principal Applied Scientist at AWS, and Chief Architect ...
1mo •

I am thrilled to announce that the Mathlib (https://lnkd.in/gx6eh4aG) port to Lean 4 has been successfully completed this weekend. It is truly remarkable that over 1 million lines of formal mathematics have been successfully migrated. Once again, the community has amazed me and surpassed all my expectations. This achievement also aligns with the 10th anniversary of my initial commit to Lean on July 15, 2013. Patrick Massot has graciously shared a delightful video commemorating this significant milestone, which can be viewed here: https://lnkd.in/gjVr72t8.



Lean 4 overview for Mathlib users - Patrick Massot
youtube.com

Leonardo de Moura (He/Him) • You
Senior Principal Applied Scientist at AWS, and Chief Architect ...
1mo •

Ecstatic to come across the following post today! 😃 Here is the link to the original: https://lnkd.in/dSDFSVhS, and website: https://lnkd.in/dB9427pU

Daniel J. Bernstein
@djb@cr.yp.to

Formally verified theorems about decoding Goppa codes: cr.yp.to/2023/leangoppa-202307... This is using the Lean theorem prover; I'll try formalizing the same theorems in HOL Light for comparison. This is a step towards full verification of fast software for the McEliece cryptosystem.

Graydon Hoare
@graydon@types.pl

I fairly often find myself in conversations with people who wish Rust had more advanced types. And I always say it's pretty much at its cognitive-load and compatibility induced design limit, and if you want to go further you should try building a newer language. And many people find this answer disappointing because starting a language is a long hard task especially if it's to be a sophisticated one. And so people ask for a candidate project they might join and help instead. And my best suggestion for a while now has been Lean 4. I think it's really about the best thing going in terms of powerful research languages. Just a remarkable achievement on many many axes.

Extensibility

We build **with (not for)** the community

Mathlib is not just math, but many Lean extensions too.

The community extends Lean using Lean itself.

We wrote Lean 4 in Lean to make sure every single part of the system is extensible.

```
elab "ring" : tactic => do
  let g ← getMainTarget
  match g.getAppFnArgs with
  | (`Eq, #[ty, e₁, e₂]) =>
    let ((e₁', p₁), (e₂', p₂)) ← RingM.run ty $ do (← eval e₁, ← eval e₂)
    if ← isDefEq e₁' e₂' then
      let p ← mkEqTrans p₁ (← mkEqSymm p₂)
      ensureHasNoMVars p
      assignExprMVar (← getMainGoal) p
      replaceMainGoal []
    else
      throwError "failed \n{← e₁'.pp}\n{← e₂'.pp}"
  | _ => throwError "failed: not an equality"
```

Lean 4 is implemented in Lean

```
inductive Expr where
  | bvar (deBruijnIndex : Nat)
  | fvar (fvarId : FVarId)
  | mvar (mvarId : MVarId)
  | sort (u : Level)
  | const (declName : Name) (us : List Level)
  | app (fn : Expr) (arg : Expr)
  | lam (binderName : Name) (binderType : Expr) (body : Expr) (binderInfo : BinderInfo)
  | forallE (binderName : Name) (binderType : Expr) (body : Expr) (binderInfo : BinderInfo)
  | letE (declName : Name) (type : Expr) (value : Expr) (body : Expr) (nonDep : Bool)
  | lit : Literal → Expr
  | mdata (data : MData) (expr : Expr)
  | proj (typeName : Name) (idx : Nat) (struct : Expr)
```

The Lean 4 Frontend Pipeline

- parser: ≈ `String → Syntax`
- macro expansion: `Syntax → MacroM Syntax`
  - actually interleaved with elaboration
- elaboration
  - terms: `Syntax → TermElabM Expr`
  - commands: `Syntax → CommandElabM Unit`
  - universes: `Syntax → TermElabM Level`
  - tactics: `Syntax → TacticM Unit`

