Lean 4

Theorem Prover and Programming Language

Leo de Moura - Microsoft Research

Lean for the Curious Mathematician - ICERM - July 15, 2022

How did we get here?

Previous project: Z3 SMT solver

The Lean project started in 2013 with very different goals

- A library for automating proofs in Dafny, F*, Coq, Isabelle, …
- Bridge the gap between interactive and automated theorem proving
- Improve the “lost-in-translation” and proof stability issues

Lean 1.0 - learning DTT

Lean 2.0 (2015) - first official release

Lean 3.0 (2017) - users can write tactics in Lean itself

Extensibility

Lean 3 users extend Lean using Lean

Approximately 5% of Mathlib is Lean extensions

Examples:

Ring Solver, Coinductive predicates, Transfer tactic,

Superposition prover, Linters,

Fourier-Motzkin & Omega, Polyrith, …

Access Lean internals using Lean

Type inference, Unifier, Simplifier, Decision procedures,

Type class resolution, …

Lean 3.x limitations

Lean programs are compiled into byte code and then interpreted (slow).

Lean expressions are foreign objects reflected in Lean.

Very limited ways to extend the parser.

```
infix >=        := ge
infix ≥         := ge
infix >         := gt
```

```
notation `∃` binders `, ` r:(scoped P, Exists P) := r
```

```
notation `[` l:(foldr `, ` (h t, list.cons h t) list.nil `]`) := l
```

Users cannot implement their own elaboration strategies.

Scalability issues, design limitations, missing features, bugs, etc.

It’s been a long time coming …

Parser refactoring + Hygienic macro system #1674

Open **leodemoura** opened this issue on Jun 16, 2017 · 32 comments

“We should really refactor the elaborator as well”

“If we rewrite the frontend, we should do it in Lean”

“We first need a capable Lean compiler for that …”

LEAN 4 begins

Sebastian Ullrich and I started Lean 4 in 2018

Lean in Lean

Extensible programming language and theorem prover

A platform for

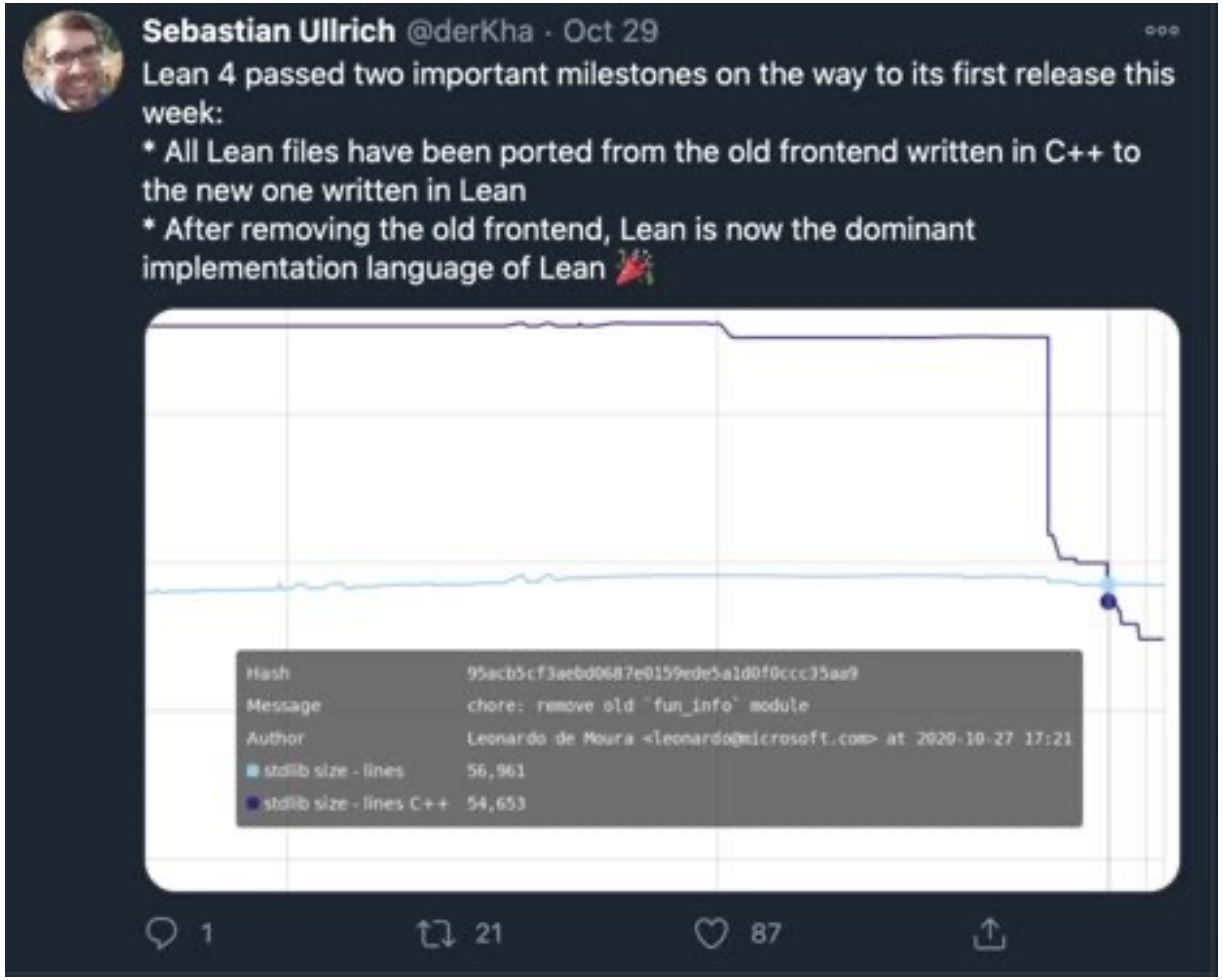
- Software verification
- Formalized mathematics
- Developing custom automation and domain-specific languages (DSL)

Lean 4 is being implemented in Lean

```
inductive Expr where
  | bvar    : Nat → Expr                                    -- bound variables
  | fvar    : FVarId → Expr                                 -- free variables
  | mvar    : MVarId → Expr                                 -- meta variables
  | sort    : Level → Expr                                  -- Sort
  | const   : Name → List Level → Expr                      -- constants
  | app     : Expr → Expr → Expr                            -- application
  | lam     : Name → Expr → Expr → BinderInfo → Expr        -- lambda abstraction
  | forallE : Name → Expr → Expr → BinderInfo → Expr        -- (dependent) arrow
  | letE    : Name → Expr → Expr → Expr → Bool → Expr       -- let expressions
  | lit     : Literal → Expr                                -- literals
  | mdata   : MData → Expr → Expr                           -- metadata
  | proj    : Name → Nat → Expr → Expr                      -- projection
```

```
/- Infer type of lambda and let expressions -/
private def inferLambdaType (e : Expr) : MetaM Expr :=
  lambdaLetTelescope e fun xs e => do
    let type ← inferType e
    mkForallFVars xs type
```

At the end of 2020 Lean 4 compiles itself



Lean 4 first milestone release: Jan 2021

We are using milestone releases for getting feedback from the community.

We are at milestone 4.

We are planning to make the official release at the end of the summer.

We have monthly update meetings online open to the whole community.

Additional details on Zulip and Twitter (leanprover).

Many thanks to the Mathlib community

Mathlib success was instrumental for getting additional funding for the project

2021 was a great year for the Lean project. We now have

- A full-time program manager (Sarah Smith)
- New developer starting soon (pending visa), trying to hire another one next year
- Engineers helping with the VS Code Lean extension and infrastructure
- Contractor for writing an introductory book for Lean
- (Trying to) hire 4 Mathlib maintainers to help with the port
- Academic gifts

Augmented Mathematical Intelligence (AMI) at Microsoft

Mission

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

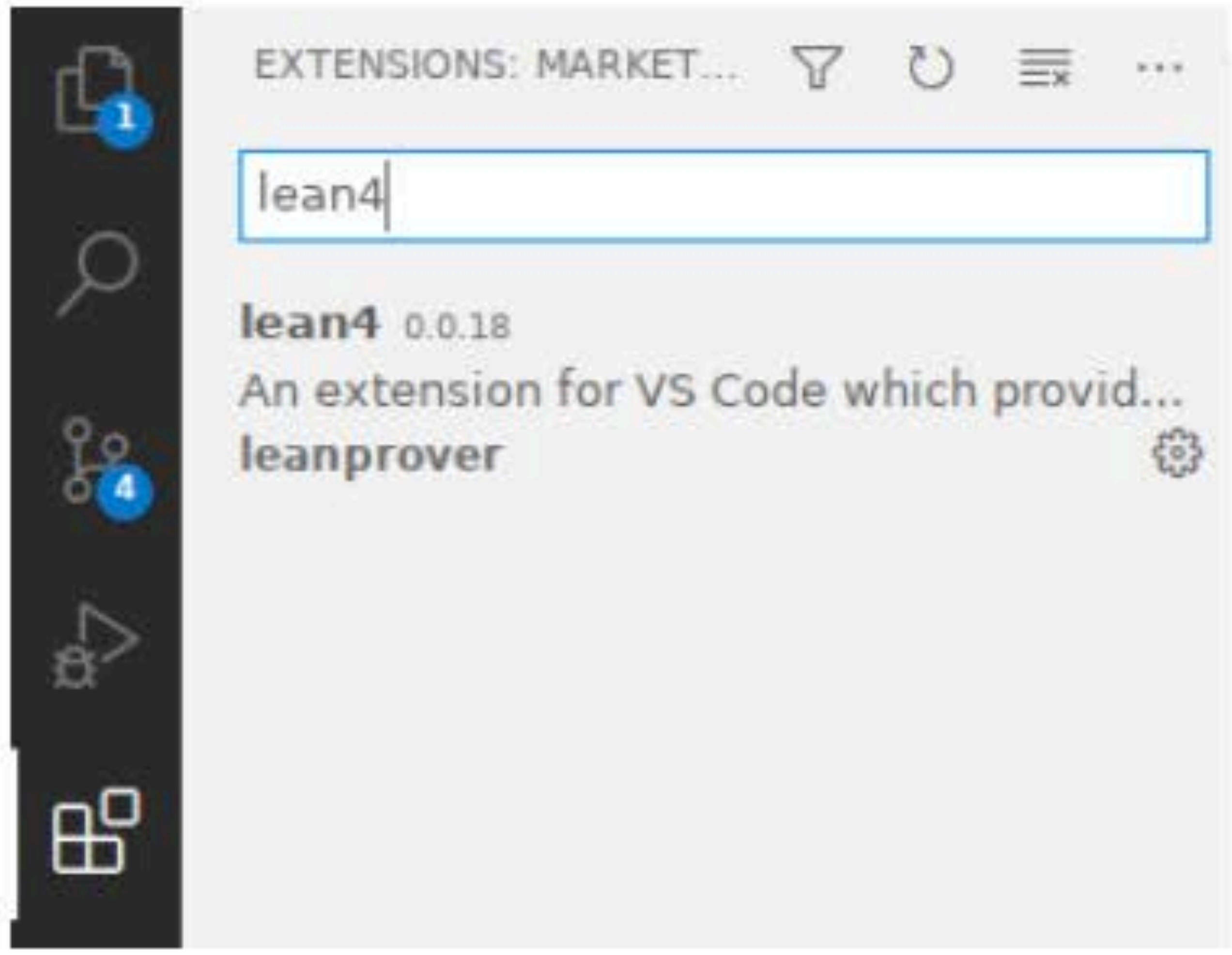
Lean 4 quick start

These instructions will walk you through setting up Lean using the "basic" setup and VS Code as the editor. See Setup for other ways, supported platforms, and more details on setting up Lean.

