# The `grind` Tactic: proof automation in Lean

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

September 18, 2025

# Lean is an open-source programming language and proof assistant.

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

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

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

lean-lang.org

# Lean is based on dependent type theory

An example *by Kim Morrison*:

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

Full example here.

An example *by Kim Morrison*:

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

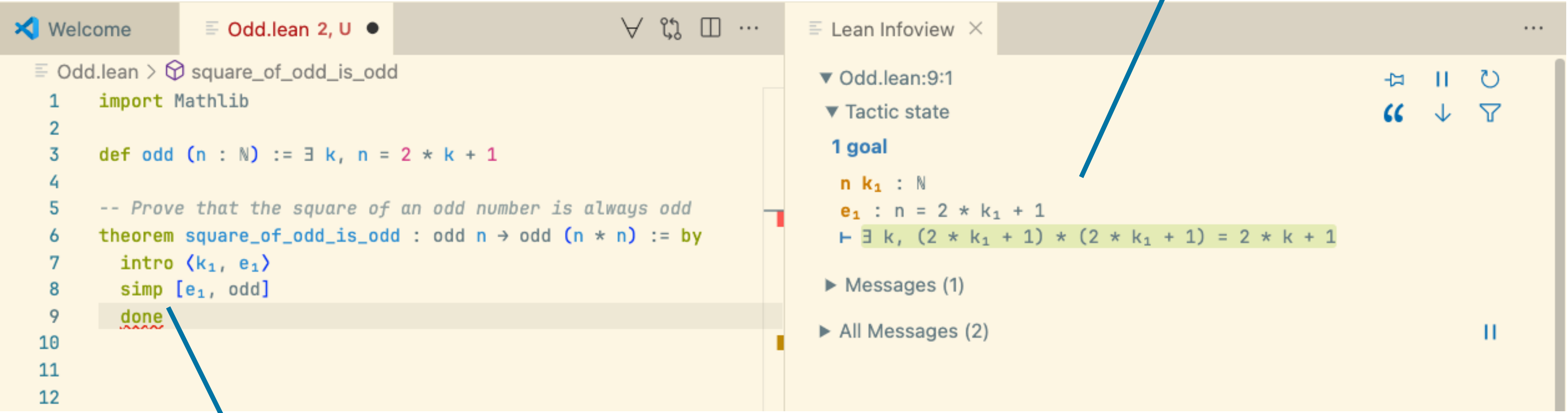
```
def insert [LawfulBEq α] (m : IndexMap α β) (a : α) (b : β) : IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    { indices := m.indices
      keys := m.keys.set i a
      values := m.values.set i b }
  | none =>
    { indices := m.indices.insert a m.size
      keys := m.keys.push a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

An example *by Kim Morrison*:

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

attribute [local grind] getIdx findIdx insert

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

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

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

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

# Theorem proving in Lean is an interactive game

The "game board"




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

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

**User-driven design philosophy**: **We Listen to Our Users.**

Classical logic and mathematics as defaults.

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

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

**Extensibility**. You can make Lean your own.

**Exceptional tooling**. Linters, CI, UX, Build System, Caches. Maintenance is the Grand Challenge.

All components work together as a **unified system.**

# Proof Automation

## Why do we need proof automation?

"I thought AI would prove all theorems for us now."

**AI at the IMO 2024**

AlphaProof (Google DeepMind) achieved silver medal level using Lean.

**AI at the IMO 2025**

Google DeepMind and OpenAI achieved gold medal level using informal reasoning.

ByteDance achieved silver* medal using Lean. (*) They reached gold after the competition.

Harmonic achieved gold medal using Lean.

## AI is playing the “Lean game"

The **moves** in this game are **tactics** from Automated Reasoning:  good old proof automation.

Here are some “moves” played by AlphaProof:

```
simp_all[Finset.sum_range_id]

zify[*]at*

norm_num at*

nlinarith[(by norm_cast:(c:ℝ)>=A*(1-⌊_⌋)+⌊_⌋+1),Int.floor_lex,Int.lt_floor_add_one x]
```

Even the most advanced AI relies on the same tactics we use every day.

By developing **better moves/tactics**, we enable even **more powerful AI.**

# Why is Proof Automation Hard in Lean?

Dependent Types: more expressive, but harder to automate.

Example: given

```
def Array.get {α : Type u} (as : Array α) (i : Nat) (h : i < as.size) : α
```

Suppose we want to rewrite/simplify

```
Array.get as (2 + i - 1) h
```

and can easily construct a proof that `2 + i - 1 = i + 1`, but the following term is not type correct.