The Lean 4 Frontend Pipeline

- parser: `≈ String → Syntax`
- macro expansion: `Syntax → MacroM Syntax`
  - actually interleaved with elaboration
- elaboration
  - terms: `Syntax → TermElabM Expr`
  - commands: `Syntax → CommandElabM Unit`
  - universes: `Syntax → TermElabM Level`
  - tactics: `Syntax → TacticM Unit`
- pretty printer
  - delaborator: `Expr → DelaboratorM Syntax`
  - parenthesizer: `Syntax → ParenthesizerM Syntax`
  - formatter: `Syntax → FormatterM Format`

Macro: simple extensions must be simple!

```
infixl:65   " + "  => Add.add  -- left associative
infix:65    " - "  => Sub.sub  -- ditto
infixr:80   " ^ "  => Pow.pow  -- right associative
prefix:100  "-"    => Neg.neg
postfix:arg "⁻¹"   => Inv.inv
```

Macro: simple extensions must be simple!

```
infixl:65   " + "  => Add.add  -- left associative
infix:65    " - "  => Sub.sub  -- ditto
infixr:80   " ^ "  => Pow.pow  -- right associative
prefix:100  "-"    => Neg.neg
postfix:arg "⁻¹"   => Inv.inv
```

These are just macros!

```
notation:65 lhs " + " rhs:66 => Add.add lhs rhs
notation:65 lhs " - " rhs:66 => Sub.sub lhs rhs
notation:80 lhs " ^ " rhs:80 => Pow.pow lhs rhs
notation:100 "-" arg:100     => Neg.neg arg
notation:arg arg "⁻¹"        => Inv.inv arg
```

Mixfix Notation

```
notation:arg "(" e ")" => e
notation:10 Γ " ⊢ " e " : " t => Typing Γ e t
```

Mixfix Notation

```
notation:arg "(" e ")" => e
notation:10 Γ " ⊢ " e " : " t => Typing Γ e t
```

Overlapping notations are parsed with a (long) “longest parse” rule

```
notation:65 a " + " b:66 " + " c:66 => a + b - c
#eval 1 + 2 + 3 -- 0

theorem bad : 1 + 2 + 3 = 0 := by
  rfl
```

Mixfix Notation

```
notation:arg "(" e ")" => e
notation:10 Γ " ⊢ " e " : " t => Typing Γ e t
```

Overlapping notations are parsed with a (long) “longest parse” rule

```
notation:65 a " + " b:66 " + " c:66 => a + b - c
#eval 1 + 2 + 3 -- 0

theorem bad : 1 + 2 + 3 = 0 := by
  rfl
```

▼Tactic state

1 goal

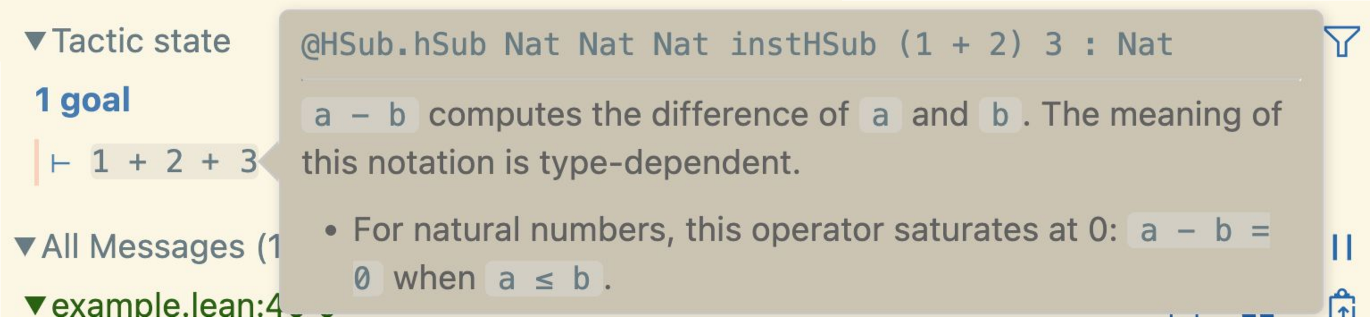
```
⊢ 1 + 2 - 3 = 0
```

Mixfix Notation

Overlapping notations are parsed with a (long) “longest parse” rule

```
notation:65 a " + " b:66 " + " c:66 => a + b - c
#eval 1 + 2 + 3 -- 0

theorem bad : 1 + 2 + 3 = 0 := by
  rfl
```



Syntax

```
notation:arg "(" e ")" => e
```

This is just a macro!

```
syntax:arg "(" term ")" : term
macro_rules
  | `(($e)) => `($e)
```

`term` is a syntax category.