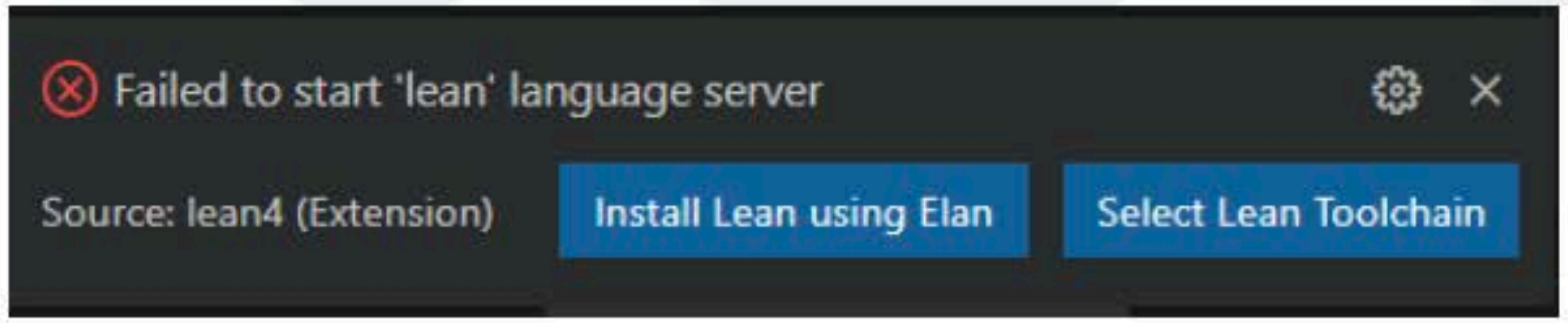
See quick walkthrough demo video.

1. Install VS Code.

2. Launch VS Code and install the `lean4` extension.



3. Create a new file using "File > New Text File" (`Ctrl+N`). Click the `Select a language` prompt, type in `lean4`, and hit ENTER. You should see the following popup:



You can use Lean 3 and Lean 4 simultaneously

Thanks to `elan` (by Sebastian Ullrich)

If you use Lean 3 you are probably already using `elan`

`elan` is the Lean version manager

Theorem Proving in Lean 4

https://leanprover.github.io/theorem_proving_in_lean4/

Theorem Proving in Lean 4

Theorem Proving in Lean 4

by Jeremy Avigad, Leonardo de Moura, Soonho Kong and Sebastian Ullrich, with contributions from the Lean Community

This version of the text assumes you're using Lean 4. See the Setting Up Lean section of the Lean 4 Manual to install Lean. The first version of this book was written for Lean 2, and the Lean 3 version is is available here.

Functional Programming in Lean

By David Christiansen

https://leanprover.github.io/functional_programming_in_lean/introduction.html

It is be updated monthly

Functional Programming in Lean

Lean is an interactive theorem prover developed at Microsoft Research, based on dependent type theory. Dependent type theory unites the worlds of programs and proofs; thus, Lean is also a programming language. Lean takes its dual nature seriously, and it is designed to be suitable for use as a general-purpose programming language—Lean is even implemented in itself. This book is about writing programs in Lean.

Many tutorial like examples

Powered by LeanInk

https://leanprover.github.io/lean4/doc/examples

Recall that, `def Expr.typeCheck ...` in Lean is notation for `namespace Expr def typeCheck ... end Expr`. The term `.found .nat .nat` is sugar for `Maybe.found Ty.nat HasType.nat`. Lean can infer the namespaces using the expected types.

```
def Expr.typeCheck (e : Expr) : {{ ty | HasType e ty }} :=
  match e with
  | nat ..   => .found .nat .nat
  | bool ..  => .found .bool .bool
  | plus a b =>
    match a.typeCheck, b.typeCheck with
    | .found .nat h₁, .found .nat h₂ => .found .nat (.plus h₁ h₂)
    | _, _ => .unknown
  | and a b =>
    match a.typeCheck, b.typeCheck with
    | .found .bool h₁, .found .bool h₂ => .found .bool (.and h₁ h₂)
    | _, _ => .unknown

theorem Expr.typeCheck_correct (h₁ : HasType e ty) (h₂ : e.typeCheck ≠ .unknown)
```

KIT lecture notes

Sebastian Ullrich's lecture notes for the following course based on Lean 4.

Theorem prover lab: applications in programming languages

https://github.com/IPDSnelting/tba-2022

https://github.com/IPDSnelting/tba-2021

Slides, exercises, and a lot of useful information about Lean 4.

The 2022 version uses the new Aesop tactic.

Metaprogramming in Lean

Manual being developed by the community.

Many thanks to Arthur Paulino for spearheading this effort.

https://github.com/arthurpaulino/lean4-metaprogramming-book

Porting Mathlib

Mathlib is massive, almost 1 million lines of code.

Lean 4 is not backward compatible with Lean 3.

Mathlib was much smaller when we started Lean 4 (approx. 45 thousand lines).

Mathport tool (by Mario Carneiro and Daniel Selsam).

- Ports Lean 3 files to Lean 4. We also have support for porting Lean 3 object files.
- It can’t port user-extensions (Mathlib tactic folder).

Mathlib has more 40 thousand lines of user-extensions.

- It will be ported manually this summer.
- Four Mathlib maintainers will be working as contractors. One of them will be full-time.
- Hackton style events.

Porting Mathlib

Rest of the talk: motivations for doing it.

LEAN 4 Compiler

Code specialization, simplification, and many other optimizations (beginning of 2019)

Generates C code

Safe destructive updates in pure code - FBIP idiom

“Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming”, Ullrich, Sebastian; de Moura, Leonardo

Benchmark	**Lean**	del	cm	**GHC**	gc	cm	**OCaml**	gc	cm
binarytrees	**1.36s**	40%	37 M/s	4.09	72	120	1.63	NA	NA
deriv	**0.99**	24	32	1.87	51	32	1.42	76	59
constfold	**1.98**	11	83	4.41	64	51	9.22	91	107
qsort	**2.27**	9	0	3.70	1	0	3.1	13	1
rbmap	**0.57**	2	6	1.37	39	24	**0.57**	31	27
rbmap_1	**0.83**	15	34	9.32	88	47	1.1	60	59
rbmap_10	**2.9**	27	55	9.41	88	48	5.86	88	89

LEAN 4 FBIP

It changes how you write pure functional programs

Hash tables and arrays are back

It is way easier to use than linear type systems. It is not all-or-nothing

Lean 4 persistent arrays are fast

"Counting immutable beans" in the Koka programming language

- "Perceus: Garbage Free Reference Counting with Reuse" (2020)
  Reinking, Alex; Xie, Ningning; de Moura, Leonardo; Leijen, Daan

Lean 4 red-black trees outperform non-persistent version at C++ stdlib

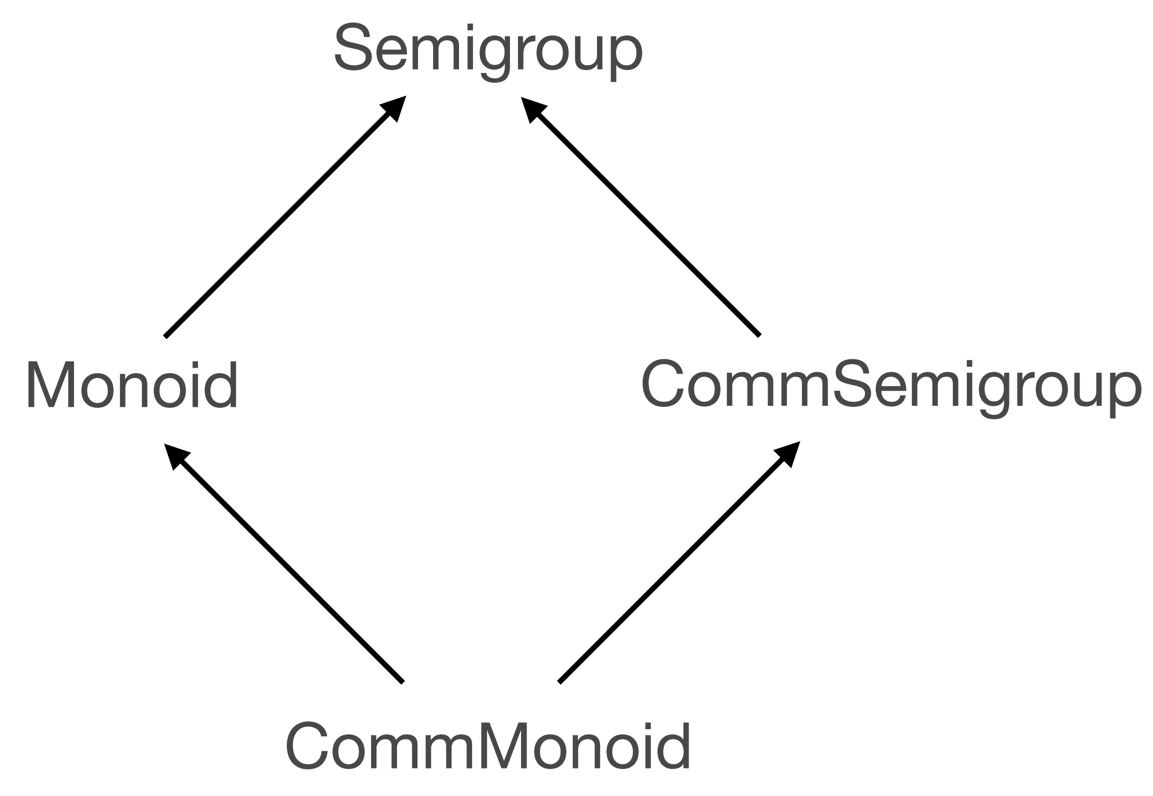
- Result has been reproduced in Koka

LEAN 4 Type class resolution

Type classes provide an elegant and effective way of managing ad-hoc polymorphism

Lean 3 TC limitations: diamonds, cycles, naive indexing

There is no ban on diamonds in Lean 3 or Lean 4



New algorithm based on tabled resolution

> "Tabled Type class Resolution"
> Selsam, Daniel; Ullrich, Sebastian; de Moura, Leonardo
>
> Addresses the first two issues

