Lean 4 Tutorial

NASA Formal Methods 2022

Leonardo de Moura (MSR) and Sebastian Ullrich (KIT)

Part I: Introduction

Introduction

Lean 4 is a platform for

- Software verification
- Formal mathematics
- Developing custom automation & domain specific languages (DSLs)

Goals

- Extensibility, Expressivity, Scalability, Proof stability
- An efficient functional programming language

Lean is based on dependent type theory

Resources

Website: https://leanprover.github.io/

Theorem Proving in Lean: https://leanprover.github.io/theorem_proving_in_lean4/

Lean 4 Manual (WIP): https://leanprover.github.io/lean4/doc/

Zulip channel: https://leanprover.zulipchat.com/

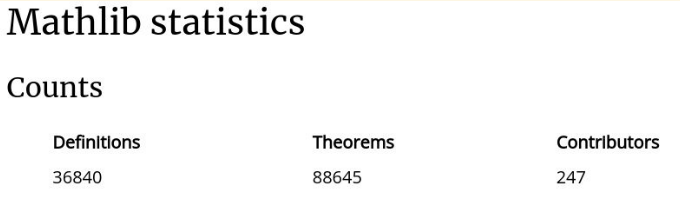
Mathlib 4: https://github.com/leanprover-community/mathlib4

Useful links: https://leanprover.github.io/links/

Community website: https://leanprover-community.github.io/

Mathlib

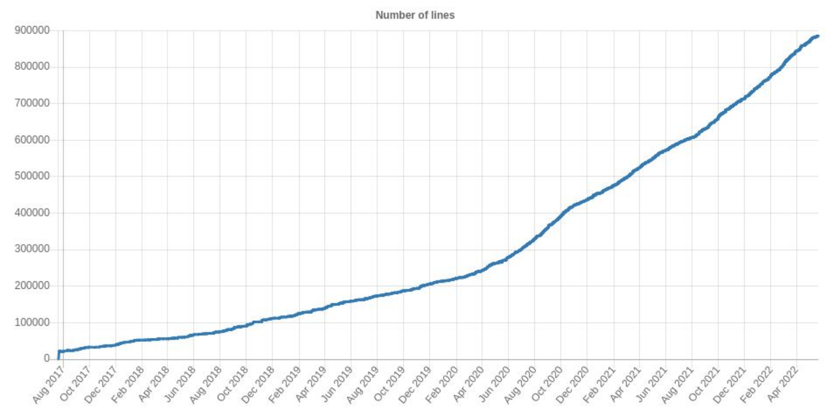
The Lean mathematical library, mathlib, is a community effort to build a unified library of mathematics in Lean.



Mathlib statistics

Counts

Definitions	Theorems	Contributors
36840	88645	247



Number of lines

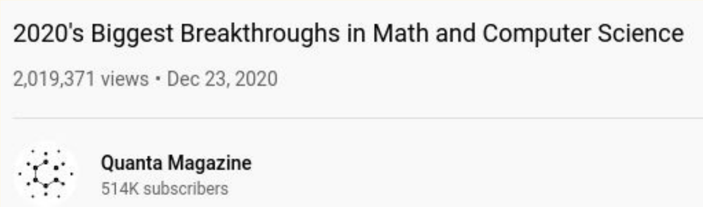
Project momentum

The Effort to Build the Mathematical Library of the Future

A community of mathematicians is using software called Lean to build a new digital repository. They hope it represents where their field is headed next.