Syntax

```
notation:arg "(" e ")" => e
```

This is just a macro!

```
syntax:arg "(" term ")" : term
macro_rules
  | `(($e)) => `($e)
```

`term` is a syntax category.

```
declare_syntax_cat index
syntax term : index
syntax term " ≤ " ident " < " term : index
syntax term " : " term : index

syntax "{" index " | " term "}" : term
```

More Syntax

```
syntax binderId := ident <|> "_"
syntax unbracketedExplicitBinders := binderId+ (" : " term)?

syntax "begin " tactic,*,? "end" : tactic
```

Summary: Parsing

Each syntax category is

- a precedence (Pratt) parser composed of a set of leading and trailing parsers
- with per-parser precedences
- following the longest parse rule

Macros

```
notation:arg "(" e ")" => e
```

This is just a macro.

```
syntax:arg "(" term ")" : term
macro_rules
  | `(($e)) => `($e)
```

which can also be written as

```
macro:arg "(" e:term ")" : term => `($e)
```

Macros

```
notation:arg "(" e ")" => e
```

This is just a macro.

```
syntax:arg "(" term ")" : term
macro_rules
  | `(($e)) => `($e)
```

which can also be written as

```
macro:arg "(" e:term ")" : term => `($e)
```

or, in this case

```
macro:arg "(" e:term ")" : term => return e
```

Quotations

```
`(let $id:ident $[$binders]* $[: $ty?]? := $val; $body)
```

has type `Syntax` in patterns.
has type `m Syntax` given `MonadQuotation m` in terms.
`id` has type `TSyntax `ident`.
`val` and `body` have type `TSyntax `term`.

Quotations

```
`(let $id:ident $[$binders]* $[: $ty?]? := $val; $body)
```

has type `Syntax` in patterns.
has type `m Syntax` given `MonadQuotation m` in terms.
`id` has type `TSyntax `ident`.
`val` and `body` have type `TSyntax `term`.
`binders` has type `Array (TSyntax `letIdBinder)`.
`ty?` has type `Option (TSyntax `term)`.

Scope of Hygiene

```
macro "foo" : term => do
  let a ← `(rfl)
  `(fun rfl => $a)
```

This unfolds to the identity function. Hygiene works *per-macro.*

Scope of Hygiene

```
macro "foo" : term => do
  let a ← `(rfl)
  `(fun rfl => $a)
```

This unfolds to the identity function. Hygiene works *per-macro.*
Nested scopes can be opened with `withFreshMacroScope`.

```
destruct (as : List Var) (x : Syntax) (body : Syntax) : MacroM Syntax := do
  match as with
    | [a, b]  => `(let $a:ident := $x.1; let $b:ident := $x.2; $body)
    | a :: as => withFreshMacroScope do
      let rest ← destruct as (← `(x)) body
      `(let $a:ident := $x.1; let x := $x.2; $rest)
    | _ => unreachable!
```

Summary: Macros

Macros are syntax-to-syntax translations

- applied iteratively and recursively
- associated with a specific parser and tried in a specific order
- with “well-behaved” (hygienic) name capturing semantics

Unexpanders: simple pretty printers

```
inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  /-- Existential introduction. If `a : α` and `h : p a`,
  then `⟨a, h⟩` is a proof that `∃ x : α, p x`. -/
  | intro (w : α) (h : p w) : Exists p
```

```
macro "∃" xs:explicitBinders ", " b:term : term => expandExplicitBinders ``Exists xs b
```

