Lean 4: an extensible proof assistant and programming language

Leonardo de Moura
Senior Principal Applied Scientist - AWS
Chief Architect - Lean FRO
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 an 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 $\Rightarrow$ kernel code

```
5 theorem euclid_exists_infinite_primes (n : ℕ) : ∃ p, n ≤ p ∧ Prime p :=
6  let p := minFac (factorial n + 1)
7  have f1 : (factorial n + 1) ≠ 1 :=
8      ne_of_gt $ succ_lt_succ $ factorial_pos _
9  have pp : Prime p :=
10     min_fac_prime f1
11  have np : n ≤ p := le_of_not_ge fun h =>
12      have h1 : p | factorial n := dvd_factorial (min_fac_pos _) h
13      have h2 : p | 1 := (Nat.dvd_add_iff_right h1).2 (min_fac_dvd _)
14      pp.not_dvd_one h2
15  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"
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.*
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

<table>
<thead>
<tr>
<th>Definitions</th>
<th>Theorems</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>66599</td>
<td>122987</td>
<td>310</td>
</tr>
</tbody>
</table>

Number of lines
The Lean Zulip Channel - [https://leanprover.zulipchat.com](https://leanprover.zulipchat.com)
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](http://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!

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

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]

theorem Real.sin_sq_add_cos_sq
(x : ℝ) :
  Real.sin x ^ 2 + Real.cos x ^ 2 = 1
I'd be tempted to make this an `abbrev` so that it doesn't cost anything
“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
2023 has been a great year for Lean

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?

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

MATH vs. AI
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.

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/dB94z7pU

Daniel J. Bernstein
@djbernstein

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 disappointed 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.

```lean
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

```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: \(\sim\) String \(\rightarrow\) Syntax
- macro expansion: Syntax \(\rightarrow\) MacroM Syntax
  - actually interleaved with elaboration
- elaboration
  - terms: Syntax \(\rightarrow\) TermElabM Expr
  - commands: Syntax \(\rightarrow\) CommandElabM Unit
  - universes: Syntax \(\rightarrow\) TermElabM Level
  - tactics: Syntax \(\rightarrow\) TacticM Unit
The Lean 4 Frontend Pipeline

- parser: $\approx$ String $\rightarrow$ Syntax

- macro expansion: Syntax $\rightarrow$ MacroM Syntax
  - actually interleaved with elaboration

- elaboration
  - terms: Syntax $\rightarrow$ TermElabM Expr
  - commands: Syntax $\rightarrow$ CommandElabM Unit
  - universes: Syntax $\rightarrow$ TermElabM Level
  - tactics: Syntax $\rightarrow$ TacticM Unit

- pretty printer
  - delaborator: Expr $\rightarrow$ DelaboratorM Syntax
  - parenthesizer: Syntax $\rightarrow$ ParenthesizerM Syntax
  - formatter: Syntax $\rightarrow$ 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
```

- For natural numbers, this operator saturates at 0: \( a - b = 0 \) when \( a \leq b \).
Syntax

term is a syntax category.

```
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

```plaintext
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

\[
\text{notation:arg } "(" e ")" \Rightarrow e
\]

This is just a macro.

\[
\text{syntax:arg } "(" \text{ term } ")" : \text{ term}  \\
\text{macro_rules}  \\
\text{| } \text{ `((e)) } \Rightarrow \text{ `($e}\text{)"
\]

which can also be written as

\[
\text{macro:arg } "(" \text{ e:term } ")" : \text{ term} \Rightarrow \text{ `($e}
\]
This is just a macro.

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

which can also be written as

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

or, in this case

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

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

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

```haskell
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`
```

```
  | `(\$_ 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

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

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

Started as a Lean example!
String Interpolation: a micro DSL

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

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

Started as a Lean example!

```lean
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.setPosition (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.

```
{ 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

```haskell
  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 := do
  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.

```lean
import Lean

syntax (name := anonCtorExt) "\{ " term,*,? " }" : term
```
Extending the anonymous constructor notation

```lean
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 (∀ get Ctors 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"
  _ => throwError "unsupported syntax"
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

```ocaml
let a : Unit := ()
let b : List Nat := []
let c : List Nat := [2, b]
let d : List Nat := [1, c,
have : b = [] := refl
have : c = [2] := refl
have : d = [1, 2] := refl
```
Extending the anonymous constructor notation

```lean
def aList (b: 
let a := ⟨1, b⟩
a ++ a
```

inductive expected Lean 4

View Problem (\F8) No quick fixes available
Extending the anonymous constructor notation

```lean
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

...`
Interactive Tactics: another DSL

```lean
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

```lean
  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](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.