Maggie Simmons
m8simmon@uwaterloo.ca

Cheriton School of Computer Science
University of Waterloo



An Introduction to Lean 4:
Proofs, Programs, and Verified Compilers

based on work by Leonardo de Moura, Sebastian Ullrich, Mario Carneiro, Leo Okawa Ericson, and others

Outline

1. Motivation
2. Lean Basics
3. System Description
4. Trusting Lean
5. Lean Ecosystem
6. Verified Compilation
7. Conclusion

Why Formal Verification?

Testing can show the presence of bugs, but never their absence.

- **Csmith** [14]: three years of randomized differential testing of C compilers.
  - More than 325 previously unknown bugs reported to GCC, LLVM, and others.
  - Includes both compiler crashes and generating wrong code on valid input.
- **CompCert** [8]: a C compiler with a provably correct optimizer.

Formal Verification Overview

1. Write a mathematical specification of what the system should do.
2. Construct a proof that the implementation meets the specification.
3. Have a machine check the proof.*

Two Families of Tools

Automatic Theorem Provers (ATPs)

SAT/SMT solvers: Z3, cvc5, Alt-Ergo, ...

- Push-button: decide whether a formula in a fixed logic is satisfiable.
- Fast on large numbers of small problems.
- Low barrier to entry.
- **Tradeoff:** little room for guidance, brittle to encoding, hard to verify by hand.

Interactive Theorem Provers (ITPs)

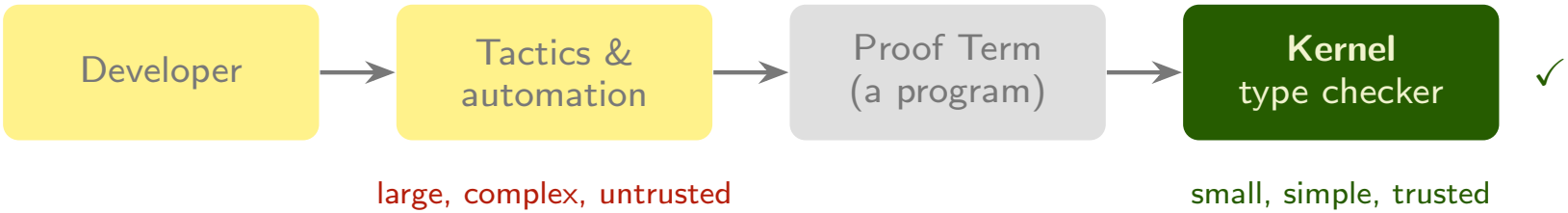
Proof assistants: Lean, Rocq (Coq), Isabelle, ...

- Human-written proof, checked by **kernel**.
- Expressive logic: (almost) all of mathematics.
- Small trusted codebase $\Rightarrow$ very high confidence.
- **Tradeoff:** much more expertise and effort; proving can be tedious.

Which Tool to Use? [2]

ITPs suit verifying one system in depth; ATPs suit solving many instances of a problem class. Increasingly, the two approaches are combined.

What Is a Proof Assistant?



- Proofs are data.
- Automation may be arbitrarily complex, or even wrong: bad proofs are *rejected*.
- **de Bruijn Criterion**: Only the kernel* must be trusted.

A Question to Think About

What exactly do I have to trust when Lean says "no errors"?



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**

A Brief History of Lean

- Started in 2013 by Leonardo de Moura at Microsoft Research [5].
- **Lean 4** is a reimplementation of Lean **in Lean** [5]*.
- Logic: a dependent type theory in the family of the *Calculus of Inductive Constructions* [5].
- Now developed by the non-profit **Lean FRO**, cofounded by de Moura [4].

A short timeline [1]

- **2013** Project starts (MSR)
- **2014** Lean 0.1
- **2015** Lean 2
- **2017** Lean 3 + separate mathlib
- **2021** Lean 4 first released
- **2023** Lean 4 official release

Propositions as Types (Curry–Howard)

Logic	Lean type
$P \Rightarrow Q$	function `P → Q`
$P \wedge Q$	pair `P ∧ Q`
$P \vee Q$	tagged union `P ∨ Q`
$\forall x.\, P(x)$	dependent function
$\exists x.\, P(x)$	dependent pair
$\bot$	empty type `False`
proof of P	term of type P
checking a proof	type checking

```
-- A proof is a program:
-- take a pair, return the swapped pair
theorem and_swap (p q : Prop) :
    p ∧ q → q ∧ p :=
  fun ⟨hp, hq⟩ => ⟨hq, hp⟩
```

- A `theorem` is just a definition whose type is a proposition [5].
- If interested, see [11].

Consequence: Functions Must be Total

A non-terminating `def f (x : Nat) : False := f x` would "prove" `False`. Lean therefore checks that recursive definitions terminate (structural or well-founded recursion).

Lean as a Programming Language

```
-- an algebraic data type
inductive BoolExpr where
  | var (name : String)
  | val (b : Bool)
  | or  (p q : BoolExpr)
  | not (p : BoolExpr)

abbrev Context := List (String × Bool)

-- an evaluator, by pattern matching
def denote (ctx : Context) :
    BoolExpr → Bool
  | .or p q => denote ctx p || denote ctx q
  | .not p  => !denote ctx p
  | .val b  => b
  | .var x  => (ctx.lookup x).getD false

#eval denote [("p", true)]
        (.or (.var "p") (.val false))
-- true
```

- ML/Haskell-style: algebraic data types, pattern matching, type inference.
- `inductive` also generates an **induction principle** that the kernel understands.
- `#eval` runs code; `#check` shows a term's type.
- Strict evaluation; compiled to C.

Code Adapted from

de Moura & Ullrich, CADE 2021 [5]. (The paper used an association list for `Context`.)

Dependent Types: Specifications Inside Types

Types may depend on values. This lets a type express a precise property.