```
Array.get as (i+1) h
```

Lean generates custom congruence theorems that "patch" the proof term.

```
theorem Array.get.congr_simp' {α : Type u} (as as' : Array α) (i i' : Nat) (h : i < as.size)
      (h₁ : as = as') (h₂ : i = i')
      : Array.get as i h = Array.get as' i' (h₁ ▸ h₂ ▸ h) := by
```

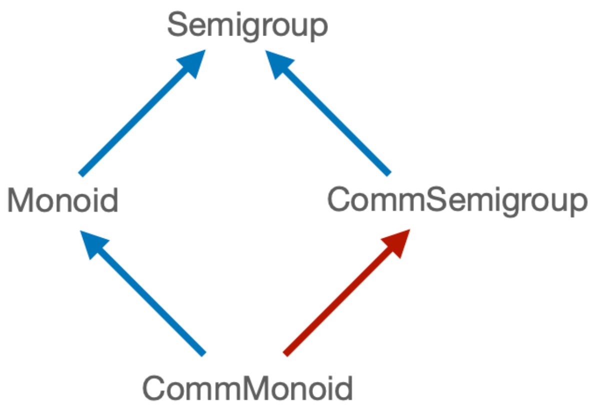
# Type Classes

Type classes provide an elegant mechanism for managing ad-hoc polymorphism.

```
class Mul (α : Type u) where
  mul : α → α → α

#check @Mul.mul    @Mul.mul : {α : Type u_1} → [self : Mul α] → α → α → α

instance : Mul Nat where
  mul := Nat.mul
instance : Mul Int where
  mul := Int.mul

def n : Nat := 1
def i : Int := -2

set_option pp.explicit true
#check Mul.mul n n    @Mul.mul Nat instMulNat n n : Nat
#check Mul.mul i i    @Mul.mul Int instMulInt i i : Int
infix:65 (priority := high) "*" => Mul.mul
#check n*n    @Mul.mul Nat instMulNat n n : Nat
#check i*i    @Mul.mul Int instMulInt i i : Int
```

# Type Classes

```
class Semigroup (α : Type u) extends Mul α where
  mul_assoc (a b c : α) : a * b * c = a * (b * c)

instance : Semigroup Nat where
  mul_assoc := Nat.mul_assoc

instance : Semigroup Int where
  mul_assoc := Int.mul_assoc

class CommSemigroup (α : Type u) extends Semigroup α where
  mul_comm (a b : α) : a * b = b * a

class Monoid (α : Type u) extends Semigroup α, One α where
  one_mul (a : α) : 1 * a = a
  mul_one (a : α) : a * 1 = a

class CommMonoid (α : Type u) extends Monoid α, CommSemigroup α where

class NoZeroDivisors (α : Type u) [Mul α] [Zero α] where
  no_zero_div (a b : α) : a ≠ 0 → a * b = 0 → b = 0
```



# Type Classes

There approx. **1.5K classes and 20K instances** in Mathlib.

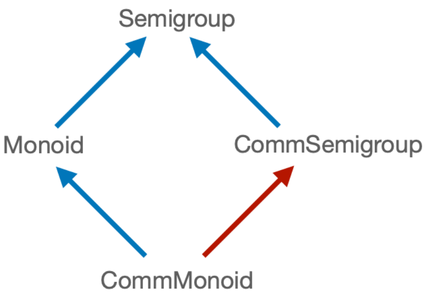
Type class resolution is backward chaining.

You can view instances as **Horn Clauses**.

```
instance [Semiring α] [AddRightCancel α] [NoNatZeroDivisors α] : NoNatZeroDivisors (OfSemiring.Q α) where
```

Lean procedure is based on **tabled resolution**.

Proof automation must be able to detect that different synthesized instances are definitionally equal.



# Type Classes

As Russian dolls.