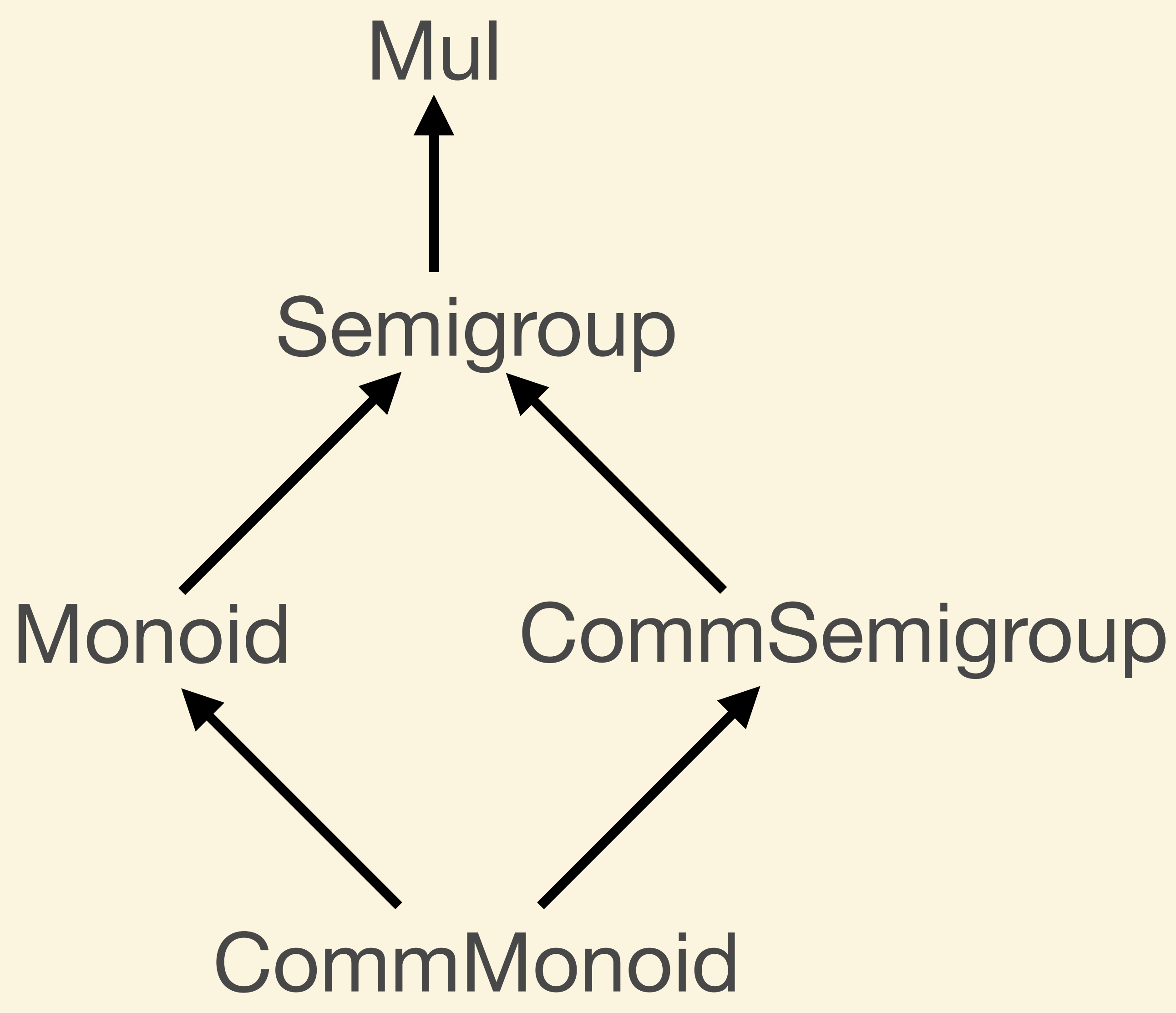
More efficient indexing based on (DTT-friendly) "discrimination trees"

Discrimination trees are also used to index: unification hints, and simp lemmas

Multiple inheritance and scalability

Lean 3 “old_structure_cmd” generates flat structures that do not scale well

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

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

class One (α : Type u) where
  one : α

instance [One α] : OfNat α (nat_lit 1) where
  ofNat := One.one

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

#check @CommMonoid.mk
-- @CommMonoid.mk : {α : Type u_1} → [toMonoid : Monoid α] → (∀ (a b : α), a * b = b * a) → CommMonoid α
```



LEAN 4 Hygienic macro system

"Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages"
Ullrich, Sebastian; de Moura, Leonardo

```
syntax "{ " ident (" : " term)? " // " term " }" : term

macro_rules
  | `({ $x : $type // $p }) => ``(Subtype (fun ($x:ident : $type) => $p))
  | `({ $x // $p })         => ``(Subtype (fun ($x:ident : _) => $p))
```

LEAN 4 Hygienic macro system

Hygiene = no accidental name capture.

```
macro "const " e:term : term => `(fun x => $e)

#eval (fun x => const (x+1)) 10 true
-- 11
```

LEAN 4 Hygienic macro system

We have many different syntax categories.

```
syntax:arg stx:max "+" : stx
syntax:arg stx:max "*" : stx
syntax:arg stx:max "?" : stx
syntax:2 stx:2 " <|> " stx:1 : stx

macro_rules
  | `(stx| $p +) => `(stx| many1($p))
  | `(stx| $p *) => `(stx| many($p))
  | `(stx| $p ?) => `(stx| optional($p))
  | `(stx| $p₁ <|> $p₂) => `(stx| orelse($p₁, $p₂))
```

LEAN 4 Big operator notation: an example

```
def bigop (init : β) (seq : List α) (op : β → β → β) (f : α → Option β) : β := Id.run do
  let mut result := init
  for a in seq do
    if let some b := f a then
      result := op result b
  return result

#eval bigop 0 [2, 3, 4] (·+·) fun elem => if elem % 2 == 0 then some (elem * 2) else none
-- 12

#eval
   bigop
    (init := 0)
    (seq := [2, 3, 4])
    (op := Nat.add)
    (f := fun elem => if elem % 2 == 0 then some (elem * 2) else none)
```

LEAN 4 Big operator notation: an example

```
def iota : Nat → Nat → List Nat
  | _, 0   => []
  | m, n+1 => m :: iota (m+1) n

def range (m n : Nat) := iota m (n - m)

#eval range 2 6
-- [2, 3, 4, 5]
```

LEAN 4 Big operator notation: an example

```
-- Declare a new syntax category for "indexing" big operators
declare_syntax_cat index
syntax term:51 "≤" ident "<" term : index
syntax term:51 "≤" ident "<" term "|" term : index
syntax ident "<-" term : index
syntax ident "<-" term "|" term : index
-- Primitive notation for big operators
syntax "_big" "[" term "," term "]" "(" index ")" term : term

-- We define how to expand `_big` with the different kinds of index
macro_rules
| `(_big [$op, $ini] ($lower:term ≤ $i < $upper) $F)
  =>
  `(bigop $ini (range $lower $upper) $op (fun $i:ident => some $F))
| `(_big [$op, $ini] ($i:ident <- $col | $p) $F)
  =>
  `(bigop $ini $col $op (fun $i:ident => if $p then some $F else none))
```

LEAN 4 Big operator notation: an example

```
-- Define `∑ `
syntax "∑ " "(" index ")" term : term
macro_rules | `(∑ ($idx) $F) => `(_big [Add.add, 0] ($idx) $F)

-- We can already use `Sum` with the different kinds of index.
#check ∑ (i <- [0, 2, 4] | i != 2) i
#eval  ∑ (1 ≤ i < 4) 2*i
-- 12


-- Define `∏`
syntax "∏" "(" index ")" term : term
macro_rules | `(∏ ($idx) $F) => `(_big [Mul.mul, 1] ($idx) $F)

-- The examples above now also work for `Prod`
#check ∏ (i <- [0, 2, 4] | i != 2) i
#eval ∏ (1 ≤ i < 4) 2*i
-- 48
```

LEAN 4 Big operator notation: an example

```
-- We can extend our grammar for the syntax category `index`.
syntax ident "|" term : index
syntax ident ":" term : index
syntax ident ":" term "|" term : index
-- And new rules
macro_rules
| `(_big [$op, $idx] ($i:ident : $type) $F)      => `(bigop $idx (elems (α := $type)) $op (fun $i:ident => some $F))
| `(_big [$op, $idx] ($i:ident : $type | $p) $F) => `(bigop $idx (elems (α := $type)) $op (fun $i:ident => if $p then some $F else none))
| `(_big [$op, $idx] ($i:ident | $p) $F)         => `(bigop $idx elems $op (fun $i:ident => if $p then some $F else none))

-- The new syntax is immediately available for all big operators that we have defined
def myPred (i : Fin 10) : Bool := i % 2 = 1
#check ∑ (i : Fin 10) i+1
#check ∑ (i : Fin 10 | i != 2) i+1
#check ∑ (i | myPred i) i+i
#check ∏ (i : Fin 10) i+1
#check ∏ (i : Fin 10 | i != 2) i+1
```

LEAN 4 Hygienic macro system

Many Lean 3 tactics are just macros, and they can be recursive.

```
syntax "funext " (colGt term:max)+ : tactic

macro_rules
  | `(tactic|funext $x) => `(tactic| apply funext; intro $x:term)
  | `(tactic|funext $x $xs*) => `(tactic| apply funext; intro $x:term; funext $xs*)
```

```
def f (x y : Nat × Nat) := x.1 + y.2
def g (x y : Nat × Nat) := y.2 + x.1

example : f = g := by
  funext (a, _) (_, d)
  show a + d = d + a
  rw [Nat.add_comm]
```

LEAN 4 Hygienic and typed macro system

```
syntax "#show" term : command

macro_rules
  | `(#show $e) => `(#print $e) -- Error `e` is Term, but ident or str expected

macro_rules
  | `(#show $e:ident) => `(#print $e) -- Ok
  | `(#show $e) => `(#check $e)

#show toString

#show 2+2
```

LEAN 4 Structured (and hygienic) tactic language

```
-- less-than is well-founded
def lt_wfRel : WellFoundedRelation Nat where
  rel := Nat.lt
  wf  := by
  apply WellFounded.intro
  intro n
  induction n with
  | zero      =>
    apply Acc.intro 0
    intro _ h
    apply absurd h (Nat.not_lt_zero _)
  | succ n ih =>
    apply Acc.intro (Nat.succ n)
    intro m h
    have : m = n ∨ m < n := Nat.eq_or_lt_of_le (Nat.le_of_succ_le_succ h)
    match this with
    | Or.inl e => subst e; assumption
    | Or.inr e => exact Acc.inv ih e
```

LEAN 4 Structured (and hygienic) tactic language

`match … with` works in tactic mode, and it is just a macro

```
theorem concatEq (xs : List α) (h : xs ≠ []) : concat (dropLast xs) (last xs h) = xs := by
  match xs, h with
  | [],  h         => contradiction
  | [x], h         => rfl
  | x₁::x₂::xs, _ => simp [concat, last, concatEq (x₂::xs) List.noConfusion]
```

LEAN 4 Structured (and hygienic) tactic language

Multi-target induction

```
theorem mod.inductionOn
      {motive : Nat → Nat → Sort u}
      (x y  : Nat)
      (ind  : ∀ x y, 0 < y ∧ y ≤ x → motive (x - y) y → motive x y)
      (base : ∀ x y, ¬(0 < y ∧ y ≤ x) → motive x y)
      : motive x y :=
  div.inductionOn x y ind base
```

```
theorem mod_lt (x : Nat) {y : Nat} : y > 0 → x % y < y := by
  induction x, y using mod.inductionOn with
  | base x y h₁ =>
    intro h₂
    have h₁ : ¬ 0 < y ∨ ¬ y ≤ x := Iff.mp (Decidable.not_and_iff_or_not _ _) h₁
    match h₁ with
    | Or.inl h₁ => exact absurd h₂ h₁
    | Or.inr h₁ =>
      have hgt : y > x := gt_of_not_le h₁
      have heq : x % y = x := mod_eq_of_lt hgt
      rw [← heq] at hgt
      exact hgt
  | ind x y h h₂ =>
    intro h₃
    have ⟨_, h₁⟩ := h
```

LEAN 4 Structured (and hygienic) tactic language

Default elimination principle.