An inductive family of data

```
-- vectors indexed by their length
inductive Vec (α : Type) : Nat → Type where
  | nil  : Vec α 0
  | cons : α → Vec α n → Vec α (n + 1)

-- head cannot fail: the type rules out nil
def Vec.head : Vec α (n + 1) → α
  | .cons a _ => a
```

An inductive predicate

```
inductive Even : Nat → Prop where
  | zero   : Even 0
  | addTwo : Even k → Even (k + 2)

-- a proof is a derivation tree
theorem even_four : Even 4 :=
  .addTwo (.addTwo .zero)
```

- `Type` holds data; `Prop` holds propositions (statements to prove) [11].
- An inductive predicate is the *smallest* relation closed under its rules.

Writing Proofs: Computation and Case Analysis

```
def myOr : Bool → Bool → Bool
  | true,  _ => true
  | false, b => b

-- proof by computation
theorem myOr_true (b : Bool) :
    myOr true b = true := rfl

-- proof by pattern matching
theorem myOr_self : ∀ b : Bool, myOr b b = b
  | true  => rfl
  | false => rfl

-- the same proof, using tactics
theorem myOr_self' (b : Bool) :
    myOr b b = b := by
  cases b <;> rfl
```

- `rfl : a = a` proves an equation when both sides **reduce to the same value**: they are *definitionally equal* [5].
- `myOr true b` reduces to `true`, so `rfl` works.
- `myOr b b` is stuck on the unknown `b`: we must split on `b` first.
- `by` switches to **tactic mode**: commands that build the proof term for you.

Tactics and the Goal View

```
-- simplify : BoolExpr → BoolExpr  folds
-- constants bottom-up (helpers mkOr, mkNot)
@[simp] theorem denote_mkOr (ctx : Context)
    (p q : BoolExpr) :
    denote ctx (simplify.mkOr p q)
      = denote ctx (.or p q) := by
  ...  -- proof elided

theorem denote_simplify (ctx : Context)
    (p : BoolExpr) :
    denote ctx (simplify p) = denote ctx p := by
  induction p with
  | or p q ih1 ih2 => simp [ih1, ih2]
  | not p ih       => simp [ih]
  | _              => rfl
```

Abridged from [5].

What you see in the editor (case `or`):

```
ctx :  Context
p q :  BoolExpr
ih1 :  denote ctx (simplify p)
    = denote ctx p
ih2 :  denote ctx (simplify q)
    = denote ctx q
⊢ denote ctx (simplify (p.or q))
    = denote ctx (p.or q)
```

- `induction` generates one goal per constructor, with induction hypotheses.
- `simp` rewrites with all `@[simp]` lemmas plus the ones you pass.
- The editor (via the Language Server Protocol) shows the goal state live [5].

Typeclasses

```
-- (as in the standard library)
class Inhabited (α : Sort u) where
  default : α

instance : Inhabited BoolExpr where
  default := .val false

-- instances can be chained
instance [Inhabited α] [Inhabited β] :
    Inhabited (α × β) where
  default := (default, default)

deriving instance DecidableEq for BoolExpr
#eval decide (BoolExpr.val true = .val false)
-- false
```

- Ad-hoc polymorphism, as in Haskell: `Add`, `Repr`, `DecidableEq`, ...
- Resolution is a small logic-programming search over instances [5].
- Lean 4 uses **tabled resolution** [5].
- Mathlib's algebraic hierarchy (groups, rings, fields, ...) is built on typeclasses.

Extensibility: Syntax Is a Library

```
-- an embedded DSL for BoolExpr
syntax "`[BExpr|" term "]" : term
macro_rules
  | `(`[BExpr| true])   => `(BoolExpr.val true)
  | `(`[BExpr| false])  => `(BoolExpr.val false)
  | `(`[BExpr| $x:ident]) =>
      `(BoolExpr.var $(quote x.getId.toString))
  | `(`[BExpr| $p ∨ $q]) =>
      `(BoolExpr.or `[BExpr| $p] `[BExpr| $q])
  | `(`[BExpr| ¬ $p])   =>
      `(BoolExpr.not `[BExpr| $p])

#check `[BExpr| p ∨ true]
-- BoolExpr.or (BoolExpr.var "p")
--   (BoolExpr.val true) : BoolExpr
```

From [5].

- Users can extend the **parser, elaborator, tactics, decision procedures, pretty printer, and code generator** [5].
- **Hygienic** macros avoid accidental name capture [5].
- Tactics are ordinary Lean programs (*metaprograms*) with access to Lean's internals.
- Makes custom automation (SMT bridges, domain-specific tactics) practical.

an example

```
class Distrib (α : Type u) extends Mul α, Add α where
  left_distrib  : ∀ a b c : α, a*(b+c) = a*b + a*c
  right_distrib : ∀ a b c : α, (a+b)*c = a*c + b*c

class Ring (α : Type u)
  extends AddCommGroup α, Monoid α, Distrib α
```

```
/-- `Matrix m n α` is the type of matrices whose rows
  are indexed by `m` and whose columns are indexed by
  `n`, and the elements have type `α`. -/
def Matrix (m : Type u) (n : Type v) (α : Type w) :
    Type (max u v w) :=
  m → n → α
```

