Lean 4 - an overview

an extensible programming language and theorem prover

Leonardo de Moura - Microsoft Research
Sebastian Ullrich - Karlsruhe Institute of Technology

January 4, 2021

Thanks

Daniel Selsam - type class resolution, feedback, design discussions

Marc Huisinga and Wojciech Nawrocki - Lean Server

Joe Hendrix, Andrew Kent, Rob Dockins, Simon Winwood (Galois Inc) - early adopters, suggestions, feedback

Daan Leijen, Simon Peyton Jones, Nikhil Swamy, Sebastian Graf, Max Wagner - design discussions, feedback, suggestions

How did we get here?

Previous project: Z3 SMT solver (aka push-button theorem prover)

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

LEAN 4 begins

Sebastian and I started Lean 4 in 2018

Lean in Lean

There is no specification!

Extensible programming language and theorem prover

A platform for trying new ideas in programming language and theorem prover design

A workbench for

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

“You can't **please** everybody, so you've got to **please yourself**.” George R.R. Martin

How we did it?

Lean is based on the Calculus of Inductive Constructions (CIC)

- All functions are total

We want

- General recursion
- Foreign functions
- Unsafe features (e.g., pointer equality)

The unsafe keyword

Unsafe functions may not terminate.

Unsafe functions may use (unsafe) type casting.

Regular (non unsafe) functions cannot call unsafe functions.

Theorems are regular (non unsafe) functions.

A compromise

Make sure you cannot prove False in Lean

- Theorems proved in Lean 4 may still be checked by reference checkers

Allow developers to provide an unsafe version for any (opaque) function whose type is inhabited

LOGICAL CONSISTENCY IS PRESERVED

- Primitives implemented in C

```
@[extern "lean_uint64_mix_hash"]
constant mixHash64 : UInt64 → UInt64 → UInt64
```

- Sealing unsafe features

```
@[inline] unsafe def withPtrEqUnsafe {α : Type u} (a b : α) (k : Unit → Bool) (h : a = b → k () = true) : Bool :=
  if ptrAddrUnsafe a == ptrAddrUnsafe b then true else k ()

@[implementedBy withPtrEqUnsafe]
def withPtrEq {α : Type u} (a b : α) (k : Unit → Bool) (h : a = b → k () = true) : Bool := k ()
```

LEAN4 in LEAN4

Lean 3 is interpreted and far from being a “full featured” programming language

Significant 2018 milestones

- Removed all unnecessary features
- New runtime and memory manager
- New compiler and intermediate representation
- Parsing engine prototype in Lean
  - core.lean in 56 secs, allocated > 200 million objects
  - two weeks later using code specializer: 5 secs (10x boost)

Leijen, Daan; Zorn, Benjamin; de Moura, Leonardo (2019). "Mimalloc: Free List Sharding in Action"

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 compiler is not a transpiler!

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 Parser

beginning 2019: core.lean in 20ms

- Using new compiler
- New design that takes advantage of FBIP

```
structure ParserState where
  stxStack : Array Syntax := #[]
  pos      : String.Pos := 0
  cache    : ParserCache
  errorMsg : Option Error := none
```