NEWS | 18 June 2021

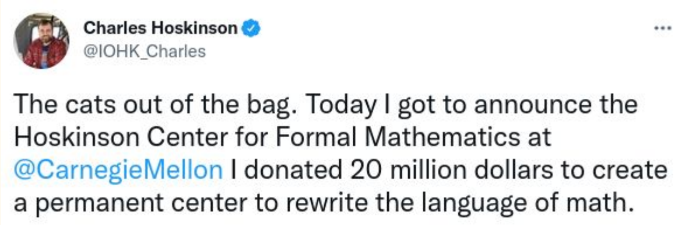
Mathematicians welcome computer-assisted proof in ‘grand unification’ theory



2020's Biggest Breakthroughs in Math and Computer Science

2,019,371 views • Dec 23, 2020

Quanta Magazine
514K subscribers



Charles Hoskinson
@IOHK_Charles

The cats out of the bag. Today I got to announce the Hoskinson Center for Formal Mathematics at @CarnegieMellon I donated 20 million dollars to create a permanent center to rewrite the language of math.

Augmented Mathematical Intelligence (AMI) at Microsoft

Mission

- Empower mathematicians working on cutting-edge mathematics
- Democratize math education
- Platform for Math-AI research

Program manager, engineers, contractors, and academic gifts

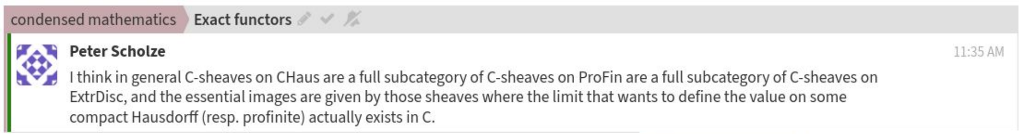
LEAN
THEOREM PROVER
Microsoft Research

Lean Zulip channel

Stanislas Polu 11:50 AM

Hi Everyone. We're tearing down the model that is backing the `gptf` tactic but will work on getting a new model online soon. We'll also work on providing a better experience potentially looking to interface with the VSCode extension more directly. If you have any idea you'd like us to explore, please let us know, the goal is really to provide the community with useful assistance from the models we train. Please let me know if you have questions 👍

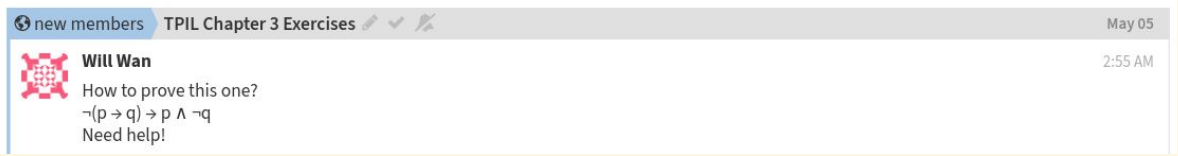
👍 2 ❤ 4



condensed mathematics Exact functors

Peter Scholze 11:35 AM

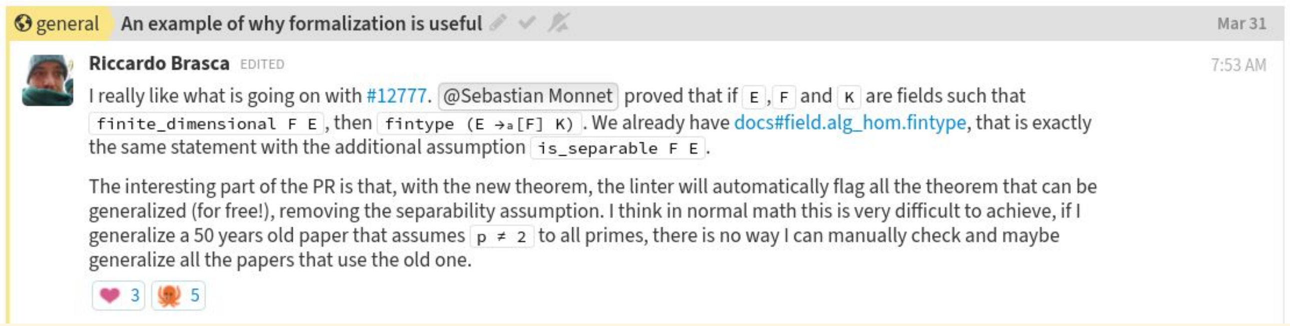
I think in general C-sheaves on CHaus are a full subcategory of C-sheaves on ProFin are a full subcategory of C-sheaves on ExtrDisc, and the essential images are given by those sheaves where the limit that wants to define the value on some compact Hausdorff (resp. profinite) actually exists in C.