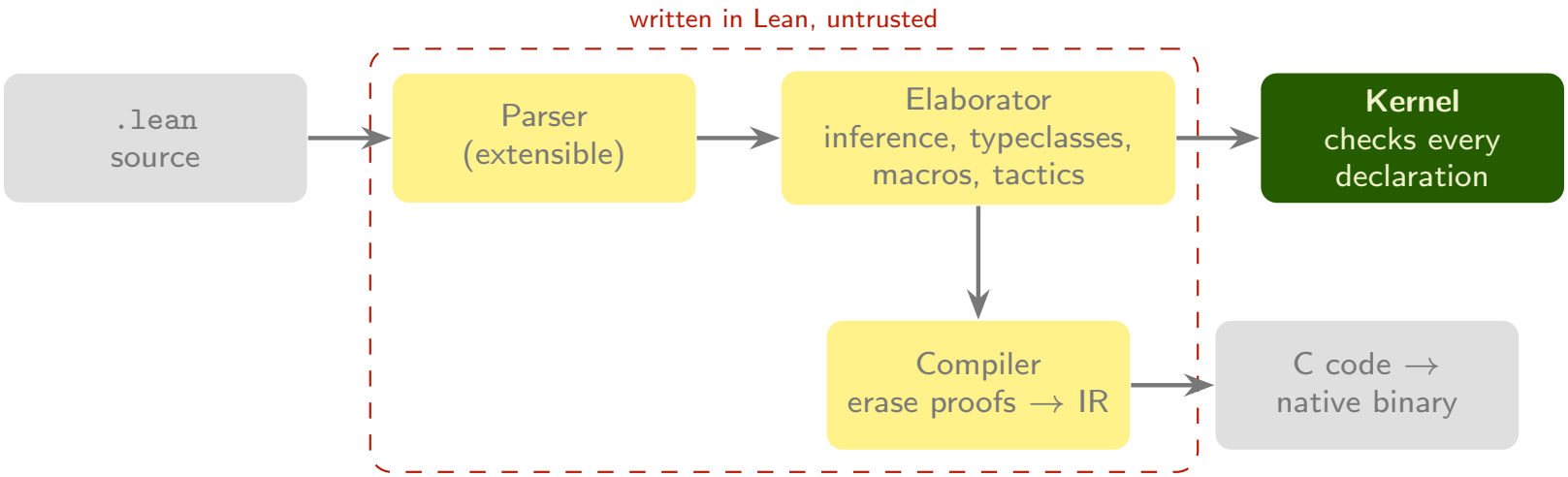
```
/- Scoped notation for accessing values
   stored in matrices. -/
scoped syntax:max term noWs "[" term ", " term "]" : term
macro_rules
  | `($x[$i, $j]) => `($x $i $j)

instance [Add α] : Add (Matrix m n α) where
  add x y i j := x[i, j] + y[i, j]
```

```
instance [Fintype n] [DecidableEq n] [Ring α] :
    Ring (Matrix n n α) :=
  { Matrix.semiring, Matrix.addCommGroup with }
```

- **Hierarchy as typeclasses**: `Ring extends` simpler structures; axioms are fields.
- **A matrix is a function** `m → n → α`.
- **Custom notation** `x[i, j]` via `syntax` + `macro_rules`; `scoped` limits where it applies.
- **Instances compose**: square matrices over a ring form a ring.

Architecture of Lean 4



- Almost everything is written in Lean; the **kernel is still C++** [1].
- The kernel's interface is tiny [1]:

```
addDecl : Environment → Declaration → Except KernelException Environment
```

- So the kernel can be **separated** from the rest of the system, or replaced by an independent re-implementation.
- The elaborator is untrusted by design: soundness must not depend on it refusing a bad term [3].

The Kernel's Type Theory

- A CIC-like dependent type theory [1]:
  - universe hierarchy `Sort u`, with `Prop = Sort 0` and `Type = Sort 1`;
  - `Prop` is **impredicative**;
  - **definitional proof irrelevance**: any two proofs of the same proposition are equal;
  - inductive types, universe polymorphism, quotient types.
- Three standard axioms: `propext`, `Quot.sound`, `Classical.choice`. `#print axioms thm` lists the ones a theorem uses.

Two judgments [1]

- typing: $\Gamma \vdash e : \alpha$
- definitional equality: $\Gamma \vdash e \equiv e'$

Conversion Rule:

$$\frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash \alpha \equiv \beta}{\Gamma \vdash e : \beta}$$

Example: β-reduction

$$\frac{\Gamma, x{:}\alpha \vdash e : \beta \qquad \Gamma \vdash e' : \alpha}{\Gamma \vdash (\lambda x{:}\alpha.\, e)\, e' \equiv e[e'/x]}$$

This is why `rfl` could prove `myOr true b = true`.

Code Generation

- Pipeline: erase proofs and types → Lean IR → C. Many optimizations follow GHC; inlining and specialization are controlled with `@[inline]` and `@[specialize]` [5].
- Memory is managed by reference counting [5].
- **Resurrection hypothesis:** objects often die just before an object of the same kind is created. If a value is *not shared* (refcount 1), Lean **updates it in place**.

```
def map (f : α → β) : List α → List β
  | []      => []
  | a :: as => f a :: map f as
```

If `as` is unshared, `map` allocates no memory: each cons cell is reused [5].

Why does this matter?

Tactics and decision procedures are Lean programs. Pure code can use arrays and hash tables without a performance penalty, and preliminary results were competitive with `ocamlopt` and GHC [5].

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 Haskell, Scala, Rust, etc.
- You can **implement your own checker**.

What Do You Actually Trust?

Trusted computing base (TCB)

- **The statement.** Did you formalize the right theorem?
- **The kernel.** About 6.5k lines of C++, vs about 20k lines of OCaml or 10k of C for Coq's kernel [1].
- **The runtime and build** (GMP, reference counting), whenever checking runs on them [4].
- **The axioms** a proof uses (`#print axioms`). `native_decide` also trusts the compiler.

Not trusted

- The parser, elaborator, macros
- Tactics and all automation
- SMT solvers (if proofs are reconstructed)
- **AI-generated proofs**: treat them as potentially *adversarial* [4]

Three ways to gain confidence in a checker [1]

(1) Testing, (2) independent re-implementation, (3) formal verification.

Is the Theory Itself Sound?

Carneiro, The Type Theory of Lean (2019) [1]

- A complete written specification of Lean's axiomatic system.
- **Consistency** relative to ZFC + $\{$there are n inaccessible cardinals $\mid n < \omega\}$, which is optimal.