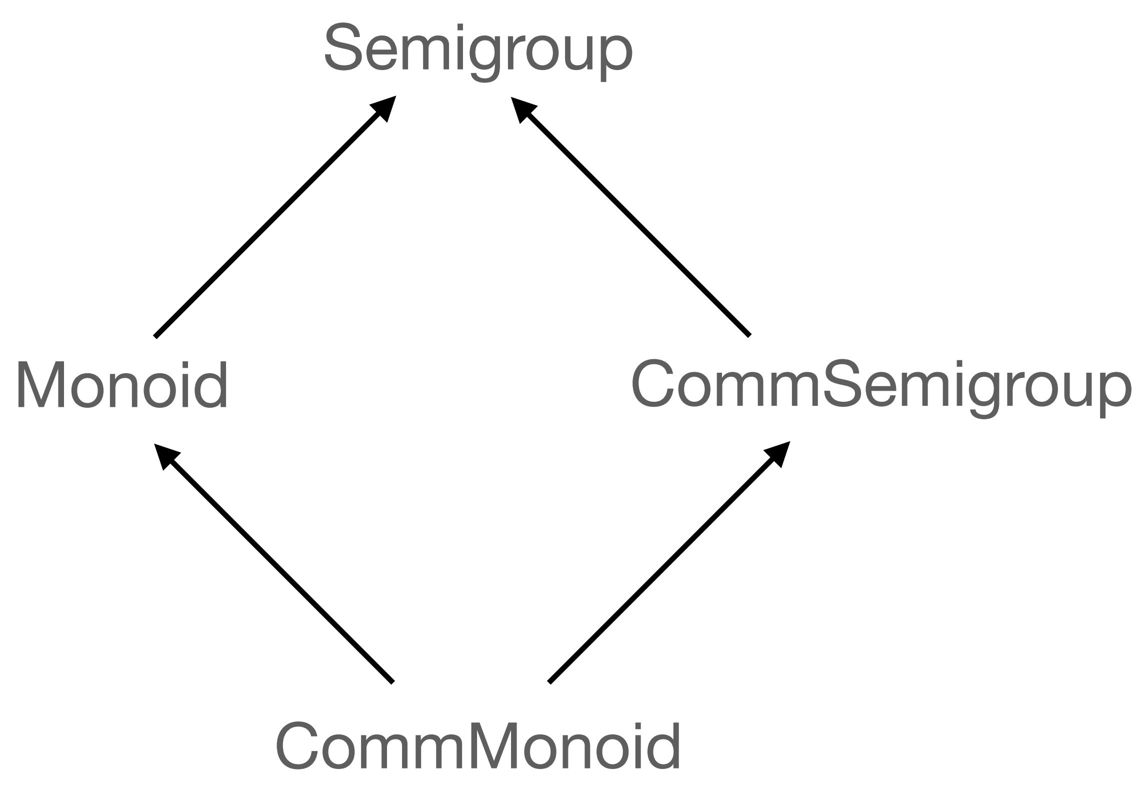
```
def pushSyntax (s : ParserState) (n : Syntax) : ParserState :=
  { s with stxStack := s.stxStack.push n }

def popSyntax (s : ParserState) : ParserState :=
  { s with stxStack := s.stxStack.pop }

def shrinkStack (s : ParserState) (iniStackSz : Nat) : ParserState :=
  { s with stxStack := s.stxStack.shrink iniStackSz }

def next (s : ParserState) (input : String) (pos : Nat) : ParserState
  { s with pos := input.next pos }
```

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

LEAN 4 extends

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

Lean 4 (and Lean 3 new structure) command produce a more efficient representation

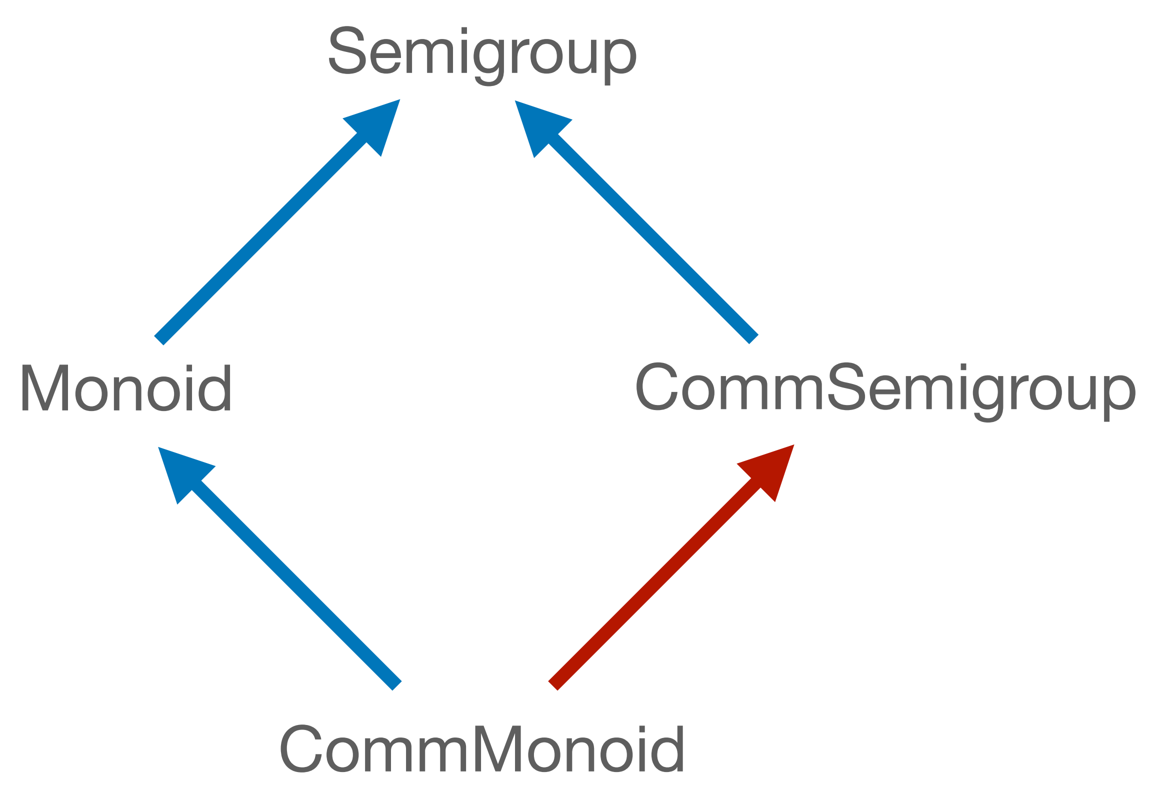
```
class Semigroup (α : Type u) extends Mul α where
  mul_assoc (a b c : α) : a * b * c = a * (b * c)
```