```lean
@[eliminator] protected def Nat.recDiag {motive : Nat → Nat → Sort u}
    (zero_zero : motive 0 0)
    (succ_zero : (x : Nat) → motive x 0 → motive (x + 1) 0)
    (zero_succ : (y : Nat) → motive 0 y → motive 0 (y + 1))
    (succ_succ : (x y : Nat) → motive x y → motive (x + 1) (y + 1))
    (x y : Nat) :  motive x y :=
```

```lean
def f (x y : Nat) :=
  match x, y with
  | 0,   0   => 1
  | x+1, 0   => f x 0
  | 0,   y+1 => f 0 y
  | x+1, y+1 => f x y
termination_by f x y => (x, y)

example (x y : Nat) : f x y > 0 := by
  induction x, y with
  | zero_zero => decide
  | succ_zero x ih => simp [f, ih]
  | zero_succ y ih => simp [f, ih]
  | succ_succ x y ih => simp [f, ih]

example (x y : Nat) : f x y > 0 := by
  induction x, y <;> simp [f, *]
```

LEAN 4 Structured (and hygienic) tactic language

By default tactic generated names are "inaccessible"
You can disable this behavior using the following command

```
set_option tactic.hygienic false in
example {a p q r : Prop} : p → (p → q) → (q → r) → r := by
  intro _ h1 h2
  apply h2
  apply h1
  exact a_1 -- Bad practice, using name generated by `intro`.

example {a p q r : Prop} : p → (p → q) → (q → r) → r := by
  intro _ h1 h2
  apply h2
  apply h1
  exact a_1 -- error "unknown identifier"

example {a p q r : Prop} : p → (p → q) → (q → r) → r := by
  intro _ h1 h2
  apply h2
  apply h1
  assumption
```

LEAN4 simp

Lean 3 simp is a major bottleneck

Two sources of inefficiency: simp set is reconstructed all the time, poor indexing

Indexing in DTT is complicated because of definitional equality

Lean 3 simp uses keyed matching (Georges Gonthier)

Keyed matching works well for the rewrite tactic because there are few failures

lean4 mathlib performance issues — Nov 06, 2019

Daniel Selsam (EDITED) 4:12 PM

There are 15,000,000 simp failures in mathlib (top few in reverse):

```
n_fails | simp lemma name
--------------------------------------
 36845 FAIL: sub_right_inj
  36858 FAIL: mul_eq_zero
  36879 FAIL: prod.mk.inj_iff
  36895 FAIL: inv_eq_one
  36923 FAIL: sub_left_inj
  37108 FAIL: sum.inl.inj_iff
  37132 FAIL: sum.inr.inj_iff
  37202 FAIL: sum.inr_ne_inl
  37208 FAIL: sum.inl_ne_inr
  37232 FAIL: tt_eq_ff_eq_false
```

```
@[simp] lemma sub_right_inj : a - b = a - c ↔ b = c :=
(add_right_inj _).trans neg_inj'
```

LEAN 4 simp

Lean 4 uses discrimination trees to index simp sets

It is the same data structure used to index type class instances

Here is a synthetic benchmark

```
@[simp] axiom s0 (x : Prop) : f (g1 x) = f (g0 x)
@[simp] axiom s1 (x : Prop) : f (g2 x) = f (g1 x)
@[simp] axiom s2 (x : Prop) : f (g3 x) = f (g2 x)
```

...

```
@[simp] axiom s498 (x : Prop) : f (g499 x) = f (g498 x)
def test (x : Prop) : f (g0 x) = f (g499 x) := by simp
#check test
```

num. lemmas + 1	Lean 3	Lean4
500	0.89s	0.18s
1000	2.97s	0.39s
1500	6.67s	0.61s
2000	11.86s	0.71s
2500	18.25s	0.93s
3000	26.90s	1.15s

LEAN 4 match … with

There is no equation compiler

Pattern matching, and termination checking are completely decoupled

Example:

```
def eraseIdx : List α → Nat → List α
  | [],     _   => []
  | _::as, 0    => as
  | a::as, n+1 => a :: eraseIdx as n
```

expands into

```
def eraseIdx (as : List α) (i : Nat) : List α :=
  match as, i with
  | [],     _   => []
  | _::as, 0    => as
  | a::as, n+1 => a :: eraseIdx as n
```

LEAN 4 match … with

```
def eraseIdx (as : List α) (i : Nat) : List α :=
  match as, i with
  | [],     _    => []
  | _::as, 0     => as
  | a::as, n+1 => a :: eraseIdx as n
```

We generate an auxiliary "matcher" function for each `match … with`
The matcher doesn't depend on the right-hand side of each alternative

```
{α : Type u_1} →
(motive : List α → Nat → Sort u_2) →
-- discriminants
(as : List α) →
(i : Nat) →
-- alternatives
((x : Nat) → motive [] x) →
((head : α) → (as : List α) → motive (head :: as) 0) →
((a : α) → (as : List α) → (n : Nat) → motive (a :: as) (Nat.succ n)) →
motive as i
```

LEAN 4 match … with

```
def eraseIdx (as : List α) (i : Nat) : List α :=
  match as, i with
  | [],     _    => []
  | _::as, 0    => as
  | a::as, n+1 => a :: eraseIdx as n
```

The new representation has many advantages
We can "change" the motive when proving termination
We "hides" all nasty details of dependent pattern matching

pp of the kernel term

```
def eraseIdx.{u_1} : {α : Type u_1} → List α → Nat → List α :=
fun {α} as i =>
  List.brecOn as
    (fun as f i =>
      (match as, i with
        | [], x => fun x => []
        | head :: as, 0 => fun x => as
        | a :: as, Nat.succ n => fun x => a :: PProd.fst x.fst n)
        f)
    i
```

LEAN 4 match … with

Equality proofs (similar to if-then-else)

```
@[inline] def withPtrEqDecEq {α : Type u} (a b : α) (k : Unit → Decidable (a = b)) : Decidable (a = b) :=
  let b := withPtrEq a b (fun _ => toBoolUsing (k ())) (toBoolUsing_eq_true (k ()));
  match h : b with
  | true  => isTrue (ofBoolUsing_eq_true h)
  | false => isFalse (ofBoolUsing_eq_false h)
```

LEAN 4 split tactic

Useful for reasoning about `match-with` containing overlapping patterns

```
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
```

```
x y z w : Nat
: x ≠ 5
: y ≠ 5
: z ≠ 5
: z = w
⊢ (match x, y, w with
  | 5, x, x_1 => y
  | x, 5, x_1 => y
  | x, x_1, 5 => y
  | x, x_1, x_2 => 1) =
  1
```

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

LEAN 4 Recursion

Termination checking is independent of pattern matching

`mutual` and `let rec` keywords

We compute blocks of strongly connected components (SCCs)

Each SCC is processed using one of the following strategies

  non rec, structural, unsafe, partial, well-founded.

```
def eraseIdx.{u_1} : {α : Type u_1} → List α → Nat → List α :=
fun {α} as i =>
  List.brecOn as
    (fun as f i =>
      (match as, i with
        | [], x => fun x => []
        | head :: as, 0 => fun x => as
        | a :: as, Nat.succ n => fun x => a :: PProd.fst x.fst n)
        f)
    i
```

LEAN 4 Avoiding auxiliary declarations with `let rec`

```
private def addSCC (a : α) : M α Unit := do
  let rec add
    | [],     newSCC => modify fun s => { s with stack := [], sccs := newSCC :: s.sccs }
    | b::bs, newSCC => do
      resetOnStack b;
      let newSCC := b::newSCC;
      if a != b then
        add bs newSCC
      else
        modify fun s => { s with stack := bs, sccs := newSCC :: s.sccs }
  add (← get).stack []
```

LEAN 4 Haskell-like “where” clause

Expands into `let rec`

```
def Tree.toListTR (t : Tree β) : List (Nat × β) :=
  go t []
where
  go (t : Tree β) (acc : List (Nat × β)) : List (Nat × β) :=
    match t with
    | leaf => acc
    | node l k v r => go l ((k, v) :: go r acc)
```

```
theorem Tree.toList_eq_toListTR (t : Tree β)
        : t.toList = t.toListTR := by
  simp [toListTR, go t []]
where
  go (t : Tree β) (acc : List (Nat × β))
     : toListTR.go t acc = t.toList ++ acc := by
    induction t generalizing acc <;>
      simp [toListTR.go, toList, *, List.append_assoc]
```

Termination Checker

```
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 a b => (a, b)
```

```
def indexOf [DecidableEq α] (a : Array α) (v : α) : Option (Fin a.size) :=
  go 0
where
  go (i : Nat) :=
    if h : i < a.size then
      if a[i] = v then some ⟨i, h⟩ else go (i+1)
    else
      none
termination_by go i => a.size - i
```

Termination Checker - Mutual Recursion

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

```
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
  theorem numConsts_replaceConst (a b : String) (e : Term)
            : 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 (a b : String) (es : List Term)
            : 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
```

LEAN 4 Elaborator: postpone and resume

Lean 3 has very limited support for postponing the elaboration of terms

```
def ex1 (xs : list (list nat)) : io unit :=
  io.print_ln (xs.foldl (fun r x, r.union x) []) -- dot-notation fails at `r.union x`

def ex2 (xs : list (list nat)) : io unit :=
  io.print_ln (xs.foldl (fun (r : list nat) x, r.union x) []) -- fix: provide type
```

LEAN 4 Elaborator: postpone and resume

```
def ex1 (xs : List (List Nat)) : IO Unit :=
  IO.println (xs.foldl (fun r x => r.union x) [])
```

Same example using named arguments

```
def ex1 (xs : List (List Nat)) : IO Unit :=
  IO.println $ xs.foldl (init := []) fun r x => r.union x
```

Same example using anonymous function syntax sugar, and F# style $

```
def ex1 (xs : List (List Nat)) : IO Unit :=
  IO.println $ xs.foldl (init := []) (·.union ·)
```

LEAN 4 Heterogeneous operators

In Lean3, +, *, -, / are all homogeneous polymorphic operators

```
has_add.add : Π {α : Type u_1} [c : has_add α], α → α → α
```

What about matrix multiplication?

Nasty interactions with coercions.