new members TPIL Chapter 3 Exercises May 05

Will Wan 2:55 AM

How to prove this one?
¬(p → q) → p ∧ ¬q
Need help!



general An example of why formalization is useful Mar 31

Riccardo Brasca EDITED 7:53 AM

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

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

❤ 3 🐙 5

Lean 4 dev update meetings

New monthly online event

First one will be on June 15th

Details will be posted on our website and twitter https://twitter.com/leanprover

Lean 4 - What is new?

Lean 4 is implemented in Lean

- Extensibility: parser, elaborator, compiler, tactics, formatter, etc
- Hygienic macro system - simple extensions should be simple to implement
- Our LSP (Language Server Protocol) server is great
- Compiler generates efficient C code
- Runtime uses reference counting for GC, and performs destructive updates if RC = 1
- Functional but in place (FBIP)
- Safe support for low-level tricks such as pointer equality
- Tabled type class resolution
- Many scalability and usability improvements

“Hello world”

```
#eval "hello" ++ " " ++ "world"
-- "hello world"

#check true
-- Bool

def x := 10
#eval x + 2
-- 12

def double (x : Int) := 2*x
#eval double 3
-- 6
#check double
-- Int → Int
example : double 4 = 8 := rfl0
```

First-class functions

```
def twice (f : Nat → Nat) (a : Nat) :=
  f (f a)

#check twice
-- (Nat → Nat) → Nat → Nat

#eval twice (fun x => x + 2) 10

theorem twice_add_2 (a : Nat) : twice (fun x => x + 2) a = a + 4 := rfl

-- `(· + 2)` is syntax sugar for `(fun x => x + 2)`.
#eval twice (· + 2) 10
```

Enumerated types

```
inductive Weekday where
  | sunday | monday | tuesday | wednesday
  | thursday | friday | saturday

#check Weekday.sunday
-- Weekday
open Weekday
#check sunday

def natOfWeekday (d : Weekday) : Nat :=
  match d with
  | sunday    => 1
  | monday    => 2
  | tuesday   => 3
  | wednesday => 4
  | thursday  => 5
  | friday    => 6
  | saturday  => 7
```

Enumerated types (cont.)

```
def Weekday.next (d : Weekday) : Weekday :=
  match d with
  | sunday    => monday
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => saturday
  | saturday  => sunday

def Weekday.previous : Weekday → Weekday
  | sunday    => saturday
  …
theorem Weekday.next_previous (d : Weekday) : d.next.previous = d :=
  match d with
  | sunday    => rfl
  | monday    => rfl
  …
  | saturday  => rfl
```

Proving theorems using tactics

```
theorem Weekday.next_previous' (d : Weekday) : d.next.previous = d := by -- switch to tactic mode
  cases d -- Creates 7 goals
  rfl; rfl; rfl; rfl; rfl; rfl; rfl

theorem Weekday.next_previous'' (d : Weekday) : d.next.previous = d := by
  cases d <;> rfl
```

What is the type of Nat?

```
#check 0
-- Nat
#check Nat
-- Type
#check Type
-- Type 1
#check Type 1
-- Type 2
#check Eq.refl 2
-- 2 = 2
#check 2 = 2
-- Prop
#check Prop
-- Type

example : Prop = Sort 0   := rfl
example : Type = Sort 1   := rfl
example : Type 1 = Sort 2 := rfl
```