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

```
class 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 where
  mul_comm (a b : α) : a * b = b * a
```

```
instance [CommMonoid α] : CommSemigroup α where
  mul_comm := CommMonoid.mul_comm
```



You can automate the generation of the last command if you want
Note that is better than naive flattening as it is done in the old_structure_cmd

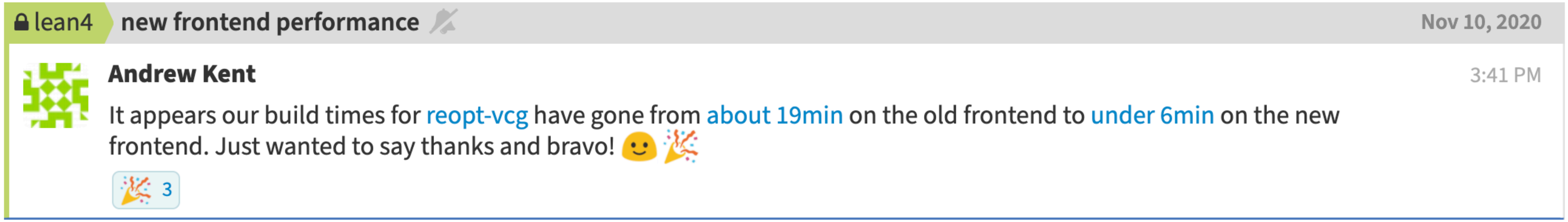
Lean 4 Elaborator

Elaborator (and auxiliary modules) were developed in 2020

- tactic framework, dependent pattern matching, structural recursion

Deleted the old frontend (implemented in C++) last October

Galois Inc finished converting their tool to the new frontend in November 10



We rarely write C/C++ code anymore, all Lean development is done in Lean itself

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))
```

Hygienic macro system

We have many different syntax categories.

```
syntax   stx "+" : stx
syntax   stx "*" : stx
syntax   stx "?" : stx
syntax:2 stx " <|> " 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))
```

You can define your own categories too.

```
-- 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
```

LEAN 4 Hygienic macro system

Your macros can be recursive.

```
syntax "[" term,* "]"  : term
syntax "%[" term,* "|" term "]" : term
```

```
macro_rules
  | `(%[ $[$x],* | $k ]) =>
    if x.size < 8 then
      x.foldrM (init := k) fun x k =>
        `(List.cons $x $k)
    else
      let m := x.size / 2
      let y := x[m:]
      let z := x[:m]
      `(let y := %[ $[$y],* | $k ]
        %[ $[$z],* | y ])
```

Hygiene guarantees that there is no accidental capture here

LEAN 4 Hygienic macro system

Many Lean 3 tactics are just macros