```
/-- A `LieAdmissibleAlgebra` is a `LieAdmissibleRing` equipped with a compatible action by scalars
from a commutative ring. -/
@[ext]
class LieAdmissibleAlgebra (R L : Type*) [CommRing R] [LieAdmissibleRing L]
  extends Module R L, IsScalarTower R L L, SMulCommClass R L L
```

# Does Lean Have Hammers?

The Lean community is also actively developing automation.

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

Duper: a superposition theorem prover written in Lean for proof reconstruction.

bv_decide: fastest verified bit-blaster. Uses the CaDiCaL SAT Solver. Bit-blaster, AIG, and LRAT SAT proof checkers are all implemented and verified in Lean.

Lean-SMT: An SMT tactic for discharging proof goals in Lean  UFMG, Stanford, University of Iowa

Lean-Auto: Interface between Lean and automated provers. Yicheng Qian (CMU and Stanford).

Lean-auto is based on a monomorphization procedure from dependent type theory to higher-order logic and a deep embedding of higher-order logic into dependent type theory. It is capable of handling dependently-typed and/or universe-polymorphic input terms.

## What is grind?

New proof automation (Lean v4.22 – released mid August) developed by Kim Morrison and myself.

Kim is a kick-ass mathematician.

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

- Discovers new equalities, inequalities, etc.
- Writes facts on the board and merges equivalent terms
- Multiple engines cooperate on the same workspace

Cooperating Engines:

- Congruence closure; E-matching; Constraint propagation; Guided case analysis
- Satellite theory solvers (linear integer arithmetic, commutative rings, linear arithmetic)

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

Produces ordinary Lean proof terms for every fact.

# What grind is NOT

**Not designed for combinatorially explosive search spaces:**

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

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

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

For massive case-analysis, use bv_decide

# grind: Design Principles

Native to **Dependent Type Theory**: No translation to first-order or higher-order logic needed.

**Solves trivial goals** automatically.

**Fast startup time**: No server startup, no external tool dependencies, no translations

Great for software verification applications.

No Mathlib dependency.

Rich **diagnostics**: When it fails, it tells you why.

**Configurable** via Type Classes.

**Extensible**: users can plugin their own theory solvers and constraint propa

Provide **grind?** similarly to bv_decide? and aesop?

Stdlib and Mathlib pre-annotated.

# grind: Architecture

**Preprocessing**: normalization, canonicalization, extracting nested proofs, hash-consing, …

**Internalization**: process of converting Lean expressions into solver's internal data-structures.

**E-graph**: congruence closure, E-matching, constraint propagation.

**Satellite Solvers**: cutsat, commutative rings, linear arithmetic, AC, etc.

# grind: Model-based theory solvers

For linear arithmetic (linarith) and linear integer arithmetic (cutsat).

linarith is parametrized by a Module over the integers. It supports preorders, partial orders, and linear orders.

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

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

OrderedVectorSpace implements IntModule, LinearOrder, IntModule.IsOrdered.

## grind: Model-based theory solvers

cutsat is parametrized by the ToInt type class used to embed types such as Int32, BitVec 64 into the integers.

```
/--
The embedding into the integers takes addition to addition, wrapped into the range interval.
-/
class ToInt.Add (α : Type u) [Add α] (I : outParam IntInterval) [ToInt α I] where
  /-- The embedding takes addition to addition, wrapped into the range interval. -/
  toInt_add : ∀ x y : α, toInt (x + y) = I.wrap (toInt x + toInt y)

/--
The embedding into the integers is monotone.
-/
class ToInt.LE (α : Type u) [LE α] (I : outParam IntInterval) [ToInt α I] where
  /-- The embedding is monotone with respect to `≤`. -/
  le_iff : ∀ x y : α, x ≤ y ↔ toInt x ≤ toInt y
```

# grind: Model-based theory solvers