Implicit arguments and universe polymorphism

```
def f (α β : Sort u) (a : α) (b : β) : α := a
#eval f Nat String 1 "hello"
-- 1

def g {α β : Sort u} (a : α) (b : β) : α := a
#eval g 1 "hello"

def h (a : α) (b : β) : α := a

#check g
-- ?m.1 → ?m.2 → ?m.1
#check @g
-- {α β : Sort u} → α → β → α
#check @h
-- {α : Sort u_1} → {β : Sort u_2} → α → β → α
#check g (α := Nat) (β := String)
```

Inductive Types

```
inductive Tree (β : Type v) where
  | leaf
  | node (left : Tree β) (key : Nat) (value : β) (right : Tree β)
 deriving Repr

#eval Tree.node .leaf 10 true .leaf
-- Tree.node Tree.leaf 10 true Tree.leaf

inductive Vector (α : Type u) : Nat → Type u
  | nil : Vector α 0
  | cons : α → Vector α n → Vector α (n+1)
```

Recursive functions

```
#print Nat -- Nat is an inductive type

def fib (n : Nat) : Nat :=
  match n with
  | 0 => 1
  | 1 => 1
  | n+2 => fib (n+1) + fib n

example : fib 5 = 8 := rfl

example : fib (n+2) = fib (n+1) + fib n := rfl
#print fib
/-
def fib : Nat → Nat :=
fun n =>
 Nat.brecOn n fun n f =>
   (match (motive := (n : Nat) → Nat.below n → Nat) n with
      …
-/
```

Well-founded recursion

```
def ack : Nat → Nat → Nat
  | 0,   y   => y+1
  | x+1, 0   => ack x 1
  | x+1, y+1 => ack x (ack (x+1) y)
termination_by ack x y => (x, y)

def sum (a : Array Int) : Int :=
  let rec go (i : Nat) :=
     if i < a.size then
        a[i] + go (i+1)
     else
        0
  go 0
termination_by go i => a.size - i

set_option pp.proofs true
#print sum.go
/-
def sum.go : Array Int → Nat → Int :=
fun a => WellFounded.fix (sum.go.proof_1 a) fun i a_1 =>
  if h : i < Array.size a then Array.getOp a i + a_1 (i + 1) (sum.go.proof_2 a i h) else 0
-/
```

Mutual recursion

```
inductive Term where
  | const : String → Term
  | app   : String → List Term → Term

namespace Term
mutual
def numConsts : Term → Nat
  | const _ => 1
  | app _ cs => numConstsLst cs

def numConstsLst : List Term → Nat
  | [] => 0
  | c :: cs => numConsts c + numConstsLst cs
end
```

```
mutual
 def replaceConst (a b : String) : Term → Term
   | const c => if a = c then const b else const c
   | app f cs => app f (replaceConstLst a b cs)

 def replaceConstLst (a b : String) : List Term → List Term
   | [] => []
   | c :: cs => replaceConst a b c :: replaceConstLst a b cs
end
```

Mutual recursion in theorems

```
mutual
  theorem numConsts_replaceConst : numConsts (replaceConst a b e) = numConsts e := by
    match e with
    | const c => simp [replaceConst]; split <;> simp [numConsts]
    | app f cs => simp [replaceConst, numConsts, numConsts_replaceConstLst a b cs]

  theorem numConsts_replaceConstLst : numConstsLst (replaceConstLst a b es) = numConstsLst es := by
    match es with
    | [] => simp [replaceConstLst, numConstsLst]
    | c :: cs =>
      simp [replaceConstLst, numConstsLst, numConsts_replaceConst a b c,
            numConsts_replaceConstLst a b cs]
end
```

Dependent pattern matching

```
inductive Vector (α : Type u) : Nat → Type u
  | nil : Vector α 0
  | cons : α → Vector α n → Vector α (n+1)

infix:67 "::" => Vector.cons

def Vector.zip : Vector α n → Vector β n → Vector (α × β) n
  | nil, nil => nil
  | a::as, b::bs => (a, b) :: zip as bs

#print Vector.zip
/-
def Vector.zip.{u_1, u_2} : {α : Type u_1} → {n : Nat} → {β : Type u_2} → Vector α n → Vector β n → Vector (α × β) n :=
fun {α} {n} {β} x x_1 =>
 Vector.brecOn (motive := fun {n} x => {β : Type u_2} → Vector β n → Vector (α × β) n) x
   ...
-/
```