```
variables (x : nat) (i : int)

#check i + x -- ok
#check x + i -- error
```

Rust supports heterogenous operators

LEAN 4 Heterogeneous operators in action

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

instance [Mul α] [Add α] [Zero α] : HMul (Matrix m n α) (Matrix n p α) (Matrix m p α) where
  hMul x y i j := dotProduct (x[i, ·]) (y[·, j])

instance [Mul α] : HMul α (Matrix m n α) (Matrix m n α) where
  hMul c x i j := c * x[i, j]
```

```
example (a b : Nat) (x : Matrix 10 20 Nat) (y : Matrix 20 10 Nat) (z : Matrix 10 10 Nat) : Matrix 10 10 Nat :=
  a * x * y + b * z

example (a b : Nat) (x : Matrix m n Nat) (y : Matrix n m Nat) (z : Matrix m m Nat) : Matrix m m Nat :=
  a * x * y + b * z
```

LEAN 4 Scoped attributes

Lean 4 supports scoped instances, notation, unification hints, simp lemmas, …

```
namespace NameOp
  scoped infixl:65 (priority := high) " + " => Nat.add
  scoped infixl:70 (priority := high) " * " => Nat.mul
end NameOp

variable (n : Nat) (i : Int)
#check n + i -- Using heterogeneous operators
#check i + n
open NameOp
#check n + n
#check n + i -- Error
```

LEAN 4 Implicit lambdas

New feature: implicit lambdas

```
structure state_t (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
(run : σ → m (α × σ))

def state_t.pure {σ} {m} [monad m] {α} (a : α) : state_t σ m α :=
⟨λ s, pure (a, s)⟩

def state_t.bind {σ} {m} [monad m] {α β} (x : state_t σ m α) (f : α → state_t σ m β) : state_t σ m β :=
⟨λ s, do (a, s') ← x.run s, (f a).run s'⟩

instance {σ} {m} [monad m] : monad (state_t σ m) :=
{ pure := @state_t.pure _ _ _,
  bind := @state_t.bind _ _ _ }
```

↑

The Lean 3 curse of @s and _s

LEAN 4 Implicit lambdas

The Lean 3 double curly braces workaround

```
structure state_t (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
(run : σ → m (α × σ))

def state_t.pure {σ} {m} [monad m] {{α}} (a : α) : state_t σ m α :=
⟨λ s, pure (a, s)⟩

def state_t.bind {σ} {m} [monad m] {{α β}} (x : state_t σ m α) (f : α → state_t σ m β) : state_t σ m β :=
⟨λ s, do (a, s') ← x.run s, (f a).run s'⟩

instance {σ} {m} [monad m] : monad (state_t σ m) :=
{ pure := state_t.pure,
  bind := state_t.bind }
```

LEAN 4 Implicit lambdas

The Lean 4 way: no @s, _s, {{}}s

```
def StateT (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
  σ → m (α × σ)

protected def pure [Monad m] (a : α) : StateT σ m α :=
  fun s => pure (a, s)

protected def bind [Monad m] (x : StateT σ m α) (f : α → StateT σ m β) : StateT σ m β :=
  fun s => do let (a, s) ← x s; f a s

instance [Monad m] : Monad (StateT σ m) where
  pure := StateT.pure
  bind := StateT.bind
```

LEAN 4 Implicit lambdas

We can make it nicer:

```
def StateT (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
  σ → m (α × σ)

instance [Monad m] : Monad (StateT σ m) where
  pure a   := fun s => pure (a, s)
  bind x f := fun s => do let (a, s) ← x s; f a s
```

It is equivalent to

```
def StateT (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
  σ → m (α × σ)

instance [Monad m] : Monad (StateT σ m) where
  pure a s   := pure (a, s)
  bind x f s := do let (a, s) ← x s; f a s
```

Fine-grain checkpoints

lean4 › Partial evaluation of a file — Mar 30

Mario Carneiro — 11:17 PM

The reasons to break up proofs also have to do with readability to other people, not just lean. These huge proofs are really not desirable in any sense. Of course we should try to address these issues with tooling support where possible, but it's papering over the issue. I am reminded of an adage I learned from who knows where: if you hit a system limit like a timeout or stack overflow, you should first consider whether you are doing something wrong before increasing the limit

11:20 PM — The "elementarity" of the proof has nothing at all to do with it. You can have large proofs and modular proofs at the high level and the low level equally well

11:23 PM — in CS, a common rule of thumb is to not let your functions get too large or too deeply nested. Saying "it's okay because I'm building on a big framework" is not an excuse, and the rule is not related to the effectiveness of the compiler on your code (although if you let it get really bad then you might hit a compiler limit).

Patrick Massot — 11:38 PM

Having abstract principles like this sounds nice, but math simply doesn't work like this.

👍 2

11:40 PM — And cutting a proof into ten lemmas that have the same assumptions and are used only once doesn't increase readability.

...

```
simp at h
split at h <;> simp <;> try assumption
rename_i k₁ v₁ m₁ k₂ v₂ m₂
save -- Local checkpoint
by_cases hltv : Nat.blt v₁ v₂ <;> simp [hltv] at h
· have ih := ih (h := h); simp [denote_eq] at ih ⊢; assumption
```

...

Unification hints & bundled structures

```
structure Magma where
  carrier   : Type u
  mul : carrier → carrier → carrier

instance : CoeSort Magma (Type u) where
  coe m := m.carrier

def mul {s : Magma} (a b : s) : s := s.mul a b
infixl:70 (priority := high) " * " => mul

example {S : Magma} (a b c : S) : b = c → a * b = a * c := by simp_all

def Nat.Magma : Magma where
  carrier := Nat
  mul a b := Nat.mul a b

example (x : Nat) : Nat := x * x -- type mismatch, ?m.carrier =?= Nat

unif_hint (s : Magma) where
  s =?= Nat.Magma |- s.carrier =?= Nat

example (x : Nat) : Nat := x * x
```

Unification hints & bundled structures

```
def Prod.Magma (m : Magma) (n : Magma) : Magma where
  carrier := m.carrier × n.carrier
  mul a b := (a.1 * b.1, a.2 * b.2)

unif_hint (s : Magma) (m : Magma) (n : Magma) (β : Type u) (δ : Type v) where
  m.carrier =?= β
  n.carrier =?= δ
  s =?= Prod.Magma m n
  |-
  s.carrier =?= β × δ

example (x y : Nat) : Nat × Nat × Nat:=
  (x, y, x) * (x, y, y)
```

Unification hints & type classes: bridge

```
def magmaOfMul (α : Type u) [Mul α] : Magma where -- Bridge between `Mul α` and `Magma`
  carrier := α
  mul a b := Mul.mul a b

unif_hint (s : Magma) (α : Type u) [Mul α] where
  s =?= magmaOfMul α
  |-
  s.carrier =?= α

example (x y : Int) : Int :=
  x * y * x -- Note that we don't have a hint connecting Magma's carrier and Int
```

Definitional Eta for Structures

gebner commented on Nov 9, 2021 • edited

...

Concretely, the following are examples of definitional equalities that would be nice to have:

- `s = { x | x ∈ s }`
- `op (unop x) = x`
- `to_dual (of_dual x) = x`
- `e.symm.symm = e`

Relevant Zulip discussions (non-exhaustive):

- Sets and order_dual: https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/with_top.20irreducible
- Sets and additive/multiplicative: https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/universe.20juggling
- Equivalences: https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/Unexpected.20non-defeq
- Opposites in category theory: https://leanprover.zulipchat.com/#narrow/stream/144837-PR-reviews/topic/.23538.20opposites

Definitional Eta for Structures

```
class TopologicalSpace (α : Type) where
  -- ..

structure Homeomorph (α β : Type) [TopologicalSpace α] [TopologicalSpace β] extends Equiv α β where
  continuousToFun : to_do -- ..
  continuousInv   : to_do -- ..

def Homeomorph.symm [TopologicalSpace α] [TopologicalSpace β] (f : Homeomorph α β) : Homeomorph β α where
  toFun := f.inv
  inv := f.toFun
  continuousToFun := f.continuousInv
  continuousInv   := by trivial

example [TopologicalSpace α] [TopologicalSpace β] (f : Homeomorph α β) : f.symm.symm = f := rfl
```

Fails in Lean 3

Computed Fields

Many thanks to Gabriel Ebner

```
inductive Exp
  | var (i : Nat)
  | app (a b : Exp)
with
  @[computedField] hash : Exp → UInt64
    | .var i   => i.toUInt64
    | .app a b => mixHash a.hash b.hash
```

```
inductive Name where
  | anonymous : Name
  | str : Name → String → Name
  | num : Name → Nat → Name
with
  @[computedField] hash : Name → UInt64
    | .anonymous => 1723
    | .str p s   => mixHash p.hash s.hash
    | .num p v   => mixHash p.hash v.hash
```

Vector/Array notation

```
example (a : Array Int) (i : Nat) : Int :=
  a[i]

example (a : Array Int) (i : Nat) (h : i < a.size) : Int :=
  a[i]

example (a : Array Int) (i : Fin a.size) : Int :=
  a[i]

example (a : Array Int) (i : Nat) : Int :=
  a[i]!

example (a : Array Int) (i : Nat) : Option Int :=
  a[i]?

example (a : Array Int) (b : Array Int) (h : a.size ≤ b.size) (i : Fin a.size) : Int :=
  a[i] + b[i]

example (f : Nat → Array Int) (h₁ : ∀ n, n < (f n).size) (i j : Nat) (h₂ : j < i): Int :=
  have := Nat.lt_trans h₂ (h₁ i)  -- proof for j < (f i).size
  (f i)[j]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is perfomed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
a : Array Int
i : Nat
⊢ i < Array.size a
```

Delaborator: kernel terms back to syntax

```
@[appUnexpander Subtype] def unexpandSubtype : Lean.PrettyPrinter.Unexpander
  | `($(_) fun ($x:ident : $type) => $p)  => `({ $x : $type // $p })
  | `($(_) fun $x:ident => $p)            => `({ $x // $p })
  | _                                     => throw ()
```