But the theory is less well-behaved than one might hope [1]

- Definitional equality is undecidable.
- The kernel's algorithmic equality is not transitive: there are $e_1 \Leftrightarrow e_2$ and $e_2 \Leftrightarrow e_3$ with $e_1 \not\Leftrightarrow e_3$.
- Subject reduction fails for the algorithmic typing judgment.
- The kernel is a *decidable under-approximation* of the ideal typing judgment.

Open Issues

A flaw was found in the unique-typing proof (now a conjecture), and Lean 4's kernel extensions (e.g., nested inductives, η for structures) fall outside the 2019 proof.

Case Study: The 2026 Kernel Soundness Bugs

- **July 25, 2026:** an AI-assisted, `sorry`-free "disproof" of the Collatz conjecture was published then later reduced to a small proof of `False` [3].
  - Cause: the kernel's handling of nested inductive types. Fixed.
  - The exploit also slipped past an outdated copy of **nanoda**.
- **July 30 – August 20:** OpenAI's internal models searched for more bugs, producing 12 PRs [4]:
  - **5 kernel soundness bugs**: e.g., `is_def_eq` caching order, `is_prop` checks;
  - **2 runtime soundness bugs**: reference-count overflow, and a CI build using GMP 6.1.2, which has a known exploitable bug;
  - 5 hardening or non-soundness fixes. All fixed in Lean v4.33.1.

Lessons

"The soundness bugs in this area were implementation mistakes, not gaps in the theory." A small kernel is *not* the same as a bug-free kernel, and most exploits needed metaprogramming to submit terms directly to the kernel.

Defense in Depth: Independent Checkers

Export the proofs, then re-check them [4]

- **lean4checker**: replays declarations in the kernel.
- **nanoda**: an independent checker in Rust.
- **lean4lean**: a kernel written in Lean, aiming at a verified kernel [1].
- **ind-models**: models inductive types from first principles.
- The **Lean Kernel Arena** runs regression tests against all of them. Planned `lake check` (v4.35) will export and re-check a project.

lean4lean performance [1]

	C++ kernel	lean4lean	slower
Lean	37.0 s	44.6 s	21%
Batteries	32.5 s	45.7 s	41%
Mathlib	44.5 min	58.8 min	32%

Independence matters! In the previous slide's bugs, nanoda and ind-models rejected most exploits. Checkers built on the Lean runtime still inherit runtime bugs [4].

Enabling 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.

Mathlib and Formal Mathematics

Mathlib [5]: one unified, community-maintained library of formalized mathematics.

- About **140,000 definitions** and **291,000 theorems** from **770+ contributors** [7].
- Also full of metaprograms: a ring solver, a Presburger arithmetic decision procedure, linters, . . . [5]

Landmark projects

- **Liquid Tensor Experiment**: a challenge posed by Peter Scholze, completed 2022 [5].
- **Empty Hexagon Number**: geometry reduced to SAT, with the reduction verified in Lean [2].

Why should CS researchers care?

The same machinery (inductive types, tactics, automation) is used to verify software.

Lean for Verified Software

Cedar (AWS) [13]

- Open-source authorization policy language.
- Verification-guided development: an executable Lean model sits next to the production Rust code.
- Properties are proved about the model; millions of differential random tests check that Rust and Lean agree.
- The Lean models are about 10× smaller than the production code.
- No release ships unless the model, proofs, and tests are up to date.

Aeneas (Rust → Lean) [6]

- Translates safe Rust into a pure functional model in Lean, using Rust's ownership discipline to remove memory reasoning.
- Proofs are about pure functions, kept separate from the code.
- Used in Microsoft's port of **SymCrypt** from C to verified Rust.
- Uses Lean's extensibility for custom tactics about the generated code.

Bridging Automatic and Interactive Proving

Four ways ATPs and ITPs cooperate [2]

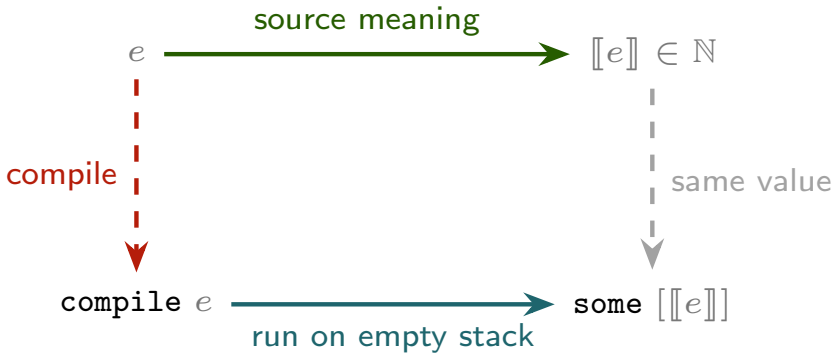
1. ITP verifies an ATP encoding.
2. ATP finds counterexamples to a false goal before you waste time on it.
3. ATP filters premises and finds proof, the ITP reproves.
4. ITP replays ATP certificates by checking the solver's proof.

Lean-SMT (CAV 2025) [9]

- A tactic that translates Lean goals to SMT, calls cvc5, and **reconstructs the SMT proof as a Lean proof**.
- Result: Sledgehammer-style automation with a smaller trusted core and little performance cost.

What Is a Verified Compiler?

Semantic preservation: the compiled program behaves like the source program.



Our case studies

- **A mini-compiler**: arithmetic expressions with `if` → a stack machine. Complete Lean 4 code and proof.
- **Scaling up**: WHILE → a stack machine with jumps, from *Concrete Semantics* [10], ported to Lean by Okawa Ericson [11].
- The industrial version of this idea: **CompCert** [8].