Structures

```
structure Point where
  x : Int := 0
  y : Int := 0
 deriving Repr

#eval Point.x (Point.mk 10 20)
-- 10
#eval { x := 10, y := 20 : Point }

def p : Point := { y := 20 }
#eval p.x
#eval p.y
#eval { p with x := 5 }
-- { x := 5, y := 20 }

structure Point3D extends Point where
  z : Int
```

Type classes

```
class ToString (α : Type u) where
  toString : α → String

#check @ToString.toString
-- {α : Type u_1} → [self : ToString α] → α → String

instance : ToString String where
  toString s := s
instance : ToString Bool where
  toString b := if b then "true" else "false"

#eval ToString.toString "hello"
export ToString (toString)
#eval toString true
-- #eval toString (true, "hello") -- Error
instance [ToString α] [ToString β] : ToString (α × β) where
  toString p := "(" ++ toString p.1 ++ ", " ++ toString p.2 ++ ")"

#eval toString (true, "hello")
-- "(true, hello)"
```

Type classes are heavily used in Lean

```
class Mul (α : Type u) where
  mul : α → α → α
infixl:70 " * " => Mul.mul
def double [Mul α] (a : α) := a * a

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 := Nat.mul
  mul_assoc := Nat.mul_assoc
#eval double 5

class Functor (f : Type u → Type v) : Type (max (u+1) v) where
  map : (α → β) → f α → f β
infixr:100 " <$> " => Functor.map

class LawfulFunctor (f : Type u → Type v) [Functor f] : Prop where
  id_map   (x : f α) : id <$> x = x
  comp_map (g : α → β) (h : β → γ) (x : f α) :(h ∘ g) <$> x = h <$> g <$> x
```

Deriving instances automatically

We have seen `deriving Repr` in a few examples.

It is an instance generator.

Lean comes equipped with generators for the following classes.

> `Repr, ToString, Inhabited, BEq, DecidableEq,`
>
> `Hashable, Ord, FromToJson, SizeOf`

Tactics

```
example : p → q → p ∧ q ∧ p := by
  intro hp hq
  apply And.intro
  exact hp
  apply And.intro
  exact hq
  exact hp

example : p → q → p ∧ q ∧ p := by
  intro hp hq; apply And.intro hp; exact And.intro hq hp
```

Structuring proofs

```
example : p → q → p ∧ q ∧ p := by
  intro hp hq
  apply And.intro
  case left => exact hp
  case right =>
    apply And.intro
    case left => exact hq
    case right => exact hp
```

```
case left
p q : Prop
hp : p
hq : q
⊢ p
case right
p q : Prop
hp : p
hq : q
⊢ q ∧ p
```

```
example : p → q → p ∧ q ∧ p := by
  intro hp hq
  apply And.intro
  . exact hp
  . apply And.intro
    . exact hq
    . exact hp
```

intro tactic variants

```
example (p q : α → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro h
  match h with
  | Exists.intro w (And.intro hp hq) => exact Exists.intro w (And.intro hq hp)

example (p q : α → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro (Exists.intro _ (And.intro hp hq))
  exact Exists.intro _ (And.intro hq hp)

example (p q : α → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro ⟨_, hp, hq⟩
  exact ⟨_, hq, hp⟩

example (α : Type) (p q : α → Prop) : (∃ x, p x ∨ q x) → ∃ x, q x ∨ p x := by
  intro
    | ⟨_, .inl h⟩ => exact ⟨_, .inr h⟩
    | ⟨_, .inr h⟩ => exact ⟨_, .inl h⟩
```