```
syntax "funext " term:max+ : tactic

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

```
theorem ex : (fun (x : Nat × Nat) (y : Nat × Nat) => x.1 + y.2)
             =
             (fun (x : Nat × Nat) (z : Nat × Nat) => z.2 + x.1) := by
  funext (a, b) (c, d)
  show a + d = d + a
  rw [Nat.addComm]
```

LEAN 4 Hygienic macro system

There is no builtin `begin … end` tactic block in Lean 4, is this a problem?

```
macro "begin " ts:tactic,* "end"%i : term => `(by { $[$ts]* }%$i)

theorem ex1 (x : Nat) : x + 0 = 0 + x :=
  begin
    rw Nat.zeroAdd,
    rw Nat.addZero
  end
```

LEAN 4 Hygienic macro system

There is no builtin `begin … end` tactic block in Lean 4, is this a problem?

```
macro "begin " ts:tactic,* "end"%i : term => `(by { $[$ts]* }%$i)

theorem ex1 (x : Nat) : x + 0 = 0 + x :=
  begin
    rw Nat.zeroAdd,
    rw Nat.addZero
  end
```

What about my dangling commas? No problem

```
macro "begin " ts:tactic,*,? "end"%i : term => `(by { $[$ts]* }%$i)

theorem ex1 (x : Nat) : x + 0 = 0 + x :=
  begin
    rw Nat.zeroAdd,
    rw Nat.addZero,
  end
```

LEAN 4 Hygienic macro system

I want to use main stream function application notation: `f(a, b, c)`

```
def f (x y : Nat) := x + 2*y

syntax term noWs "(" term,* ")" : term

macro_rules
  | `($f($args,*)) => `($f $args*)

#check f(1, 2)
```

LEAN 4 String interpolation

```
-- Assume `y = 5`
let x := y + 1
IO.println s!"x: {x}, y: {y}"
-- x: 6, y: 5
```

```
"x: " ++ toString x ++ ", y: " ++ toString y
```

Started as a Lean example

```
partial def interpolatedStrFn (p : ParserFn) : ParserFn := fun c s =>
  let input     := c.input
  let stackSize := s.stackSize
  let rec parse (startPos : Nat) (c : ParserContext) (s : ParserState) : ParserState :=
    let i     := s.pos
    if input.atEnd i then
      s.mkEOIError
    else
      let curr := input.get i
      let s    := s.setPos (input.next i)
      if curr == '\"' then
        let s := mkNodeToken interpolatedStrLitKind startPos c s
        s.mkNode interpolatedStrKind stackSize
```

...

String interpolation

```
syntax "throwError! " ((interpolatedStr term) <|> term) : term

macro_rules
  | `(throwError! $msg) =>
    if msg.getKind == interpolatedStrKind then
      `(throwError (msg! $msg))
    else
      `(throwError $msg)
```

```
syntax:max "msg!" (interpolatedStr term) : term

macro_rules
  | `(msg! $interpStr) => do
    let chunks := interpStr.getArgs
    let r ← Lean.Syntax.expandInterpolatedStrChunks chunks (fun a b => `($a ++ $b)) (fun a => `(toMessageData $a))
    `(($r : MessageData))
```

```
unless targetsNew.size == targets.size do
  throwError! "invalid number of targets #{targets.size}, motive expects #{targetsNew.size}"
```

LEAN 4 do notation

Introduced by the Haskell programming language

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

↓

```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

Lean version is a DSL with many improvements

- Nested actions
- Rust-like reassignments and “return”
- Iterators + “break/continue”

It could have been implemented by users

LEAN 4 do notation

```
def sum (xs : Array Nat) : IO Nat := do
  let mut s := 0
  for x in xs do
    IO.println s!"x: {x}"
    s := s + x
  return s

def contains (k : Nat) (pairs : Array (Nat × Nat)) : IO (Option Nat) := do
  for (x, y) in pairs do
    if k == x then
      IO.println s!"found key {k}"
      return y
  return none
```

```
def processResult (alts : Array Syntax) (result : Array Meta.InductionSubgoal) (numToIntro : Nat := 0) : TacticM Unit := do
  if alts.isEmpty then
    setGoals <| result.toList.map fun s => s.mvarId
  else
    unless alts.size == result.size do
      throwError! "mismatch on the number of subgoals produced ({result.size}) and alternatives provided ({alts.size})"
    let mut gs := #[]
    for i in [:result.size] do
      let subgoal    := result[i]
      let mut mvarId := subgoal.mvarId
      if numToIntro > 0 then
        (_, mvarId) ← introNP mvarId numToIntro
      gs ← evalAlt mvarId alts[i] gs
    setGoals gs.toList
```

LEAN 4 do notation