Five steps

(1) syntax → **(2)** source semantics → **(3)** target semantics → **(4)** compiler → **(5)** correctness proof

A Mini-Compiler: Source Language

```
inductive Binop where
| plus : Binop
| times : Binop

inductive Exp where
| const : Nat → Exp
| binop : Binop → Exp → Exp → Exp
| ite   : Exp → Exp → Exp → Exp

-- denotational semantics: Exp ↦ a number
@[simp]
def denoteExp (e : Exp) : Nat :=
  match e with
  | .const n => n
  | .binop b e1 e2 =>
      ⟦ b ⟧ (denoteExp e1) (denoteExp e2)
  | .ite e1 e2 e3 =>
    if (denoteExp e1) ≠ 0 then denoteExp e2
    else denoteExp e3
```

```
-- ⟦ x ⟧ notation via a typeclass
class Semantic (α : Type)
    (β : outParam Type) where
  denote : α → β
notation "⟦" x "⟧" => Semantic.denote x

def testExp2 : Exp :=
  -- (if 1 then 42 else 8) * 5
  .binop .times
    (.ite (.const 1) (.const 42) (.const 8))
    (.const 5)
#eval ⟦ testExp2 ⟧  -- 210
```

- **Denotational semantics**: each expression means a number. A plain recursive function works because expressions always terminate.
- ⟦b⟧ reuses typeclasses and `notation` from Part 1.
- `@[simp]` lets the simplifier unfold the definition in proofs.

An Embedded DSL in a Few Lines

```
declare_syntax_cat arith_expr
syntax num : arith_expr
syntax "(" arith_expr ")" : arith_expr
syntax:50 arith_expr " + " arith_expr : arith_expr
syntax:60 arith_expr " * " arith_expr : arith_expr
syntax "if " arith_expr " then " arith_expr
       " else " arith_expr : arith_expr
syntax "[arith|" arith_expr "]" : term

macro_rules
| `([arith| $n:num ]) => `(Exp.const $n)
| `([arith| ($e) ]) => `([arith| $e ])
| `([arith| $e1 + $e2 ]) =>
    `(Exp.binop Binop.plus [arith| $e1] [arith| $e2])
| `([arith| $e1 * $e2 ]) =>
    `(Exp.binop Binop.times [arith| $e1] [arith| $e2])
| `([arith| if $c then $t else $f ]) =>
    `(Exp.ite [arith| $c] [arith| $t] [arith| $f])
```

```
def dslExp : Exp := [arith| 3 + 4 * 5 ]
#eval ⟦ dslExp ⟧ -- 23

def dslIte : Exp :=
  [arith| if 1 then 42 else 8 * 5 ]
#eval ⟦ dslIte ⟧ -- 42
```

- A new **syntactic category** with precedences (`*` binds tighter than `+`).
- Macros translate the surface syntax into `Exp` terms at elaboration time.
- The DSL adds nothing to the trusted base: it only builds terms, which the kernel still checks.

A Mini-Compiler: The Target Stack Machine

```
inductive Instr where
| iConst : Nat → Instr     -- push a constant
| iBinop : Binop → Instr   -- pop 2, push result
| iIf    : Instr           -- pop 3, push result

abbrev Prog := List Instr
abbrev Stack := List Nat

@[simp]
def denoteInstr (i : Instr) (s : Stack) :
    Option Stack :=
  match i, s with
  | .iConst n, s => some (n :: s)
  | .iBinop b, arg1 :: arg2 :: s' =>
    some ((⟦ b ⟧ arg1 arg2) :: s')
  | .iIf, arg1 :: arg2 :: arg3 :: s' =>
    some ((if arg1 ≠ 0 then arg2 else arg3) :: s')
  | _, _ => none    -- stack underflow

@[simp]
def denoteProg (p : Prog) (s : Stack) :
    Option Stack :=
  match p with
  | [] => some s
  | i :: p' =>
    match ⟦ i ⟧ s with
    | none => none
    | some s' => denoteProg p' s'
```

```
#eval ⟦ [Instr.iConst 3, Instr.iConst 4,
        Instr.iBinop .times] ⟧ []
-- some [12]
#eval ⟦ [Instr.iConst 1, Instr.iConst 42,
        Instr.iIf] ⟧ []
-- none (iIf needs 3 operands)
```

- Programs denote stack transformers: `Stack → Option Stack`.
- Not every program makes sense: too few operands gives `none`.
- Using `Option` keeps the semantics a **total function**. No loops means no need for a relation.

A Mini-Compiler: The Compiler

```
@[simp]
def compile (e : Exp) : Prog :=
  match e with
  | .const n => [Instr.iConst n]
  | .binop b e1 e2 => compile e2 ++ compile e1 ++ [Instr.iBinop b]
  | .ite e1 e2 e3 => compile e3 ++ compile e2 ++ compile e1 ++ [Instr.iIf]
```

Running it on $(3 + 4) \times 5$

```
-- compile testExp1 =
--   [iConst 5, iConst 4, iConst 3,
--    iBinop .plus, iBinop .times]
#eval ⟦ compile testExp1 ⟧ []
-- some [35]
```

- Operands are compiled right to left, so `e1`'s value ends up on **top** of the stack, where `iBinop` expects `arg1`.
- Testing (`#eval`) checks *one* input. Next we prove it for all of them.

A Mini-Compiler: Stating Correctness

What we want

```
theorem compile_correct (e : Exp) :
  ⟦ compile e ⟧ [] = some [ ⟦ e ⟧ ]
```

Problem: induction on `e` gets stuck. In `compile e2 ++ compile e1 ++ [iBinop b]`, the code for `e1` runs on a non-empty stack and is followed by more code, but the IH only talks about the empty stack and no continuation.

Fix: strengthen the statement, for any continuation `p` and any stack `s`:

```
theorem compile_correct_aux (e : Exp) (p : Prog) (s : Stack) :
  ⟦ compile e ++ p ⟧ s = ⟦ p ⟧ (⟦ e ⟧ :: s)
```

Reading the lemma

Running `compile e` and then `p` is the same as running `p` with $[\![e]\!]$ pushed onto the stack. The main theorem is the special case `p = []`, `s = []`.