Inaccessible names

```
example : ∀ x y : Nat, x = y → y = x := by
  intros
  apply Eq.symm
  assumption
```

```
x✝ y✝ : Nat
: x✝ = y✝
⊢ y✝ = x✝
```

```
example : ∀ x y : Nat, x = y → y = x := by
  intros
  apply Eq.symm
  rename_i a b hab
  exact hab
```

```
case h
a b : Nat
hab : a = b
⊢ a = b
```

More tactics

```
example (p q : Nat → Prop) : (∃ x, p x ∧ q x) → ∃ x, q x ∧ p x := by
  intro h
  cases h with
  | intro x hpq =>
    cases hpq with
    | intro hp hq =>
      exists x

example : p ∧ q → q ∧ p := by
  intro p
  cases p
  constructor <;> assumption

example : p ∧ ¬ p → q := by
  intro h
  cases h
  contradiction
```

Structuring proofs (cont.)

```
example : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro h
  have hp : p := h.left
  have hqr : q ∨ r := h.right
  show (p ∧ q) ∨ (p ∧ r)
  cases hqr with
  | inl hq => exact Or.inl ⟨hp, hq⟩
  | inr hr => exact Or.inr ⟨hp, hr⟩
```

```
example : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro ⟨hp, hqr⟩
  cases hqr with
  | inl hq =>
    have := And.intro hp hq
    apply Or.inl; exact this
  | inr hr =>
    have := And.intro hp hr
    apply Or.inr; exact this
```

Tactic combinators

```
example : p → q → r → p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  intros
  repeat (any_goals constructor)
  all_goals assumption

example : p → q → r → p ∧ ((p ∧ q) ∧ r) ∧ (q ∧ r ∧ p) := by
  intros
  repeat (any_goals (first | assumption | constructor))
```

Rewriting

```
example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  rw [h₂] -- replace k with 0
  rw [h₁] -- replace f 0 with 0

example (f : Nat → Nat) (k : Nat) (h₁ : f 0 = 0) (h₂ : k = 0) : f k = 0 := by
  rw [h₂, h₁]

example (f : Nat → Nat) (a b : Nat) (h₁ : a = b) (h₂ : f a = 0) : f b = 0 := by
  rw [← h₁, h₂]

example (f : Nat → Nat) (a : Nat) (h : 0 + a = 0) : f a = f 0 := by
  rw [Nat.zero_add] at h
  rw [h]

def Tuple (α : Type) (n : Nat) := { as : List α // as.length = n }

example (n : Nat) (h : n = 0) (t : Tuple α n) : Tuple α 0 := by
  rw [h] at t
  exact t
```

Simplifier

```
example (p : Nat → Prop) : (x + 0) * (0 + y * 1 + z * 0) = x * y := by
  simp

example (p : Nat → Prop) (h : p (x * y)) : p ((x + 0) * (0 + y * 1 + z * 0)) := by
  simp; assumption

example (p : Nat → Prop) (h : p ((x + 0) * (0 + y * 1 + z * 0))) : p (x * y) := by
  simp at h; assumption

def f (m n : Nat) : Nat :=
  m + n + m

example (h : n = 1) (h' : 0 = m) : (f m n) = n := by
  simp [h, ← h', f]

example (p : Nat → Prop) (h₁ : x + 0 = x') (h₂ : y + 0 = y')
      : x + y + 0 = x' + y' := by
  simp at *
  simp [*]
```