```
abbrev M := StateT Nat (ExceptT String Id)

def withdraw (v : Nat) : M Unit := do
  let s ← get
  if v > s then
    throw "not enough funds"
  modify (fun s => s - v)

def withdraw' (v : Nat) : M Unit := do
  if v > (← get) then
    throw "not enough funds"
  modify (· - v)

abbrev N := ReaderT Nat M

def deposit (v : Nat) : N Unit := do
  if v + (← get) > (← read) then
    throw s!"exceeded maximum allowed {← read}"
  modify (· + v)

def test (v w : Nat) : N Unit := do
  deposit v
  withdraw w
```

LEAN 4 Structured (and hygienic) tactic language

```
def Nat.ltWf : WellFounded Nat.lt := by
  apply WellFounded.intro
  intro n
  induction n with
  | zero      =>
    apply Acc.intro 0
    intro _ h
    apply absurd h (Nat.notLtZero _)
  | succ n ih =>
    apply Acc.intro (Nat.succ n)
    intro m h
    have m = n ∨ m < n from Nat.eqOrLtOfLe (Nat.leOfSuccLeSucc 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        =>
    apply False.elim
    apply h rfl
  | [x], h        => rfl
  | x₁::x₂::xs, h =>
    have x₂::xs ≠ [] by intro h; injection h
    have ih := concatEq (x₂::xs) this
    show x₁ :: concat (dropLast (x₂::xs)) (last (x₂::xs) this) = x₁ :: x₂ :: xs
    rewrite ih
    rfl
```

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 :=
```

```
theorem modLt (x : Nat) {y : Nat} (h : y > 0) : x % y < y := by
  induction x, y using Nat.mod.inductionOn generalizing h with
  | ind x y h₁ ih =>
    rw [Nat.modEqSubMod h₁.2]
    exact ih h
  | base x y h₁ =>
    match Iff.mp (Decidable.notAndIffOrNot ..) h₁ with
    | Or.inl h₁ => exact absurd h h₁
    | Or.inr h₁ =>
      have hgt := Nat.gtOfNotLe h₁
      have heq := Nat.modEqOfLt hgt
      rw [← heq] at hgt
      assumption
```

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 hygienicIntro false in
theorem ex1 {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`.
```

```
theorem ex2 {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"
```

```
theorem ex3 {a p q r : Prop} : p → (p → q) → (q → r) → r := by
  intro _ h1 h2
  apply h2
  apply h1
  assumption
```

LEAN 4 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
  | [],     _    => []
  | a::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
  | [],     _    => []
  | a::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
  | [],      _    => []
  | a::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} →
(motive : List α → Nat → Sort v) →
-- discriminants
(as : List α) →
(i : Nat) →
-- alternatives
((x : Nat) → motive [] x) →
((a : α) → (as : List α) → motive (a :: 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
  | [],     _    => []
  | a::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 List.Foo.eraseIdx.{u} : {α : Type u} → List α → Nat → List α :=
fun {α : Type u} (as : List α) (i : Nat) =>
  List.brecOn as
    (fun (t : List α) (f : List.below t) (i_1 : Nat) =>
      (match t, i_1 with
        | [], x => fun (x_1 : List.below []) => []
        | a :: as_1, 0 => fun (x : List.below (a :: as_1)) => as_1
        | a :: as_1, Nat.succ n => fun (x : List.below (a :: as_1)) => a :: PProd.fst x.fst n)
        f)
    i
```