A Mini-Compiler: The Proof

Explicit version

```
theorem compile_correct_aux (e : Exp)
    (p : Prog) (s : Stack) :
  ⟦ compile e ++ p ⟧ s
    = ⟦ p ⟧ (⟦ e ⟧ :: s) := by
  induction e generalizing p s
  case const n =>
    simp
  case binop b e1 e2 ih1 ih2 =>
    simp [-denoteBinop] at *
    rw [ih2, ih1]
    simp
  case ite e1 e2 e3 ih1 ih2 ih3 =>
    simp [-denoteExp] at *
    rw [ih3, ih2, ih1]
    simp
```

With more automation

```
  induction e generalizing p s <;> simp_all
```

The `binop` case by hand

with C = `compile` and B = `iBinop b`:

$$
\begin{aligned}
& [\![C\, e_2 \mathbin{++} C\, e_1 \mathbin{++} [B] \mathbin{++} p]\!]\, s \\
= {} & [\![C\, e_1 \mathbin{++} [B] \mathbin{++} p]\!]\, ([\![e_2]\!] :: s) && \text{IH}_2 \\
= {} & [\![[B] \mathbin{++} p]\!]\, ([\![e_1]\!] :: [\![e_2]\!] :: s) && \text{IH}_1 \\
= {} & [\![p]\!]\, ([\![b]\!]\, [\![e_1]\!]\, [\![e_2]\!] :: s) && \texttt{simp} \\
= {} & [\![p]\!]\, ([\![\mathit{binop}\ b\ e_1\ e_2]\!] :: s) && \text{def.}
\end{aligned}
$$

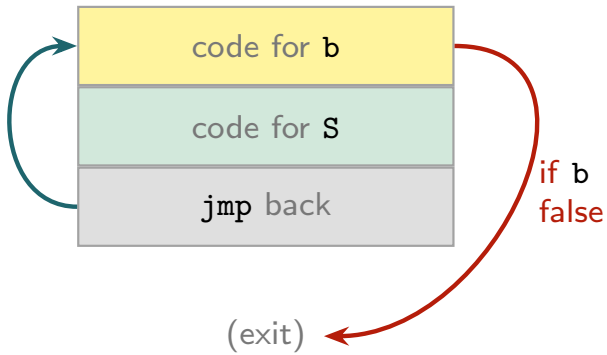
- `generalizing p s` makes the IHs hold for *every* continuation and stack. That is exactly what the `binop` case needs.
- The final theorem: specialize to `p = []`, `s = []`, then `simp`.

From Expressions to a Real Language

Adding variables, control flow, and loops (the WHILE language [10, 11]) changes three things:

1. **Semantics needs state, loops may diverge.** A function `Stmt → State → State` is impossible in Lean (it must be total), so execution becomes an inductive relation `BigStep (c, s) t`.
2. **The machine needs jumps.** Configurations carry a program counter; one-step execution is a relation, closed under reflexivity and transitivity.
3. **The proof follows the derivation**: induction on `BigStep`, plus lemmas for running code inside a larger program (the analogue of our continuation `p`).

Layout of a `while` loop



Full Lean code for this compiler is in the backup slides.

What Did We Actually Prove?

Mini-compiler

- **Full correctness** for *every* expression: the compiled code produces exactly `some [⟦e⟧]`.
- In particular, compiled code never underflows the stack.

Still trusted

- The **semantics**: if `denoteExp` or `denoteInstr` is wrong, the theorem is about the wrong languages.
- The parser.
- Running the compiler: `#eval` uses Lean's code generator, which is not verified.

Takeaway

The theorem is only as good as its statement.

Scaling Up

Two ways to trust a compiler

- **Verified compiler**: prove once, for all inputs (toy example, CompCert [8]).
- **Translation validation** [12]: after each run, check that *this* output matches *this* input. The optimizer stays untrusted, like tactics vs. the kernel.

More automation

- `simp`, `omega`, custom tactics.
- SMT for side conditions with proof reconstruction (Lean-SMT [9]).

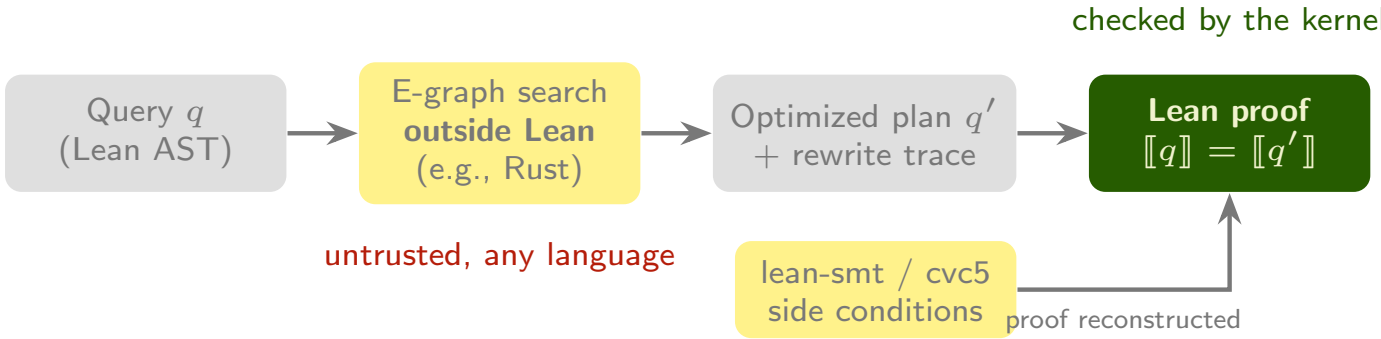
Richer languages

- Optimizations: e.g., constant-fold `Exp` and prove ⟦`fold e`⟧ = ⟦`e`⟧ [10].
- Queries: SQL query equivalence was decided with a tool built on Lean 3 [5].
- Real languages: Rust via Aeneas [6].

Same Process as Shown

Inductive syntax → semantics (functions, or relations when programs can loop) → compiler as a function → proof by induction, after generalizing the statement.