Simplifier

```
def mk_symm (xs : List α) :=
  xs ++ xs.reverse

@[simp] theorem reverse_mk_symm : (mk_symm xs).reverse = mk_symm xs := by
  simp [mk_symm]

theorem tst : (xs ++ mk_symm ys).reverse = mk_symm ys ++ xs.reverse := by
  simp

#print tst
-- Lean reverse_mk_symm, and List.reverse_append
```

split tactic

```
def f (x y z : Nat) : Nat :=
  match x, y, z with
  | 5, _, _ => y
  | _, 5, _ => y
  | _, _, 5 => y
  | _, _, _ => 1

example : x ≠ 5 → y ≠ 5 → z ≠ 5
       → z = w → f x y w = 1 := by
  intros
  simp [f]
  split
  . contradiction
  . contradiction
  . contradiction
  . rfl
```

```
def g (xs ys : List Nat) : Nat :=
  match xs, ys with
  | [a, b], _ => a+b+1
  | _, [b, c] => b+1
  | _, _      => 1

example (xs ys : List Nat) (h : g xs ys = 0) : False := by
  unfold g at h; split at h <;> simp_arith at h
```

induction tactic

```
example (as : List α) (a : α) : (as.concat a).length = as.length + 1 := by
  induction as with
  | nil => rfl
  | cons x xs ih => simp [List.concat, ih]

example (as : List α) (a : α) : (as.concat a).length = as.length + 1 := by
  induction as <;> simp! [*]
```

Part II: Extending Lean in Lean

Local Imperative Programming in Lean

Monadic programming is ubiquitous in Lean

do notation makes it manageable

```
def main : IO Unit := do
  let stdin ← IO.getStdin
  let name  ← stdin.getLine
  IO.println s!"Hello, {name}!"
```

Emulation of an “ordered sequence of commands” from imperative languages

Local Imperative Programming in Lean

Lean 4 extends *do* notation with

conditional control flow

early return

iteration

mutable variables

```
def main : IO UInt32 := do
  let stdin ← IO.getStdin
  let name  ← stdin.getLine
  if name.isEmpty then
    IO.println "Please enter a name!"
    return 1
  let mut sum := 0
  while true do
    let line ← stdin.getLine
    if line.isEmpty then
      break
    sum := sum + line.toNat!
  IO.println s!"{name}, your sum is {sum}"
  return 0
```

warning: potentially highly addictive

Local Imperative Programming in Lean

Lean 4 is still a purely functional language!

Extended *do* notation is still compiled down to pure, monadic code

```
example [Monad m] [LawfulMonad m] (f : β → α → m β) (xs : List α) :
   (do let mut y := init
       for x in xs do
         y ← f y x
       return y)
 =
   xs.foldlM f init
:= by induction xs generalizing init <;> simp_all!
```

Can be used in pure contexts via the `Id` monad

Extending Lean: Syntax & Semantics

Syntax

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

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

syntax "enum" ident "where" ("|" ident)*
: command
```

open categories

concrete syntax trees

Macros: Syntax → Syntax

```
macro_rules
  | `({ $x:ident < $h | $e }) =>
    `(setOf (fun $x => $x < $h ∧ $e))
  | …

macro_rules
  | `(enum $id where $[| $ids ]*) =>
`(inductive $id where $[| $ids:ident ]*
  namespace $id
    def toString : …)
```

hygienic by default

Racket/Rust-inspired

Elaborators: Syntax → Core

```
elab "⟨" args:term,* "⟩" : term <= τ => do
  let Expr.const C .. := τ.getAppFn | throw …
  let [c] ← getCtors C | throw …
  let stx ← `($(mkIdent c) $args*)
  elabTerm stx τ

elab "trivial" : tactic => do
  …
```

type-aware

flexible order

Macro Showcase: leanprover/doc-gen4

```
syntax jsxAttrName := ident <|> str
syntax jsxAttrVal := str <|> group("{" term "}")
…

syntax "<" ident jsxAttr* "/>" : jsxElement
syntax "<" ident jsxAttr* ">" jsxChild* "</" ident ">" : jsxElement
…

macro_rules
  | `(<$n $attrs* />) =>
    `(Html.element $(quote (toString n.getId)) …)
  | `(<$n $attrs* >$children*</$m>) => …
```