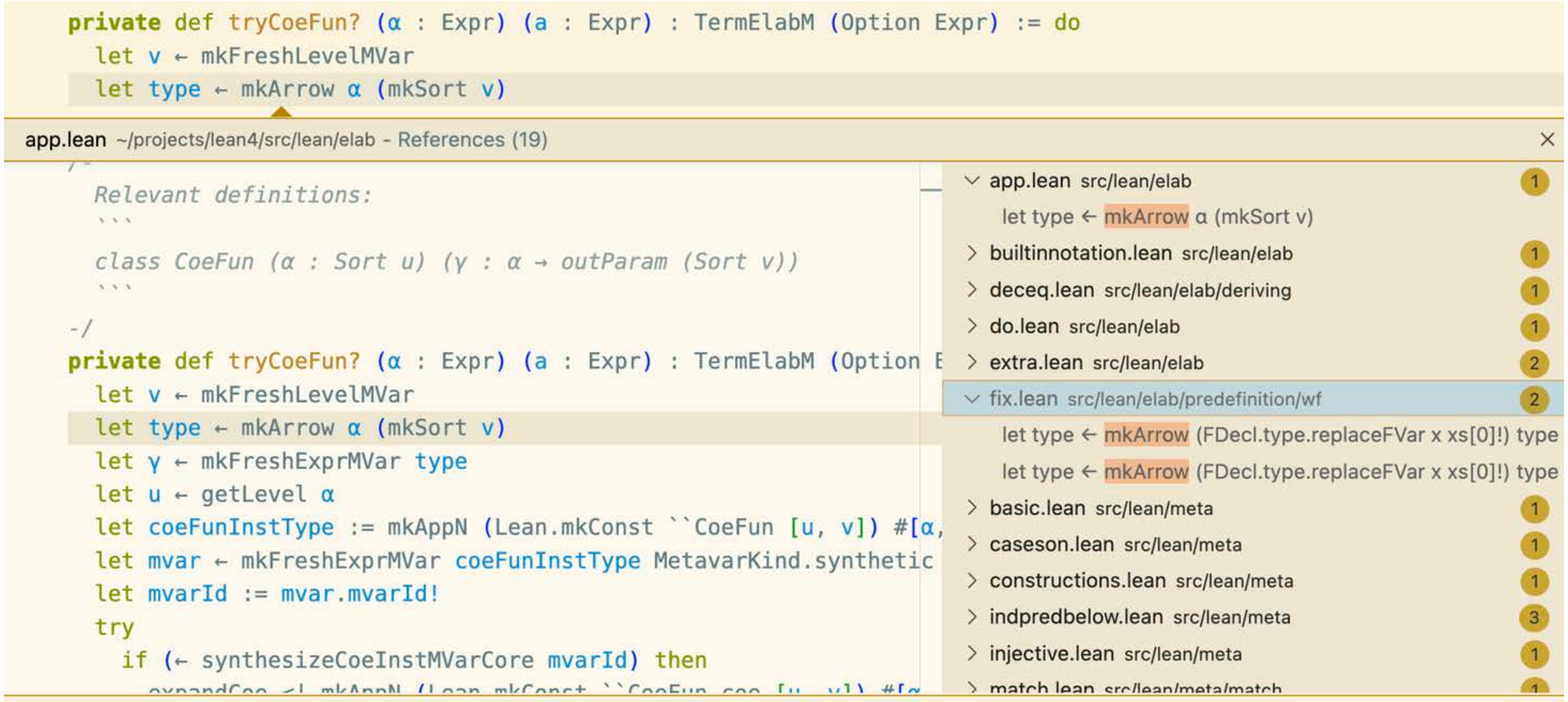
```
@[appUnexpander GetElem.getElem] def unexpandGetElem : Lean.PrettyPrinter.Unexpander
  | `(getElem $array $index $_) => `($array[$index])
  | _ => throw ()
```

```
@[builtinDelab app.dite]
def delabDIte : Delab := whenPPOption getPPNotation do
  -- Note: we keep this as a delaborator for now because it actually accesses the expression.
  guard $ (← getExpr).getAppNumArgs == 5
  let c ← withAppFn $ withAppFn $ withAppFn $ withAppArg delab
  let (t, h) ← withAppFn $ withAppArg $ delabBranch none
  let (e, _) ← withAppArg $ delabBranch h
  `(if $(mkIdent h):ident : $c then $t else $e)
where
  delabBranch (h? : Option Name) : DelabM (Syntax × Name) := do
    let e ← getExpr
    guard e.isLambda
    let h ← match h? with
      | some h => return (← withBindingBody h delab, h)
      | none   => withBindingBodyUnusedName fun h => do
        return (← delab, h.getId)
```

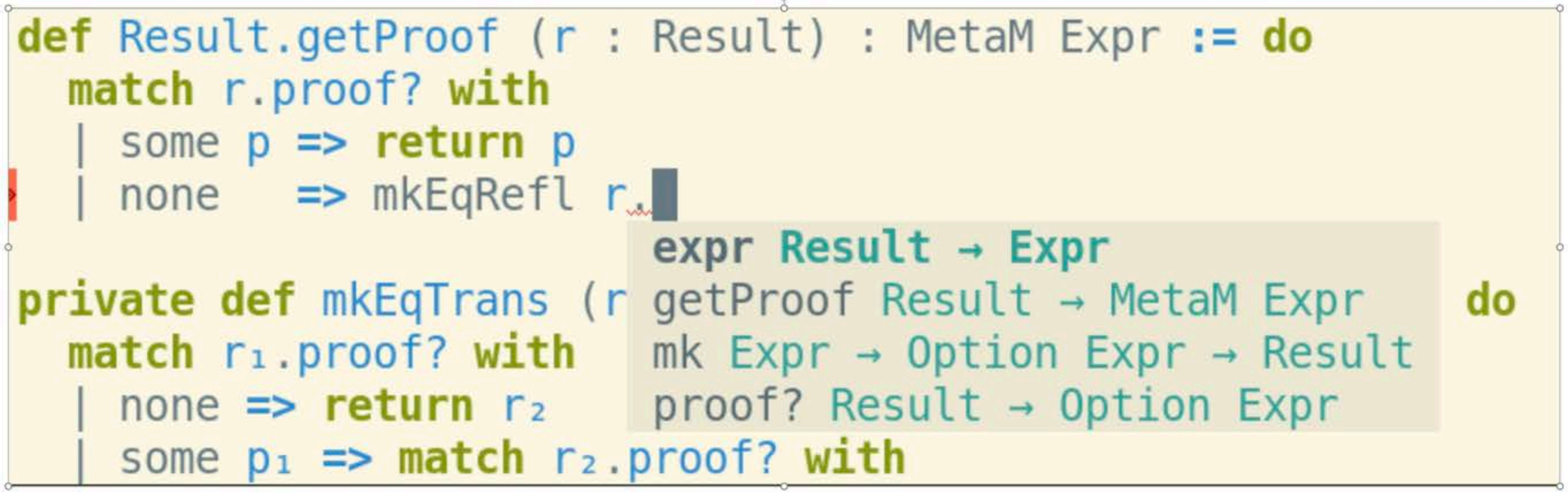
The Lean 4 LSP Server is feature complete

Big team effort: Marc Huisinga, Wojciech Nawrocki, Ed Ayers, Sebastian Ullrich, Gabriel Ebner, Lars König , Leo de Moura

The Lean 4 LSP Server is feature complete

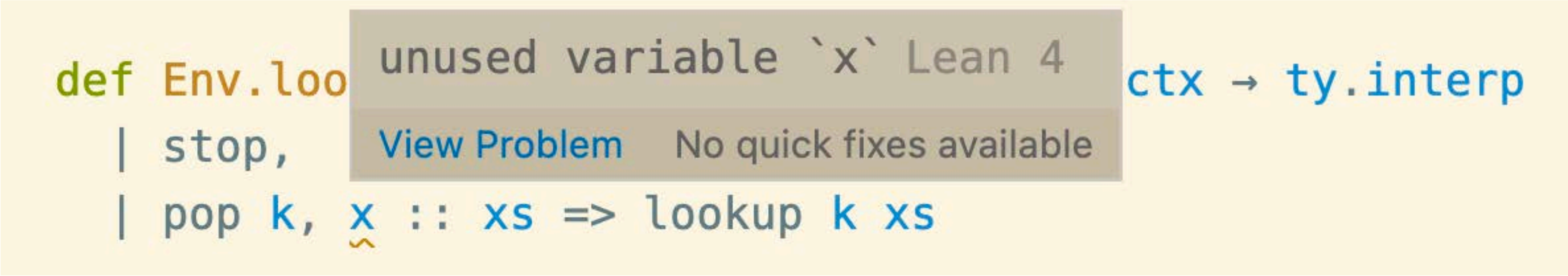


The Lean 4 LSP Server is feature complete



New feature: unused variable linter

Many thanks to Lars König



New LSP features coming soon …

Lean is becoming much more visual/interactive.

Many thanks to: Ed Ayers and Wojciech Nawrocki

tests > playground > widget > dynkin.lean > …

```
end Matrix

def cartanMatrix.E₈ : Matrix (Fin 8) (Fin 8) Int :=
  fun i j =>
    [[ 2,  0, -1,  0,  0,  0,  0,  0],
     [ 0,  2,  0, -1,  0,  0,  0,  0],
     [-1,  0,  2, -1,  0,  0,  0,  0],
     [ 0, -1, -1,  2, -1,  0,  0,  0],
     [ 0,  0,  0, -1,  2, -1,  0,  0],
     [ 0,  0,  0,  0, -1,  2, -1,  0],
     [ 0,  0,  0,  0,  0, -1,  2, -1],
     [ 0,  0,  0,  0,  0,  0, -1,  2]].get! i |>.get! j

instance : ToHtmlFormat (Matrix (Fin n) (Fin n) Int) where
  formatHtml M :=
    <div style="height: 100px; width: 300px; background: grey">
      {Html.element "svg" #[] (M.get_edges_html ++ Matrix.
      get_nodes_html n).toArray}
    </div>

#check cartanMatrix.E₈
```

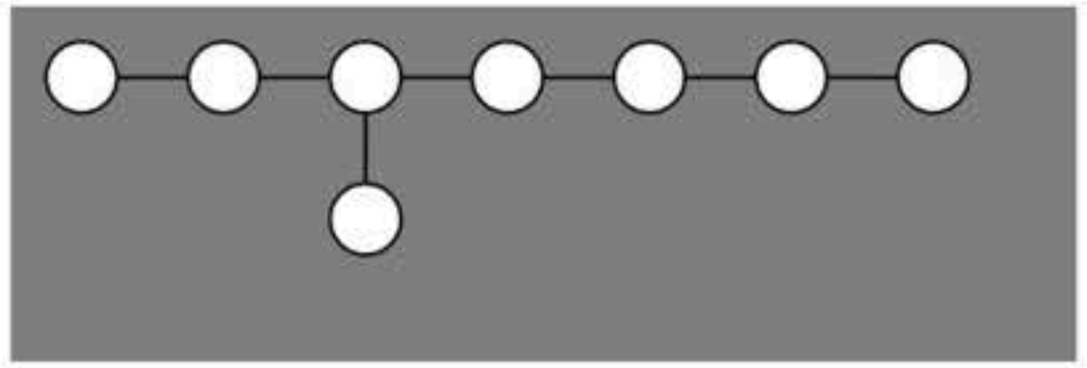
▼ dynkin.lean:87:22

▼ Messages (1)