```
example (x y : Int) :
    27 ≤ 11*x + 13*y → 11*x + 13*y ≤ 45 →
    -10 ≤ 7*x - 9*y → 7*x - 9*y ≤ 4 → False := by
  grind

example (a b c : UInt32) :
    -a + -c > 1 →
    a + 2*b = 0 →
    -c + 2*b = 0 → False := by
  grind

example (a : Fin 4) : 1 < a → a ≠ 2 → a ≠ 3 → False := by grind
```

# grind: Commutative rings and Fields

Support for commutative rings and fields uses Grobner basis.

Parametrized by the type classes: CommRing, CommSemiring, NoNatZeroDivisors, Field, AddRightCancel, and IsCharP

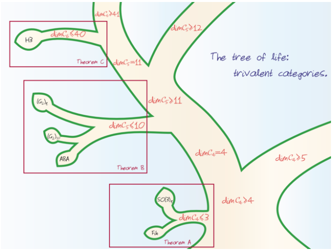
```
example {α} [CommRing α] (a b c : α)
  : a + b + c = 3 →
    a^2 + b^2 + c^2 = 5 →
    a^3 + b^3 + c^3 = 7 →
    a^4 + b^4 + c^4 = 9 := by
  grind

example [Field R] (a : R) : (2 * a)⁻¹ = a⁻¹ / 2 := by grind

example [Field R] (a : R) : (2 : R) ≠ 0 → 1 / a + 1 / (2 * a) = 3 / (2 * a) := by grind

example [Field R] [IsCharP R 0] (a : R) : 1 / a + 1 / (2 * a) = 3 / (2 * a) := by grind

example (x y : BitVec 16) : x^2*y = 1 → x*y^2 = y → y*x = 1 := by grind
```

# Automating Quantum Algebra

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

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

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



# Automating Quantum Algebra

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

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

**grind can handle this kind of reasoning automatically, in milliseconds.**

# Associative (commutative, idempotent) operators & neutral elements

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

Parametrized by the type classes: Associative, Commutative, IdempotentOp, LawfulIdentity.

Long-term: AC E-matching.

```
example {α} (f : α → α) (op : α → α → α) [Associative op] [Commutative op] (a b : α)
    : op (f (op a b)) b = op b (f (op b a)) := by
  grind only

example {α} (bar : α → α) (op : α → α → α) [Associative op] [IdempotentOp op]
    (a b c d e f x y w : α)
    : op d (op x c) = op a b →
      op e (op f (op y w)) = op (op d a) (op b c) →
      bar (op d (op x c)) = bar (op e (op f (op y w))) := by
  grind only

example (a b c : Nat) : min a (max b c) = min (max c b) a := by
  grind -cutsat only

example (a b c : Nat) : min a (max b (max c 0)) = min (max c b) a := by
  grind -cutsat only
```

# grind: E-matching

E-matching is a heuristic for instantiating theorems. It is used in many SMT solvers.

It is matching modulo equalities.

```
@[grind =] theorem fg {x} : f (g x) = x := by
  unfold f g; omega

example {a b c} : f a = b → a = g c → b = c := by
  grind
```

```
-- Whenever `grind` sees `cos` or `sin`, it adds `(cos x)^2 + (sin x)^2 = 1` to the whiteboard.
-- That's a polynomial, so it is sent to the Grobner basis module.
-- And we can prove equalities modulo that relation!
example {x} : (cos x + sin x)^2 = 2 * cos x * sin x + 1 := by
  grind
```

# grind: E-matching and Dependent Type Theory

```
def pbind {α β} : (o : Option α) → (f : (a : α) → o = some a → β) → Option β
  | none, _ => none
  | some a, f => some (f a rfl)

theorem pbind_some {α o β} {f : (a : α) → some o = some a → β} : pbind (some o) f = some (f o rfl) :=
  rfl

example {b} (x : Option Nat) (h : x = some b) : pbind x (fun a h => a + 1) = some (b + 1) := by
  /-
  E-matching instantiates:
  pbind_some: pbind (some b) (cast ⋯ fun a h => a + 1)
  = some (cast ⋯ (fun a h => a + 1) b ⋯)
  -/
  grind [pbind_some] -- fails

@[grind gen]
theorem pbind_some' {x a f} (h : x = some a): pbind x f = some (f a h) := by
  subst h; rfl

example {a} (x : Option Nat) (h : x = some a) : pbind x (fun x _ => x + 1) = some (a + 1) := by
  grind -- success
```

# grind: Extensibility

You can configure grind using type classes.

You can annotate theorems and definitions with the `[grind]` attributes.