Application: A Translation-Validated Query Optimizer



The plan

- **Translation validation** [12]: don't verify the optimizer; prove each output plan equivalent to its input.
- **Equality saturation** searches many rewrites at once for a cheaper plan. It can run **outside Lean** (e.g., a Rust e-graph library) and hand back the plan and its rewrite trace.
- **SMT-assisted proofs** (lean-smt + cvc5) discharge side conditions.
- **Stretch goal:** update-optimization rewrites.

How it connects to today

- **Mini-compiler recipe**: query syntax (+ DSL), semantics $[\![q]\!]$, proof.
- **De Bruijn criterion**: like a tactic, the e-graph is untrusted, whatever language it is written in; the kernel checks its output.
- **ATP + ITP**: SMT proofs are reconstructed, so cvc5 is untrusted [9].
- **Trust**: only as good as my query semantics.

Conclusion

- Lean 4 is a **proof assistant and a programming language**: programs, specifications, and proofs live in one language.
- Its design rests on a **small trusted kernel**. Everything else, including AI and SMT solvers, can be untrusted.
- Trust is layered: the statement, kernel, runtime, and axioms. The 2026 bugs show why **independent checkers** matter.
- A verified compiler follows a simple recipe: **syntax, semantics, compiler, proof by induction**. The hard part is stating the right theorem.

Questions?

Backup: While Compiler – Syntax

```
-- arithmetic expressions
inductive AExp where
  | num  : Int → AExp
  | var  : String → AExp
  | plus : AExp → AExp → AExp
```

```
-- boolean expressions
inductive BExp where
  | bc   : Bool → BExp
  | not  : BExp → BExp
  | and  : BExp → BExp → BExp
  | less : AExp → AExp → BExp
```

```
-- statements
inductive Stmt where
  | skip       : Stmt
  | assign     : String → AExp → Stmt
  | seq        : Stmt → Stmt → Stmt
  | ifThenElse : BExp → Stmt → Stmt → Stmt
  | whileDo    : BExp → Stmt → Stmt

abbrev State := String → Int
```

- Programs are **abstract syntax trees**: values of an inductive type.
- A `State` maps variable names to integers.
- No parser: we assume one already exists [11].

Backup: While Compiler – Big-Step Semantics

Expressions always terminate
⇒ ordinary recursive functions:

```
def aval : AExp → State → Int
  | .num n,    _ => n
  | .var x,    s => s x
  | .plus a b, s => aval a s + aval b s

-- bval : BExp → State → Bool
-- is defined the same way
```

Statements may loop forever
⇒ a function is impossible (Lean functions must be total), so we use an **inductive relation** [11].

```
-- BigStep (c, s) t :
--   "c, run from s, ends in t"
inductive BigStep :
    Stmt × State → State → Prop
  | skip : BigStep (.skip, s) s
  | assign : BigStep (.assign x a, s)
      (fun y => if y = x then aval a s
                else s y)
  | seq : BigStep (S, s) t →
      BigStep (T, t) u →
      BigStep (.seq S T, s) u
  -- ifTrue / ifFalse are analogous
  | whileTrue : bval b s = true →
      BigStep (S, s) t →
      BigStep (.whileDo b S, t) u →
      BigStep (.whileDo b S, s) u
  | whileFalse : bval b s = false →
      BigStep (.whileDo b S, s) s
```

A non-terminating loop simply has **no** derivation.

Backup: While Compiler – Stack Machine with Jumps

```
inductive Instr where
  | loadi   : Int → Instr
  | load    : String → Instr
  | add     : Instr
  | store   : String → Instr
  | jmp     : Int → Instr
  | jmpless : Int → Instr
  | jmpge   : Int → Instr

-- (program counter, memory, stack)
abbrev Config :=
  Int × State × List Int

-- reflexive-transitive closure
inductive Star (r : α → α → Prop) :
    α → α → Prop
  | refl : Star r a a
  | step : r a b → Star r b c →
      Star r a c

def Exec (P : List Instr) :=
  Star (Exec1 P)
```

```
-- one instruction (selected rules)
inductive IExec : Instr → Config → Config → Prop
  | loadi : IExec (.loadi n) (i, s, stk)
                             (i+1, s, n :: stk)
  | add   : IExec .add (i, s, a :: b :: stk)
                       (i+1, s, (b + a) :: stk)
  | jmp   : IExec (.jmp n) (i, s, stk)
                           (i+1+n, s, stk)
  -- ...

-- fetch the instruction at pc, then execute it
inductive Exec1 (P : List Instr) :
    Config → Config → Prop
  | step : 0 ≤ i → P[i.toNat]? = some ins →
      IExec ins (i, s, stk) c' →
      Exec1 P (i, s, stk) c'
```

- **Small-step** semantics: one instruction at a time; `Exec P` ($\to^*$) runs any number of steps.
- A relation again, since `add` on a too-short stack has no defined result [11].

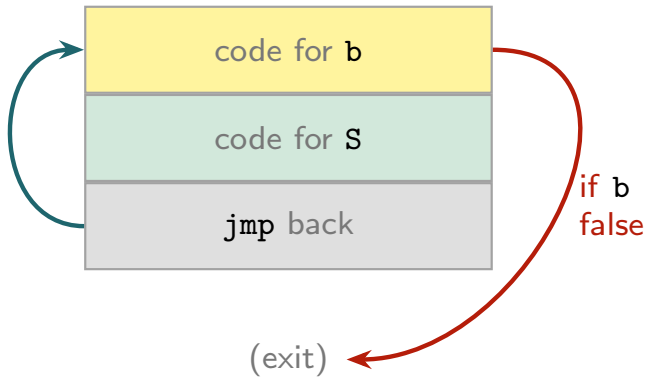
Backup: While Compiler – Compiler

```
def acomp : AExp → List Instr
  | .num n    => [.loadi n]
  | .var x    => [.load x]
  | .plus a b => acomp a ++ acomp b ++ [.add]

-- bcomp b f n : jump forward n iff b = f
-- (definition omitted, see [Nipkow & Klein])

def ccomp : Stmt → List Instr
  | .skip       => []
  | .assign x a => acomp a ++ [.store x]
  | .seq S T    => ccomp S ++ ccomp T
  | .ifThenElse b S T =>
      let cS := ccomp S; let cT := ccomp T
      bcomp b false (cS.length + 1) ++ cS ++
        [.jmp cT.length] ++ cT
  | .whileDo b S =>
      let cS := ccomp S
      let cb := bcomp b false (cS.length + 1)
      cb ++ cS ++
        [.jmp (-(cb.length + cS.length + 1))]
```

Layout of `whileDo b S` [10]



- Jumps are **relative**, so code can be concatenated freely.
- `bcomp` short-circuits `and` and folds constants [10].