LEAN 4 match … with

Information about named patterns and inaccessible terms is preserved

```
inductive Imf {α : Type u} {β : Type v} (f : α → β) : β → Type (max u v)
| mk : (a : α) → Imf f (f a)

def h {α β} {f : α → β} : {b : β} → Imf f b → α
| _, Imf.mk a => a
```

```
def h.{u_1, u_2} : {α : Type u_1} → {β : Type u_2} → {f : α → β} → {b : β} → Imf f b → α :=
fun {α : Type u_1} {β : Type u_2} {f : α → β} (x : β) (x_1 : Imf f x) =>
  match x, x_1 with
  | .(f a), Imf.mk a => a
```

```
def f : List Nat → List Nat
  | a::xs@(b::bs) => xs
  | _             => []
```

```
def f : List Nat → List Nat :=
fun (x : List Nat) =>
  match x with
  | a :: xs@(b :: bs) => xs
  | x_1 => []
```

pp of the kernel term

LEAN 4 match … with

Equality proofs (similar to if-then-else)

```
theorem ex (a : Bool) (p q : Prop) (h₁ : a = true → p) (h₂ : a = false → q) : p ∨ q :=
  match h:a with
  | true  => Or.inl (h₁ h)
  | false => Or.inr (h₂ h)

def head {α} (xs : List α) (h : xs ≠ []) : α :=
  match he:xs with
  | []   => absurd he h
  | x::_ => x
```

LEAN 4 match … with

Lean 3 bugs in the dependent pattern matcher have been fixed

Daniel was the first to report the bug, and it was “rediscovered” many times

```
inductive Op : Nat → Nat → Type where
  | mk : ∀ n, Op n n

structure Node : Type where
  id₁ : Nat
  id₂ : Nat
  o   : Op id₁ id₂

def h (x : List Node) : Bool :=
  match x with
  | _ :: Node.mk _ _ (Op.mk 0) :: _  => true
  | _                                => false
```

```
inductive Foo : Bool → Type where
  | bar : Foo false
  | baz : Foo false

def g {b : Bool} (x : Foo b) : Bool :=
  match b, x with
  | _, Foo.bar => true
  | _, _       => false
```

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 (todo)

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

```
def List.Foo.eraseIdx.{u} : {α : Type u} → List α → Nat → List α :=
fun {α : Type u} (as : List α) (i : Nat) =>
  List.brecOn as
    (fun (t : List α) (f : List.below t) (i_1 : Nat) =>
      (match t, i_1 with
        | [], x => fun (x_1 : List.below []) => []
        | a :: as_1, 0 => fun (x : List.below (a :: as_1)) => as_1
        | a :: as_1, Nat.succ n => fun (x : List.below (a :: as_1)) => 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 let rec in theorems

```
theorem Tree.acyclic (x t : Tree) : x = t → x ≮ t := by
  let rec right (x s : Tree) (b : Tree) (h : x ≮ b) : node s x ≠ b ∧ node s x ≮ b := by
    match b, h with
    | leaf,     h =>
      apply And.intro _ trivial
      intro h; injection h
    | node l r, h =>
      have ihl : x ≮ l → node s x ≠ l ∧ node s x ≮ l from right x s l
      have ihr : x ≮ r → node s x ≠ r ∧ node s x ≮ r from right x s r
      have hl : x ≠ l ∧ x ≮ l from h.1
      have hr : x ≠ r ∧ x ≮ r from h.2.1
```

...

```
  let rec aux : (x : Tree) → x ≮ x
    | leaf     => trivial
    | node l r => by
        have ih₁ : l ≮ l from aux l
        have ih₂ : r ≮ r from aux r
        show (node l r ≠ l ∧ node l r ≮ l) ∧ (node l r ≠ r ∧ node l r ≮ r) ∧ True
        apply And.intro
        focus
          apply left
          assumption
        apply And.intro _ trivial
        focus
          apply right
          assumption
  intro h
  subst h
  apply aux
```

LEAN 4 Haskell-like "where" clause

Expands into `let rec`

```
private def toKey (n : Name) : List NamePart :=
  loop n []
where
  loop
    | Name.str p s _, parts    => loop p (NamePart.str s :: parts)
    | Name.num p n _, parts    => loop p (NamePart.num n :: parts)
    | Name.anonymous, parts => parts
```

```
def h : Nat -> Nat
  | 0   => g 0
  | x+1 => g (h x)
where
  g x := x + 1
```

LEAN 4 Elaborator: named arguments

Named arguments enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list

```
def sum (xs : List Nat) :=
  xs.foldl (init := 0) (fun s x => s + x)

example {a b : Nat} {p : Nat → Nat → Nat → Prop} (h₁ : p a b b) (h₂ : b = a)
    : p a a b :=
  Eq.subst (motive := fun x => p a x b) h₂ h₁

def sumOdd (xs : List Nat) :=
  xs.foldl (init := 0) fun s x =>
    if x % 2 == 1 then s + x else s
```

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 ·)
```