```
@[grind =]
theorem getElem?_cons {a l i} : (a :: l)[i]? = if i = 0 then some a else l[i-1]? := by
  cases i <;> simp
```

```
@[grind →]
theorem getElem_of_getElem? {α i a} {l : List α} : l[i]? = some a → ∃ h : i < l.length, l[i] = a :=
  getElem?_eq_some_iff.mp
```

That said, grind is implemented in Lean, and you can extend its implementation using Lean itself.

No need to learn another programming language, or how to create shared objects.

# grind: Extensibility - propagators

```
/--
Propagates equalities for a disjunction `a ∨ b` based on the truth values
of its components `a` and `b`. This function checks the truth value of `a` and `b`,
and propagates the following equalities:

- If `a = False`, propagates `(a ∨ b) = b`.
- If `b = False`, propagates `(a ∨ b) = a`.
- If `a = True`, propagates `(a ∨ b) = True`.
- If `b = True`, propagates `(a ∨ b) = True`.
-/
builtin_grind_propagator propagateOrUp ↑Or := fun e => do
  let_expr Or a b := e | return ()
  if (← isEqFalse a) then
    -- a = False → (a ∨ b) = b
    pushEq e b <| mkApp3 (mkConst ``Grind.or_eq_of_eq_false_left) a b (← mkEqFalseProof a)
  else if (← isEqFalse b) then
    -- b = False → (a ∨ b) = a
    pushEq e a <| mkApp3 (mkConst ``Grind.or_eq_of_eq_false_right) a b (← mkEqFalseProof b)
  else if (← isEqTrue a) then
    -- a = True → (a ∨ b) = True
    pushEqTrue e <| mkApp3 (mkConst ``Grind.or_eq_of_eq_true_left) a b (← mkEqTrueProof a)
  else if (← isEqTrue b) then
    -- b = True → (a ∧ b) = True
    pushEqTrue e <| mkApp3 (mkConst ``Grind.or_eq_of_eq_true_right) a b (← mkEqTrueProof b)
```

```
Lean.Grind.or_eq_of_eq_false_left {a b : Prop} (h : a = False) : (a ∨ b) = b
```

*import Init.Grind.Lemmas*

# grind: Extensibility - solvers

You can plugin your own solver. We implemented all built-in solvers using the plugin API.

```
/-- State for all associative operators detected by `grind`. -/
structure State where
  /--
  Structures/operators detected.
  We expect to find a small number of associative operators in a given goal.
  Thus, using `Array` is fine here.
  -/
  structs : Array Struct := {}
  /--
  Mapping from operators to its "operator id". We cache failures using `none`.
  `opIdOf[op]` is `some id`, then `id < structs.size`. -/
  opIdOf : PHashMap ExprPtr (Option Nat) := {}
  /--
  Mapping from expressions/terms to their structure ids.
  Recall that term may be the argument of different operators. -/
  exprToOpIds : PHashMap ExprPtr (List Nat) := {}
  steps := 0
  deriving Inhabited

builtin_initialize acExt : SolverExtension State ← registerSolverExtension (return {})
```

# grind: Extensibility - solvers

After you declare your solver extension. You implement your internalizer, propagators, and equality handlers.

```
def processNewDiseq (a b : Expr) : GoalM Unit := withExprs a b do
  let ea ← asACExpr a
  let lhs ← norm ea
  let eb ← asACExpr b
  let rhs ← norm eb
  { lhs, rhs, h := .core a b ea eb : DiseqCnstr }.assert
```

```
builtin_initialize
  acExt.setMethods
    (internalize := AC.internalize)
    (newEq := AC.processNewEq)
    (newDiseq := AC.processNewDiseq)
    (check := AC.check)
    (checkInv := AC.checkInvariants)
```

# grind: Tooling

How to maintain annotations in a huge libraries with more than 2M lines of code?

```
/-- Analyzes all theorems in the standard library marked as E-matching theorems. -/
def analyzeEMatchTheorems (c : Config := {}) : MetaM Unit := do
  let origins := (← getEMatchTheorems).getOrigins
  let decls := origins.filterMap fun | .decl declName => some declName | _ => none
  for declName in decls.mergeSort Name.lt do
    try
      analyzeEMatchTheorem declName c
    catch e =>
      logError m!"{declName} failed with {e.toMessageData}"
  logInfo m!"Finished analyzing {decls.length} theorems"

/-- Macro for analyzing E-match theorems with unlimited heartbeats -/
macro "#analyzeEMatchTheorems" : command => `(
  set_option maxHeartbeats 0 in
  run_meta analyzeEMatchTheorems
)