Unexpanders: simple pretty printers

```
inductive Exists {α : Sort u} (p : α → Prop) : Prop where
  /-- Existential introduction. If `a : α` and `h : p a`,
  then `⟨a, h⟩` is a proof that `∃ x : α, p x`. -/
  | intro (w : α) (h : p w) : Exists p
```

```
macro "∃" xs:explicitBinders ", " b:term : term => expandExplicitBinders ``Exists xs b
```

```
@[app_unexpander Exists] def unexpandExists : Lean.PrettyPrinter.Unexpander
  | `($(_) fun $x:ident => ∃ $xs:binderIdent*, $b) => `(∃ $x:ident $xs:binderIdent*, $b)
  | `($(_) fun $x:ident => $b)                      => `(∃ $x:ident, $b)
  | `($(_) fun ($x:ident : $t) => $b)               => `(∃ ($x:ident : $t), $b)
  | _                                               => throw ()
```

Lean is a platform for Domain-Specific Languages (DSLs)

Extensible syntax.

Hygienic macros.

Extensible elaborator & pretty printer.

You can design DSLs, write code using them, and reason about this code.

Extensible LSP server coming soon.

String Interpolation: a micro DSL

```
def foo (x : Nat) : IO Unit :=
  let y := x + 1
  IO.println s!"x: {x}, y: {y}"

#eval foo 5
-- x: 5, y: 6
```

→

```
"x: " ++ toString x ++ ", y: " ++ toString y
```

Started as a Lean example!

String Interpolation: a micro DSL

```
def foo (x : Nat) : IO Unit :=
  let y := x + 1
  IO.println s!"x: {x}, y: {y}"

#eval foo 5
-- x: 5, y: 6
```

→

```
"x: " ++ toString x ++ ", y: " ++ toString y
```

Started as a Lean example!

```
partial def interpolatedStrFn (p : ParserFn) : ParserFn := fun c s =>
  let input     := c.input
  let stackSize := s.stackSize
  let rec parse (startPos : String.Pos) (c : ParserContext) (s : ParserState) : ParserState :=
    let i     := s.pos
    if input.atEnd i then
      let s := s.pushSyntax Syntax.missing
      let s := s.mkNode interpolatedStrKind stackSize
      s.setError "unterminated string literal"
    else
      let curr := input.get i
      let s    := s.setPos (input.next i)
      if curr == '\"' then
        let s := mkNodeToken interpolatedStrLitKind startPos c s
        s.mkNode interpolatedStrKind stackSize
```

“do” notation: another DSL

Introduced by the Haskell programming language.

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

↓

```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

Lean has many extensions: nested actions, reassignments, for-loops, etc.

“do” notation: another DSL

```
def Poly.eval? (e : Poly) (a : Assignment) : Option Rat := Id.run do
  let mut r := 0
  for (c, x) in e.val do
    if let some v := a.get? x then
      r := r + c*v
    else
      return none
  return r
```

Using “do” notation to expand interpolated string notation

```
def expandInterpolatedStrChunks (chunks : Array Syntax) (mkAppend : Syntax → Syntax → MacroM Syntax)
                                (mkElem : Syntax → MacroM Syntax) : MacroM Syntax := do
  let mut i := 0
  let mut result := Syntax.missing
  for elem in chunks do
    let elem ← match elem.isInterpolatedStrLit? with
      | none     => mkElem elem
      | some str => mkElem (Syntax.mkStrLit str)
    if i == 0 then
      result := elem
    else
      result ← mkAppend result elem
    i := i+1
  return result

def expandInterpolatedStr (interpStr : TSyntax interpolatedStrKind) (type : Term) (toTypeFn : Term) : MacroM Term := d
  let r ← expandInterpolatedStrChunks interpStr.raw.getArgs (fun a b => `($a ++ $b)) (fun a => `($toTypeFn $a))
  `(($r : $type))
```

Extending the anonymous constructor notation

Anonymous constructor notation for inductive types with one constructor.