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: first attempt

```
class HAdd (α : Type u) (β : Type v) (γ : outParam (Type w)) where
  hAdd : α → β → γ

infixl:65 (priority := high) "+" => HAdd.hAdd

instance : HAdd Nat Nat Nat where
  hAdd := Nat.add

variable (x : Nat)

#check fun y => x + y -- Error: we can't synthesize HAdd instance
```

LEAN 4 Default instances: the missing feature

```
@[defaultInstance]
instance [Add α] : HAdd α α α where
  hAdd := Add.add

variable (x : Nat)

#check fun y => x + y
```

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]
```

```
def ex1 (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

def ex2 (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 NatOp

  scoped infixl:65 (priority := high) "+" => Nat.add
  scoped infixl:70 (priority := high) "*" => Nat.mul
  -- ...

end NatOp

variables (n : Nat) (i : Int)
#check n + i
#check i + n
-- We are still using the builtin heterogeneous +
open NatOp -- activate notation in the NatOp namespace
#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 StateT.pure [Monad m] (a : α) : StateT σ m α :=
  fun s => pure (a, s)

protected def StateT.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
```

LEAN 4 Unification hints

```
structure Magma.{u} where
  α   : Type u
  mul : α → α → α

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

def mul {s : Magma} (a b : s) : s :=
  s.mul a b

infixl:70 (priority := high) "*" => mul

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

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

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

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

def f (x y : Nat) : Nat × Nat × Nat :=
  (x, y, y-1) * (x, y, y+1)

#eval f 2 10
-- (4, 100, 99)
```

What about the kernel?

Same design philosophy:
Minimalism, no termination checker in the kernel, external type checkers

You can write your own type checker if you want

Foundations: the Calculus of Inductive Constructions (CIC)

No inconsistency has ever been reported to a Lean developer

Lean 4 kernel is actually smaller than Lean 3

Kernel changes

Support for mutual inductive types (Lean 2 supported them) and nested inductives

Mutual inductive types are well understood (Dybjer 1997)

Nested inductives can be mapped into mutual, but very convenient in practice

```
inductive Expr where
  | var : Nat → Expr
  | app : List Expr → Expr
```

```
mutual

  inductive Expr where
  | var : Nat → Expr
  | app : ListExpr → Expr

  inductive ListExpr where
  | nil  : ListExpr
  | cons : Expr → ListExpr → ListExpr

end
```

The kernel checks them by performing the expansion above

Kernel changes

Support for reducing Nat operations efficiently in the kernel

It only impacts performance

It is easy to support them in external type checkers

For additional details https://leanprover.github.io/lean4/doc/nat.html

```
theorem ex
      : 1000000000000000 * 2000000000000000000 = 2000000000000000000000000000000000 :=
  rfl
```

```
def BV (n : Nat) : Type := ...
```

```
def concat (x : BV n) (y : BV m) : BV (n+m) := ...
```

```
def f (x : BV 512) (y : BV 1024) : BV 2048 :=
  concat x (concat x y)
```

Kernel changes

No inconsistency has ever been reported to a Lean developer

If an inconsistency is found in the future, it will be tagged as a high priority bug

ITPs are still not widely used, soundness is not the issue

There are roughly two kinds of bugs in ITPs: conceptual and programming mistakes

Programming mistakes are easy to fix

Conceptual bugs are often much harder to fix

Lean minimalism is our defense against conceptual bugs

“During wartime, you don't study the mating rituals of butterflies”

Lean 4 Documentation

The Lean manual is available at https://leanprover.github.io/lean4/doc/

It is still working in progress

Focus is “Lean as a programming language”

Lean Manual

What is Lean

Lean is a functional programming language that makes it easy to write correct and maintainable code. You can also use Lean as an interactive theorem prover.

Lean programming primarily involves defining types and functions. This allows your focus to remain on the problem domain and manipulating its data, rather than the details of programming.