#analyzeEMatchTheorems

-- -- We can analyze specific theorems using commands such as
set_option trace.grind.ematch.instance true

-- 1. grind immediately sees `(#[] : Array α) = ([] : List α).toArray` but probably this should be hidden.
-- 2. `Vector.toArray_empty` keys on `Array.mk []` rather than `#v[].toArray`
-- I guess we could add `(#[].extract _ _).extract _ _` as a stop pattern.
run_meta analyzeEMatchTheorem ``Array.extract_empty {}
```

# **grind: Diagnostics** at your fingertips

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

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

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

# **grind: Diagnostics** at your fingertips

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

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

[grind] Issues ▶

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

# **grind: Diagnostics** at your fingertips

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

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

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

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

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

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

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

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

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

Interactive tactic suggestion tool: the `try?` tactic

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

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

Lean Infoview

▼Suggestions

```
Try these:

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

# grind: Initial reactions

**Markus de Medeiros** Jul 22nd at 1:43 PM

I keep being surprised by how many nuisance goals `grind` is able to solve. Props to everyone who worked on it!

🎉 7 +1 1

**You and Kim Morrison, Oliver Nash** AUG 8

**Oliver Nash** EDITED 8:27 AM

I was just singing `grind`s praises in the Mathlib community meeting and highlighted my favourite example was #27372 (which massively golfs some of my work).

After the call somebody suggested I highlight it to you both for your enjoyment :)

❤️ You

SEP 7

**Fabrizio Montesi** 11:28 AM

Testing a bundled definition of `Bisimulation`, and holy cow does `grind` shine. With the right annotations, it managed to prove that bisimilarity is a bisimulation.

```
def Bisimilarity (lts : Lts State Label) : Bisimulation lt:
  rel s1 s2 := ∃ r : Bisimulation lts, r s1 s2
  is_bisimulation := by grind
```

Chris Henson, Shreyas Srinivas, Kim Morrison

```
-  refine ⟨_, _, _, _, ha, haj, hb, hbj, hc, hcj, hd, hdj, ?_, ?_, ?_, ?_, ?_⟩
-    <;> rw [mem_insert] at * <;> try rintro rfl
-  · obtain (rfl | ha) := ha
-    · obtain (rfl | hb) := hb
-      · exact hw.isPathGraph3Compl.fst_ne_snd rfl
-      · exact hw.fst_notMem_right hb
-    · obtain (rfl | hb) := hb
-      · exact hw.snd_notMem_left ha
-      · exact haj <| hW <| mem_inter_of_mem ha hb
-  · obtain (rfl | ha) := ha
-    · obtain (rfl | hd) := hd
-      · exact hw.isPathGraph3Compl.ne_fst rfl
-      · exact hw.fst_notMem_right hd
-    · obtain (rfl | hd) := hd
-      · exact hw.notMem_left ha
-      · exact haj <| hW <| mem_inter_of_mem ha hd
-  · obtain (rfl | hb) := hb
-    · obtain (rfl | hc) := hc
-      · exact hw.isPathGraph3Compl.ne_snd rfl
-      · exact hw.snd_notMem_left hc
-    · obtain (rfl | hc) := hc
-      · exact hw.notMem_right hb
-      · exact hbj <| hW <| mem_inter_of_mem hc hb
-  · intro hat
-    obtain (rfl | ha) := ha
-    · exact hw.fst_notMem_right hat
-    · exact haj <| hW <| mem_inter_of_mem ha hat
-  · intro hbs
-    obtain (rfl | hb) := hb
-    · exact hw.snd_notMem_left hbs
-    · exact hbj <| hW <| mem_inter_of_mem hbs hb
+  exact ⟨_, _, _, _, ha, haj, hb, hbj, hc, hcj, hd, hdj, by grind⟩
```

Comment on line R312

**YaelDillies** 19 minutes ago · Collaborator

Wow! 😍

# grind: Initial reactions

Terence Tao
@tao@mathstodon.xyz

In contrast, AI chatbots are usually tuned to avoid a "failure mode" as much as possible, at the expense of increasing the occurrence of "intermediate modes" where the chatbot response looks potentially useful, and invites further interaction from the user, but is not exactly providing what the user wants, and could contain hallucinations or some fundamental misunderstanding of the task that would take significant effort to uncover. Paradoxically, such tools may become significantly more useful if they simply reported that they were unable to provide a high quality answer to a query in such cases.