▼ dynkin.lean:87:0

```
cartanMatrix.E₈ : Matrix (Fin 8) (Fin 8) Int
```

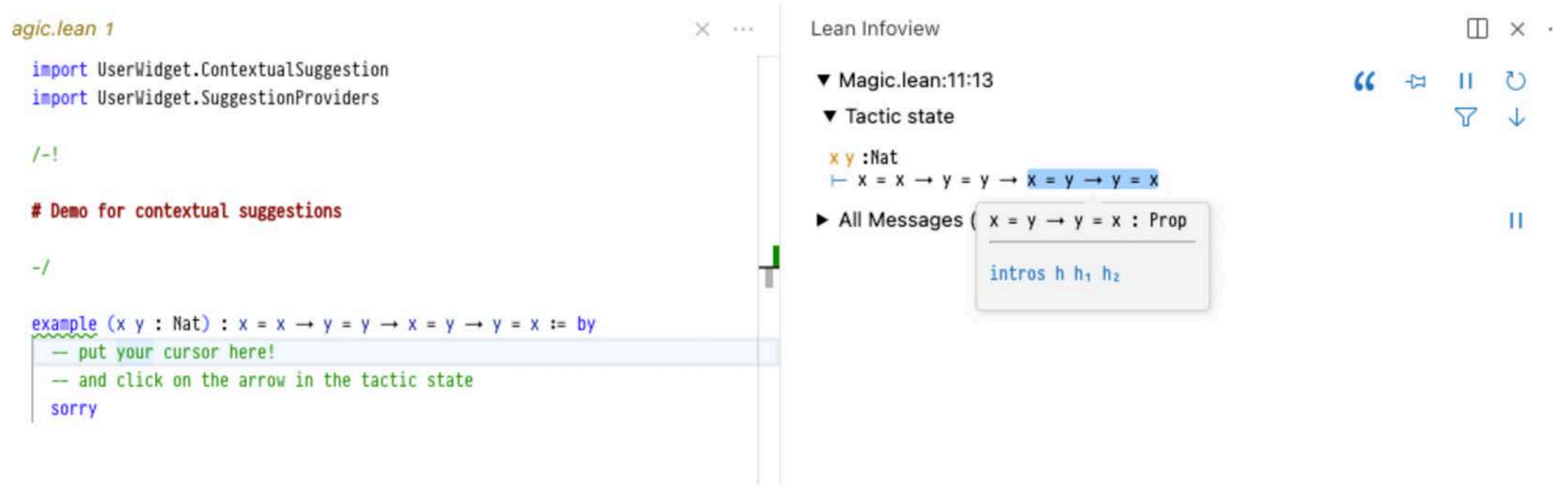
▼ Visualization



▶ Log

▶ All Messages (1)

New LSP features coming soon …



Lake = Lean + Make

Lake is the new Lean build system - https://github.com/leanprover/lake

By Lewis “Mac” Malone

Lake is extensible and implemented in Lean 4

```
import Lake
open Lake DSL System

package scilean
  -- defaultFacet := PackageFacet.staticLib
require mathlib from git
  "https://github.com/leanprover-community/mathlib4.git"@"8f609e0ed826dde127c8bc322cb6f91c5369d37a"

-- #check  LeanLibConfig
@[defaultTarget]
lean_lib SciLean {
  roots := #[`SciLean]
}

script tests (_args) do
  let cwd ← IO.currentDir
  -- let testDir := cwd / "test"
  let searchPath := SearchPath.toString
                     ["build" / "lib",
                      "lean_packages" / "mathlib" / "build" / "lib"]
```

Lake - precompiled extensions

Your Lean extensions are compiled to native machine code.

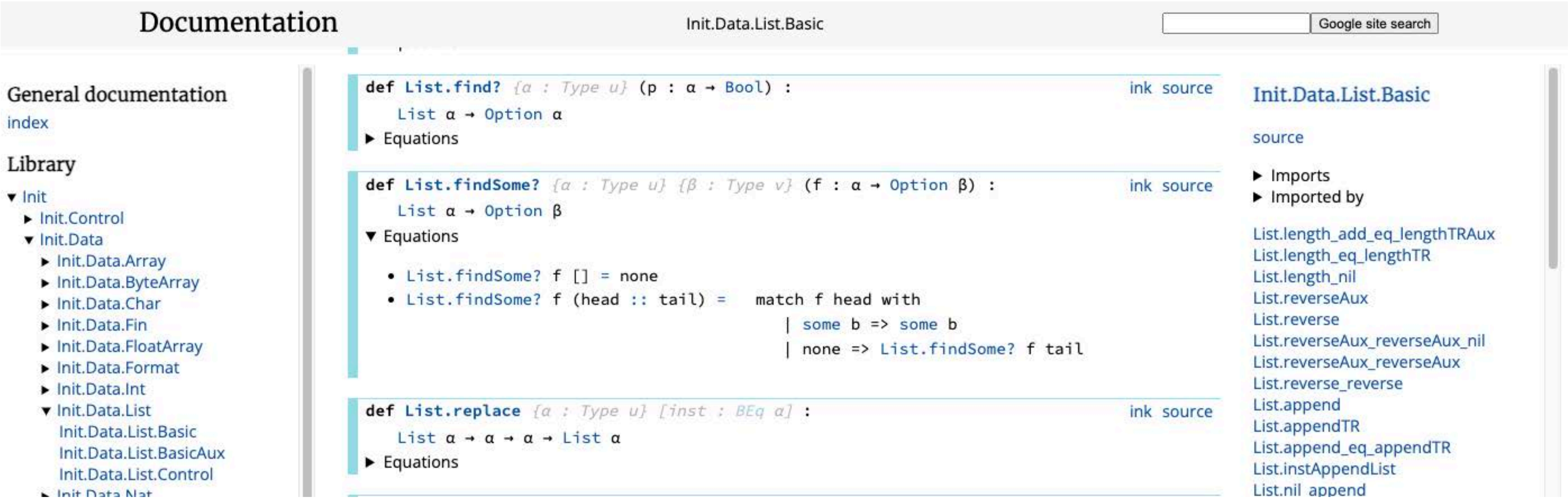
You can use “extern C” functions in your extensions.

```
import Lake

open Lake DSL

package aesop {
  precompileModules := true
}

@[defaultTarget]
lean_lib Aesop {}
```

```
import Lake
open Lake DSL

package AesopDemo {}

lean_lib AesopDemo {}

require aesop from git
  "https://github.com/JLimperg/aesop"@"1b02414e73e42808cebadea7fe594406dc589332"
```

doc-gen4: Documentation Generator for Lean 4

By Henrik Böving https://github.com/leanprover/doc-gen4



doc-gen4: Documentation Generator for Lean 4

```
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>
```

By Niklas Bülow

Literate programming for Lean 4.

Relies on the same infrastructure we use for the IDEs.

Future: Doc-gen4 + LeanInk integration

Lean Manual

We use the function `List.last` to prove the following theorem that says that if a list `as` is not empty, then removing the last element from `as` and appending it back is equal to `as`. We use the attribute `@[simp]` to instruct the `simp` tactic to use this theorem as a simplification rule.

```
@[simp] theorem List.dropLast_append_last (h : as ≠ []) : as.dropLast ++ [as.last h] = as
:= by
  match as with
  | [] => contradiction
  | [a] => simp_all [last, dropLast]
  | a₁ :: a₂ :: as =>

  α† : Type u_1   as† : List α†   a₁, a₂ : α†   as : List α†   h : a₁ :: a₂ :: as ≠ []
  ─────────────────────────────────────────────────────────────
  dropLast (a₁ :: a₂ :: as) ++ [last (a₁ :: a₂ :: as) h] = a₁ :: a₂ :: as

    simp [last, dropLast]
    exact dropLast_append_last (as := a₂ :: as) (by simp)
```

We now define the following auxiliary induction principle for lists using well-founded recursion on `as.length`. We can read it as follows, to prove `motive as`, it suffices to show that: (1) `motive []`; (2) `motive [a]` for any `a`; (3) if `motive as` holds, then `motive ([a] ++ as ++ [b])` also holds for any `a`, `b`, and `as`. Note that the structure of this induction principle is very similar to the `Palindrome` inductive predicate.

Cool projects using Lean 4

SciLean - Tomas Skrivan

Formalization: Gardam’s disproof of the Kaplansky Unit Conjecture -   Siddhartha Gadgil

Aesop - White Box Automation for Lean 4 - Jannis Limperg

Computational Law - Chris Bailey

Zero Knowledge Type Certificates - Yatima Inc.

CVC 5 / Lean 4 integration - Abdal Mohamed, Tomaz Mascarenhas, Haniel Barbosa, Cesare Tinelli