```
structure Person where
  name : String
  age  : Nat

def mkPerson (n : String) (a : Nat) : Person :=
  ⟨n, a⟩

theorem mkAndSelf {p : Prop} (h : p) : p ∧ p :=
  ⟨h, h⟩

example : 1 = 1 ∧ 1 = 1 :=
  mkAndSelf (Eq.refl 1)
```

Extending the anonymous constructor notation

Let's define a notation that tries to find a constructor with the right number of arguments.

```
import Lean

syntax (name := anonCtorExt) "⟪ " term,*,? " ⟫" : term
```

Extending the anonymous constructor notation

```
syntax (name := anonCtorExt) "《 " term,*,? " 》" : term
```

```
open Lean Meta Elab Term in
@[term_elab anonCtorExt] def elabAnonCtorExt : TermElab := fun stx expectedType? => do
  match stx with
  | `(《 $[$args],* 》) =>
    for ctorName in (← getCtors expectedType?) do
      let ctorInfo ← getConstInfoCtor ctorName
      if ctorInfo.numFields == args.size then
        let newStx ← `($(mkCIdentFrom stx ctorName) $(args)*)
        return (← withMacroExpansion stx newStx (elabTerm newStx expectedType?))
    throwError "did not find compatible constructor"
  | _ => throwUnsupportedSyntax
where
  getCtors (expectedType? : Option Expr) : MetaM (List Name) := do
    let some type := expectedType? | throwError "expected type is not known"
    let .const declName .. := (← whnf type).getAppFn | throwError "inductive expected"
    let .inductInfo val ← getConstInfo declName | throwError "inductive expected"
    return val.ctors
```

Extending the anonymous constructor notation

```
let a : Unit := ⟨⟩
let b : List Nat := ⟨⟩
let c : List Nat := ⟨2, b⟩
let d : List Nat := ⟨1, c,⟩
have : b = [] := rfl
have : c = [2] := rfl
have : d = [1, 2] := rfl
```

Extending the anonymous constructor notation

inductive expected Lean 4

View Problem (⌥F8) No quick fixes available

```
def aList (
  let a := ⟪1, b⟫
  a ++ a
```

Extending the anonymous constructor notation

```
open Lean Meta Elab Term in
@[term_elab anonCtorExt] def elabAnonCtorExt : TermElab := fun stx expectedType? => do
  match stx with
  | `(⟨ $[$args],* ⟩) =>
    tryPostponeIfNoneOrMVar expectedType? ←
    for ctorName in (← getCtors expectedType?) do
```

...

```
def aList (b : List Nat) : List Nat :=
  let a := ⟨1, b⟩
  a ++ a
```

Interactive Tactics: another DSL

```
theorem State.erase_le_of_le_cons (h : σ' ≼ (x, v) :: σ) : σ'.erase x ≼ σ := by
  intro y w hf'
  by_cases hxy : x = y <;> simp [*] at hf'
  have hf := h hf'
  simp [hxy, Ne.symm hxy] at hf
  assumption
```

Interactive Tactics: another DSL

```
@[builtin_tactic Lean.Parser.Tactic.intro] def evalIntro : Tactic := fun stx => do
  match stx with
  | `(tactic| intro)                   => introStep none `_
  | `(tactic| intro $h:ident)          => introStep h h.getId
  | `(tactic| intro _%$tk)             => introStep tk `_
  /- Type ascription -/
  | `(tactic| intro ($h:ident : $type:term)) => introStep h h.getId type
  /- We use `@h` at the match-discriminant to disable the implicit lambda feature -/
  | `(tactic| intro $pat:term)         => evalTactic (← `(tactic| intro h; match @h with | $pat:term => ?_; try clear h))
  | `(tactic| intro $h:term $hs:term*) => evalTactic (← `(tactic| intro $h:term; intro $hs:term*))
  | _ => throwUnsupportedSyntax
```

Conclusion

LEAN is an extensible theorem prover. http://leanprover.github.io

Decentralized collaboration.

The Mathlib community will change how mathematics is done and taught.

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 power.