A comparison may be drawn with the increasingly advanced, but stringently verified, "tactics" used in a modern proof assistant such as Lean. I have been experimenting recently with the new tactic `grind` in Lean, which is a powerful tool (inspired more by "good old-fashioned AI" such as satisfiability modulo theories (SMT) solvers, than modern data-driven AI) to try to close complex proof goals if all the tools needed to do so are already provided in the proof environment; roughly speaking, this corresponds to proofs that can be obtained by "expanding everything out and trying all obvious combinations of the hypotheses". When I apply `grind` to a given subgoal, it can report a success within seconds, closing that subgoal in a Lean-verified fashion and allowing me to move on to the next subgoal. But, importantly, when this does not work, I quickly get a "`grind` failed" message, in which case I simply delete `grind` from the code and proceed by a more pedestrian sequence of lower level tactics. (2/3)

## grind: Roadmap

AC E-matching.

More tooling: grind parameter minimizer, deploy annotation analyzers.

Make `try?` as a hub for all proof automation in Lean, AI-based ones included.

Mathlib annotations: crowd sourcing.

Nonlinear inequality support. Interval propagator.

```
theorem historicalVaR_monotonic (ar : AssetReturns) (c₁ c₂ : ConfidenceLevel) (v₁ v₂ : Int)
    : c₁ ≤ c₂ → historicalVaR ar c₁ = some v₁ → historicalVaR ar c₂ = some v₂ → v₂ ≤ v₁ := by
  fun_cases historicalVaR ar c₁ <;> fun_cases historicalVaR ar c₂ <;> simp
  next sorted₁ n₁ p₁ i₁ _ _ sorted₂ n₂ p₂ i₂ _ =>
  intros
  have : p₂ * n₁ ≤ p₁ * n₂ := by apply Nat.mul_le_mul_right <;> grind
  grind
```

## grind: Roadmap

Isn't this example a good candidate for AI?

```
theorem historicalVaR_monotonic (ar : AssetReturns) (c₁ c₂ : ConfidenceLevel) (v₁ v₂ : Int)
    : c₁ ≤ c₂ → historicalVaR ar c₁ = some v₁ → historicalVaR ar c₂ = some v₂ → v₂ ≤ v₁ := by
  fun_cases historicalVaR ar c₁ <;> fun_cases historicalVaR ar c₂ <;> simp
  next sorted₁ n₁ p₁ i₁ _ _ sorted₂ n₂ p₂ i₂ _ =>
  intros
  have : p₂ * n₁ ≤ p₁ * n₂ := by apply Nat.mul_le_mul_right <;> grind
  grind
```

Yes, AI can figure out the missing steps in this example, and it even feels like a good candidate for AI since *nonlinear integer arithmetic* is *undecidable*, but a **simple heuristic** is **cheaper** and more effective.

# grind: Roadmap – New tooling for Maintenance

**The challenge**: maintaining a 2M LoC library with grind annotations (aka hints)

**Library build vs. use time**: They often require different annotations. We use `[local grind]`

**New tools for suggesting and maintaining annotations.**

**User request**: An un-grind tool that expands a grind proof into a detailed tactic proof.

**For our AI experts**: AI agents that can use all these tooling, create PRs with improvements, suggest new annotations.

# Conclusion

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

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

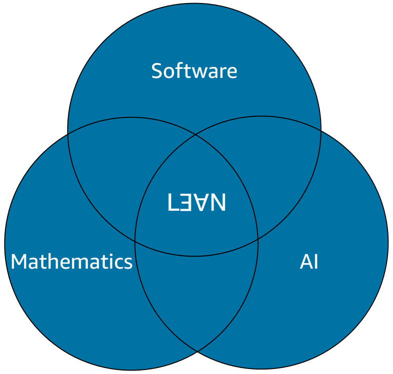
**grind is a new extensible tactic based on SMT techniques.**

`try?` will be a hub for all proof automation in Lean, AI-based ones included.

Many new extensions and features are in development.

We expect AI systems will adopt grind as key tactic within months.

**Maintaining formal proofs is as hard as writing them in the first place.**



# Thank You

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

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