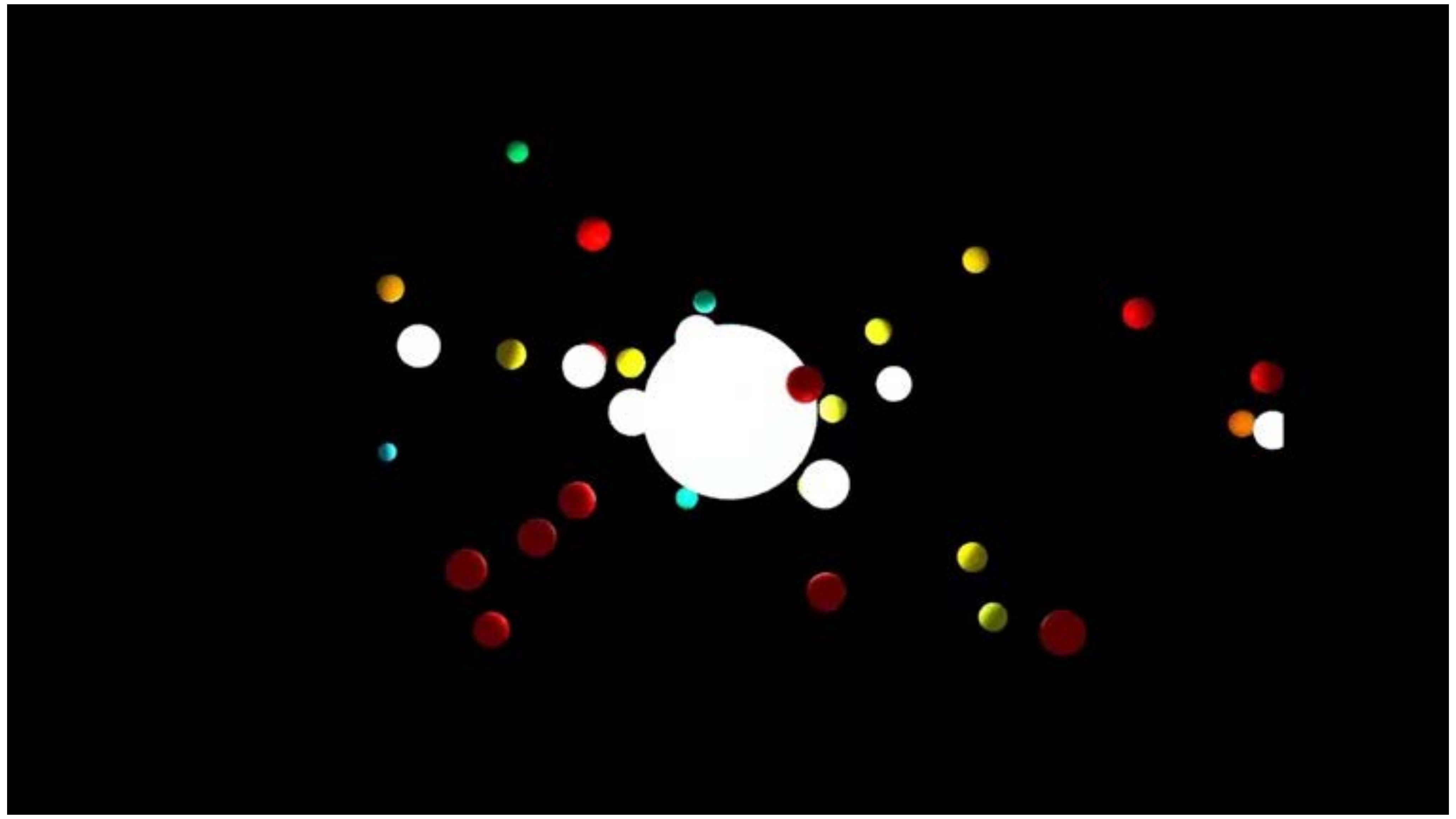
Papyrus - Lewis “Mac” Malone

SciLean

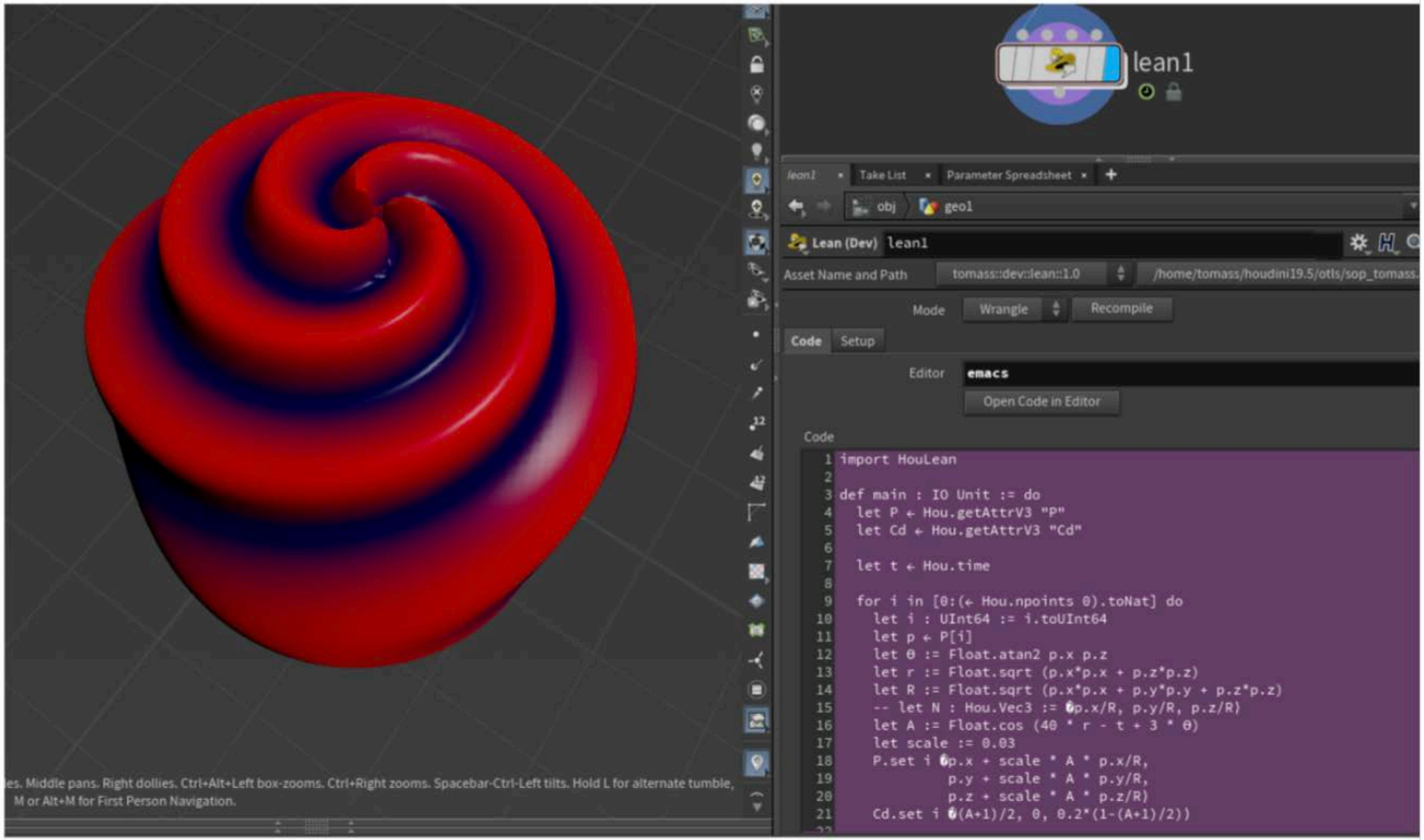
A framework for scientific computing based on Lean 4

https://github.com/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
```



SciLean - Houdini



Aesop

White box automation for Lean 4 - by Jannis Limperg

https://github.com/JLimperg/aesop

```
inductive Perm : List α → List α → Prop where
  | nil : Perm [] []
  | cons : Perm xs xs' → Perm (x :: xs) (x :: xs')
  | swap : Perm (x :: y :: xs) (y :: x :: xs)
  | trans : Perm xs ys → Perm ys zs → Perm xs zs

attribute [aesop safe] Perm.nil
attribute [aesop unsafe] Perm.cons
attribute [aesop unsafe] Perm.swap
attribute [aesop unsafe] Perm.trans

theorem Perm.symm : Perm xs ys → Perm ys xs := by
  intro h
  induction h <;> aesop

@[aesop safe]
theorem perm_insertInOrder {xs : List α} : Perm (x :: xs) (insertInOrder x xs) := by
  induction xs <;> aesop
```

Computational Law in Lean 4

Chris Bailey - Law Student - UIUC

Intern this summer at Microsoft Research

Mentors: Jonathan Protzenko and Leo de Moura

The Federal Rules of Civil Procedure

Overview

Procedural rules govern how a case or controversy may be adjudicated in civil court

Example: "party π must perform action α before time $\tau + n$, otherwise consequence κ"

In practice, the rules give rise to a high level of complexity

Federal courts have taken a hard-line approach to interpreting and applying procedural rules, ruling against litigants even when the court is in error (see *Bowles v. Russell*)

Litigants may forfeit important substantive rights, or simply lose outright

"Because the civil justice system directly touches everyone in contemporary American society [..] ineffective civil case management by state courts has an outsized effect on public trust and confidence compared to the criminal justice system" - NCSC civil justice report 2015

The Federal Rules of Civil Procedure

The Prevalence of Civil Legal Problems

Most low-income households have dealt with at least one civil legal problem in the past year – and many of these problems have had substantial impacts on people's lives.

3 in 4 (74%) low-income households experienced 1+ civil legal problems in the past year.

2 in 5 (39%) experienced 5+ problems and 1 in 5 (20%) experienced 10+ problems.

Most common types of problems: consumer issues, health care, housing, income maintenance.

1 in 2 (55%) low-income Americans who personally experienced a problem say these problems substantially impacted their lives – with the consequences affecting their finances, mental health, physical health and safety, and relationships.

Data source: 2021 Justice Gap Measurement Survey

The Federal Rules of Civil Procedure

Mission

Use Lean to produce a reliable library of functional components and a collection of relevant correctness proofs

Library components can be used by downstream consumers to implement a larger body software, both practical and analytical (case management software, web portals for courts, document generation, etc.)

There is an institutional appetite for the adoption of software in these roles, but a lack of sophistication in the tools has been cited as a major reason for lack of adoption in the large (see NCSC civil justice report 2015)

The Federal Rules of Civil Procedure

Goals

Level the playing field between teams of expert lawyers and everyone else

Prevent forfeiture of substantive rights by underrepresented or pro se litigants

Expand access to the courts; see more cases adjudicated on the merits rather than dismissed du
to procedural defects.

Help lawyers better serve clients by making fewer mistakes in less time

Improve matchmaking between those in need of legal services and service providers (more accurately place clients with clinical/pro bono resources)

Improve clarity in future revisions of procedural rules

Improve availability of labelled data for statistical analysis and ML/AI initiatives

The Federal Rules of Civil Procedure

Implementation

Layered architecture resembling a kernel/elaborator split, with a very simple model of computation.

A civil action and the procedural rules are encoded as a transition system $(S \times S_0 \times R)$

S as the type of all possible states, S_0 of valid initial states, and the transition relation $R : S \to S \to Prop$

With the procedural history acting viewed a sequence of steps and the procedural posture acting as state, a triple given triple is valid when $s \in S_0 \wedge \text{EvalR } c\ s\ s'$

A given procedural posture is reachable if it is in the reflexive transitive closure of R, starting at a valid initial state

The Federal Rules of Civil Procedure

Components:

Timelib

A general-purpose date and time library for the Lean ecosystem
(github.com/ammkrn/timelib)

UsCourts

An API for federal judicial districts and courts
(github.com/ammkrn/UsCourts)

JohnDoe

Through the pleading phase of the Federal Rules of Civil Procedure
(github.com/ammkrn/JohnDoe)

Yatima Inc.

Zero Knowledge Type Certificates

- Yatima IR: A content-addressed intermediate representation for Lean 4
- Lurk-Lang: A Lisp-like recursive zkSNARK language using microsoft/Nova
- By compiling a typechecker for Yatima IR to Lurk-Lang, we can produce zero-knowledge proofs of type correctness for Lean 4
- **Zero Knowledge Type Certificates are cryptographic proofs that a program validly typechecks, which can be verified in constant-time**

Conclusion

We implemented Lean 4 in Lean

Very extensible system: syntax, elaborators, delaborators, tactics, …

Compiler generates efficient code

User-extensions can be pre-compiled

We barely scratched the surface of the design space

The feedback on the milestone releases has been amazing, many new exciting applications.

Mathlib port is the next challenge