```
def classInstanceToHtml (name : Name) : HtmlM Html :=
  return <li><a href={←declNameToLink name}>{name.toString}</a></li>

def classInstancesToHtml (instances : Array Name) : HtmlM Html :=
  return
    <details class="instances">
      <summary>Instances</summary>
      <ul>
        [← instances.mapM classInstanceToHtml]
      </ul>
    </details>
```

Syntax Showcase: arthurpaulino/FxyLang

```
declare_syntax_cat       literal
syntax ("-" noWs)? num : literal -- int
…

def mkLiteral : Lean.Syntax → Except String Literal
  | `(literal| $[-%$neg]?$n:num) =>
    if neg.isNone
      then return .int <| Int.ofNat n.toNat
      else return .int <| Int.negOfNat n.toNat
  | …
```

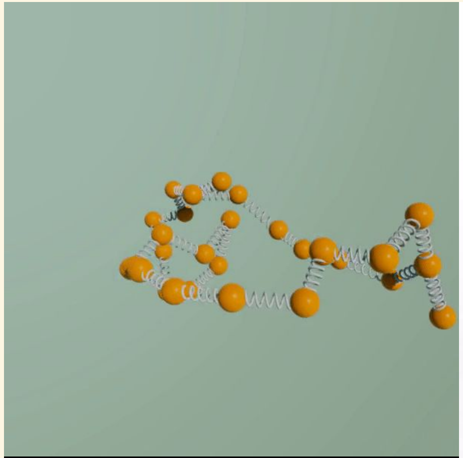
```
#eval >>
f n :=
  s := 0
  i := 0
  while i < n do
    i := i + 1
    s := s + i
  s
!print f 5
<<.run  -- 15
```

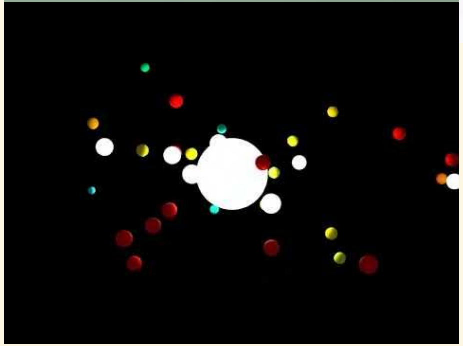
(Meta-)Programming Showcase: lecopivo/SciLean

```
-- wave equation
def H (m k : ℝ) (x p : ℝ^n) : ℝ :=
  let Δx := (1 : ℝ)/(n : ℝ)
  (Δx/(2*m)) * ‖p‖² + (Δx * k/2) * (∑ i , ‖x[i] - x[i - 1]‖²)
argument x
  isSmooth, diff, hasAdjDiff, adjDiff
argument p
  isSmooth, diff, hasAdjDiff, adjDiff

def solver (m k : ℝ) (steps : Nat)
    : Impl (ode_solve (HamiltonianSystem (H m k))) := by
  -- Unfold Hamiltonian definition and compute gradients
  simp [HamiltonianSystem]
  -- Apply RK4 method
  rw [ode_solve_fixed_dt runge_kutta4_step]
  lift_limit steps "Number of ODE solver steps."; admit; simp
  finish_impl
```

Integrated as a scripting language into Houdini





Macro Showcase: [dwrensha/lean4-maze]

```
syntax "░" : game_cell -- empty
syntax "▓" : game_cell -- wall
syntax "@" : game_cell -- player

syntax "│" game_cell* "│\n" : game_row
…

macro_rules
| `(┌ $tb:horizontal_border* ┐
   $rows:game_row*
   └ $bb:horizontal_border* ┘) => …

macro "west" : tactic =>
  `(first | apply step_west; simp | fail "cannot step west")
…
```

```
def maze1 := ┌──────┐
             │▓▓▓▓▓▓│
             │▓░░@░▓│
             │▓░░░░▓│
             │▓░░░░▓│
             │▓▓▓▓░▓│
             └──────┘

example : can_escape maze1 := by
  west
  west
  east
  south
  south
  east
  east
  south
  out
```