Backup: While Compiler – Correctness Statement

```
-- If c runs from s to t, then the compiled code runs from
-- pc 0 to the end of the code, with the same final state t
theorem ccomp_correct :
    BigStep (c, s) t →
    Exec (ccomp c) (0, s, stk) ((ccomp c).length, t, stk)
```

```
-- Lemma for expressions: the value ends up on top of the stack
theorem acomp_correct (a : AExp) :
    Exec (acomp a) (0, s, stk)
         ((acomp a).length, s, aval a s :: stk)
```

- The proof follows the **structure of the data**:
  - `acomp_correct`: induction on the expression `a`;
  - `bcomp_correct`: induction on the boolean expression;
  - `ccomp_correct`: induction on the **derivation** of `BigStep (c, s) t`.
- The stack `stk` is universally quantified (`induction ... generalizing stk`): the inductive hypothesis is used with a *bigger* stack.

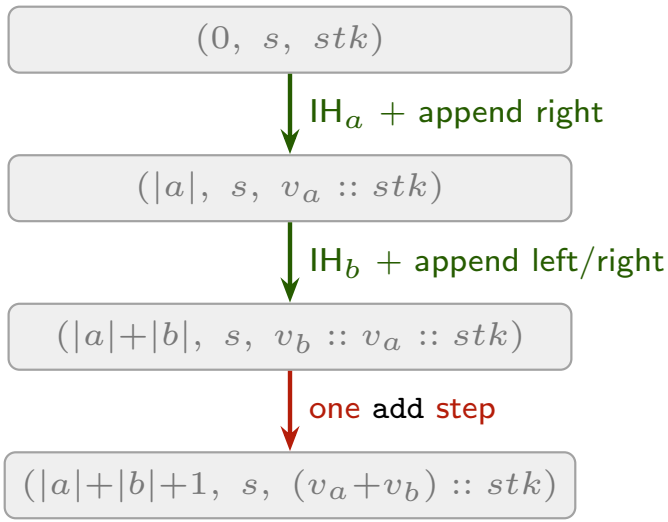
Backup: While Compiler – Proof Idea

Two lemmas about concatenated programs [11]

```
theorem exec_append_right : Exec P c c' → Exec (P ++ Q) c c'
theorem exec_append_left  : Exec P (i, s, stk) (i', s', stk') →
    Exec (Q ++ P) (i + Q.length, s, stk) (i' + Q.length, s', stk')
```

The `plus a b` case

code = `acomp a ++ acomp b ++ [.add]`



- Chain the three runs with **transitivity** of $\to^*$.
- v_a = `aval a s`, v_b = `aval b s`, and $|a|$ = length of `acomp a`.
- The IH for `b` is used with the *larger* stack $v_a :: stk$.
- Most of the effort is bookkeeping on program counters (`Nat`/`Int` casts, list lengths), where automation such as `simp` and `omega` pays off [11].

References I

[1] Rutger Broekhoff. Don't trust, verify: Guarantees for the Lean proof assistant. MFoCS Seminar presentation, Radboud University Nijmegen, January 2025.

[2] Joshua Clune. Connecting automatic and interactive theorem proving. Carnegie Mellon University CSD PhD Blog, May 2025. URL https://www.cs.cmu.edu/~csd-phd-blog/2025/atp-itp-connection/. Accessed 28 October 2026.

[3] Leonardo de Moura. Postmortem for kernel soundness bug #14576. Blog post, August 2026. URL https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/. Accessed 28 October 2026.

[4] Leonardo de Moura. Postmortem for the kernel soundness bug hunt. Blog post, August 2026. URL https://leodemoura.github.io/blog/2026-8-24-postmortem-for-the-kernel-soundness-bug-hunt/. Accessed 28 October 2026.

[5] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language (system description). In Automated Deduction – CADE 28, volume 12699 of Lecture Notes in Computer Science, pages 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

[6] Lean FRO. Aeneas: Verifying Rust programs in Lean. Lean use case, 2026. URL https://lean-lang.org/use-cases/aeneas/. Accessed 28 October 2026.

[7] Lean Prover Community. Mathlib statistics, 2026. URL https://leanprover-community.github.io/mathlib_stats.html. Accessed 28 October 2026.

[8] Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814.

[9] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. Lean-SMT: An SMT tactic for discharging proof goals in Lean. In Computer Aided Verification (CAV 2025), volume 15931 of Lecture Notes in Computer Science, pages 197–212. Springer, 2025. doi: 10.1007/978-3-031-98682-6_11.

[10] Tobias Nipkow and Gerwin Klein. Concrete Semantics: With Isabelle/HOL. Springer, 2014. doi: 10.1007/978-3-319-10542-0.

[11] Leo Okawa Ericson. Simple formally verified compiler in Lean. Bachelor's thesis (examensarbete 15 hp), it 21 097, Department of Information Technology, Uppsala University, November 2021.

[12] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 1384 of Lecture Notes in Computer Science, pages 151–166. Springer, 1998.

[13] Emina Torlak and Kesha Hietala. Lean powers secure software at AWS: Cedar's journey with verified development. Lean use case, 2026. URL https://lean-lang.org/use-cases/cedar/. Accessed 28 October 2026.

[14] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 283–294, 2011. doi: 10.1145/1993498.1993532.