```
-- Defines a function that takes a name and produces a greeting.
def getGreeting (name : String) := s!"Hello, {name}! Isn't Lean great?"
```

Next steps

Well-founded recursion, auto-generated induction principles

UI feature parity with Lean 3: goal view, go to definition, and basic autocompletion

Missing tactics and decision procedures

Diagnostic tools (e.g., user-friendly traces)

Typed syntax quotations

Lean compiler in Lean

Interactive compilation DSL (conv for code generation)

User-defined #lang extensions (Racket)

Cleaning leftovers from old frontend

Testing and leanchecker tool

leanpkg polishing

How can I contribute?

Experiments, experiments, experiments, …

Try Lean 4 and isolate issues

Isolate Lean 3 issues and report to us

Example: Reid1.lean

```
structure constantFunction (α β : Type) :=
(f : α → β)
(h : ∀ a₁ a₂, f a₁ = f a₂)

instance {α β : Type} : has_coe_to_fun (constantFunction α β) :=
⟨_, constantFunction.f⟩

def g {α : Type} : constantFunction α (option α) :=
{ f := fun a, none,
  h := fun a₁ a₂, rfl }

#check g 3
#check @g nat 3
```

```
structure ConstantFunction (α β : Type) where
  f : α → β
  h : ∀ a₁ a₂, f a₁ = f a₂

instance : CoeFun (ConstantFunction α β) (fun _ => α → β) where
  coe c := c.f

def g {α : Type} : ConstantFunction α (Option α) where
  f a := none
  h a₁ a₂ := rfl

#check g 3
#check g (α := Nat) 3
#check @g Nat 3
```

Experiments

- Crafted benchmarks that reflect performance problems in mathlib
  - Learn to profile
- Heterogeneous vs homogeneous operators
- Bundled vs unbundled structures
  - Lean 4 unification hints are much more robust than the ones available in Lean 3
- Happy to reserve 1-2 hours per week to discuss issues using Zoom

“I need spiritual warriors” Alejandro Jodorowsky

I want to contribute to the Lean code base

Different cultural backgrounds CS vs Math

Happy to collaborate and listen, but time is finite

Many unsuccessful attempts in the past

Funny

- “The inquisitor” asks a bunch of questions but doesn’t do anything
- “The dreamer” has big ideas, but never delivers anything
- “The socializer” wants to have fun, tell jokes, discuss wild ideas
- “The clueless” requires a lot of attention, and can’t figure out anything
- “The over confident” knows it all, although never built anything

Lean 4 is very extensible, you can customize it without modifying the main repository

“Programming is only fun, when the program doesn’t have to work” Mafé

Example of successful contribution

Andrew Kent (Galois Inc) wanted a better `for .. in`, traverse multiple structures in parallel

```
def f (xs : Array Nat) (ys : List (Nat × Nat)) : IO Unit :=
  for x in xs, (y₁, y₂) in ys do
    IO.println s!"{x} {y₁ + y₂}"
```

Approaches used in Rust and Racket create technical difficulties (e.g., termination)

Andrew prototypes a hybrid encoding where we have

- Main traversal which guarantees termination
- Auxiliary streams a-la Racket

We integrate Andrew’s idea at Do.lean

Andrew Kent

Coming back after a weekend and seeing it already pushed is a wonderful surprise - it looks great! 😁 I'm going to have to go brag to my kids I got one of my presents early this year 😆

🎁 3

Thoughts on mathlib conversion

- Play with Lean 4 before trying any serious conversion effort
- Try feasibility experiments
- Will Lean 4 keep changing?
  - There is no spec for Lean, we are trying new ideas
  - some features will be modified/removed
- Suggestion (take it with a grain of salt)
  - Modify Lean 3 to export notation, class instances, and other mathlib relevant metadata
  - Write a tool for importing these data in Lean 4
  - Setup your build system to allow Lean 3 and Lean 4 files to coexist in the same project
  - Use Lean 4 for writing new files, and convert old ones on demand

Projects on our radar

Custom automation for the IMO grand challenge (Daniel Selsam)

Optimizing tensor computations and HPC (Olli Saarikivi)

SAT/SMT solver integration

Rust integration

DSLs on top of Lean, example: model checker

Conclusion

We implemented Lean4 in Lean

Very extensible system

Sealing unsafe features. Logical consistency is preserved

Compiler generates C code. Allows users to mix compiled and interpreted code

It is feasible to implement functional languages using RC

We barely scratched the surface of the design space

Source code available online. http://github.com/leanprover/lean4