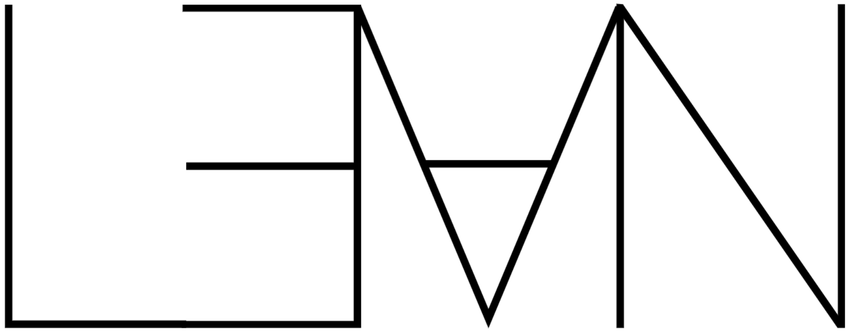
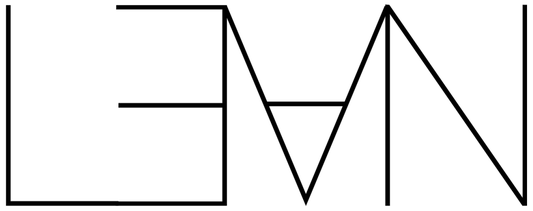


Lean 4

an extensible programming language and theorem prover

Leonardo de Moura - November 5, 2020



- Goals
  - Extensibility, Expressivity, Scalability, Proof stability
  - Functional Programming (efficiency)
- Platform for
  - Developing custom automation and domain specific languages (DSLs)
  - Software verification
  - Formalized Mathematics
- Dependent Type Theory
- de Bruijn’s principle: small trusted kernel, external proof/type checkers

Extensibility

Lean 3 users extend Lean using Lean

Examples:

- Ring Solver, Coinductive predicates, Transfer tactic,
- Superposition prover, Linters,
- Fourier-Motzkin & Omega,
- Many more

- Access Lean internals using Lean
  - Type inference, Unifier, Simplifier, Decision procedures,
  - Type class resolution, …

Applications

IMO Grand Challenge Daniel Selsam, MSR

Tom Hales, University of Pittsburgh

Jesse Michael Han, Floris Van Doorn

VU Amsterdam

Other applications

- Certigrad, Daniel Selsam, Stanford
- IVy metatheory, Ken McMillan, MSR Redmond
- AliveInLean, Nuno Lopes, MSR Cambridge
- Protocol Verification, Galois Inc
- SQL Query Verification, Univ. Washington
- Education
  - Introduction to Logic (CMU), Type theory (CMU), Introduction to Proof (Imperial College),
  - Software verification and Logic (VU Amsterdam)
  - Programming Languages (UW)

Online resources

Theorem Proving in Lean

1. Introduction
2. Dependent Type Theory
3. Propositions and Proofs
4. Quantifiers and Equality
5. Tactics
6. Interacting with Lean
7. Inductive Types
8. Induction and Recursion
9. Structures and Records
10. Type Classes
11. Axioms and Computation

PDF version
Lean Home

Quick search

Go

Theorem Proving in Lean

Anne Baanen
Alexander Bentkamp
Jasmin Blanchette
Johannes Hölzl
Jannis Limperg

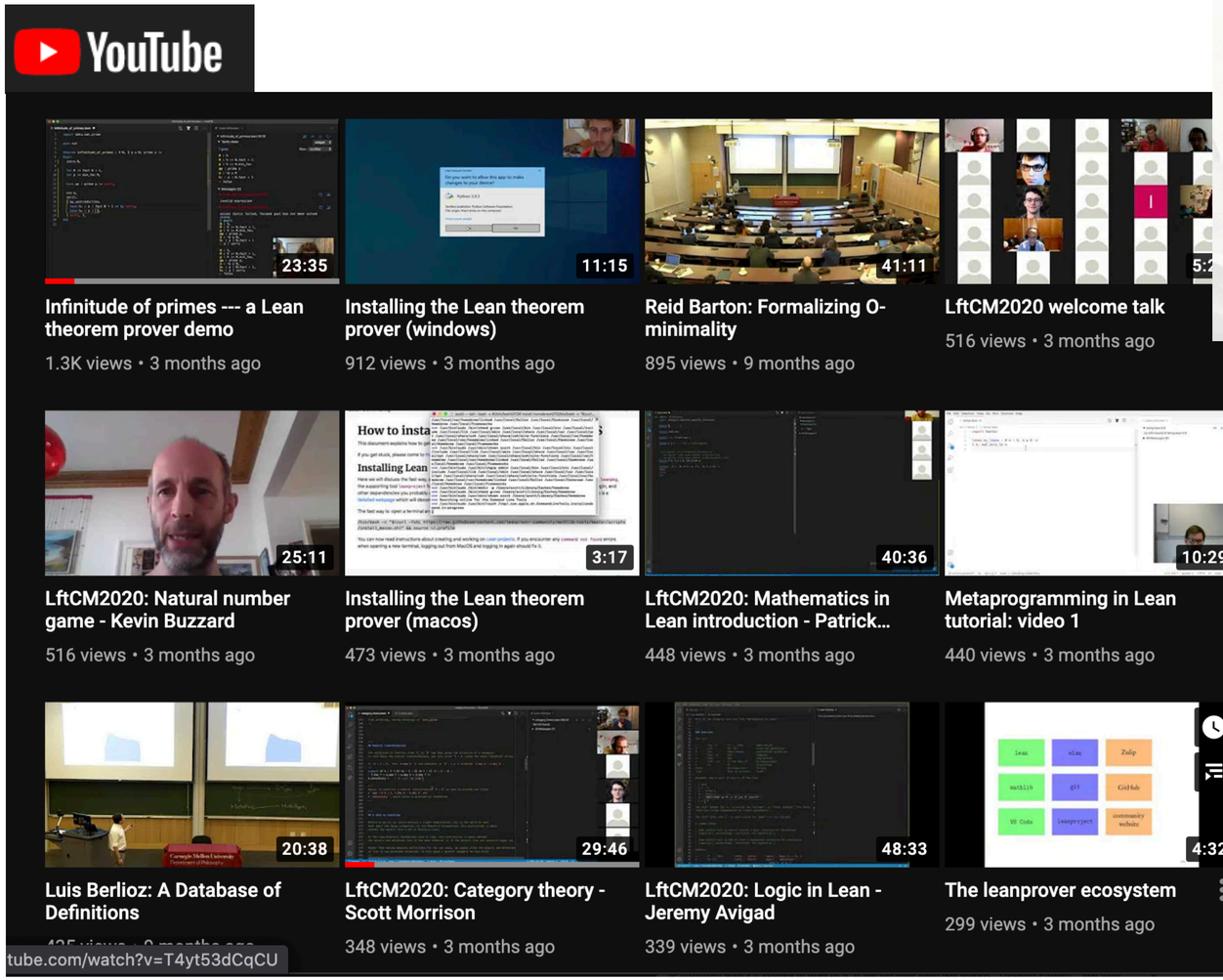
The Hitchhiker's Guide to Logical Verification

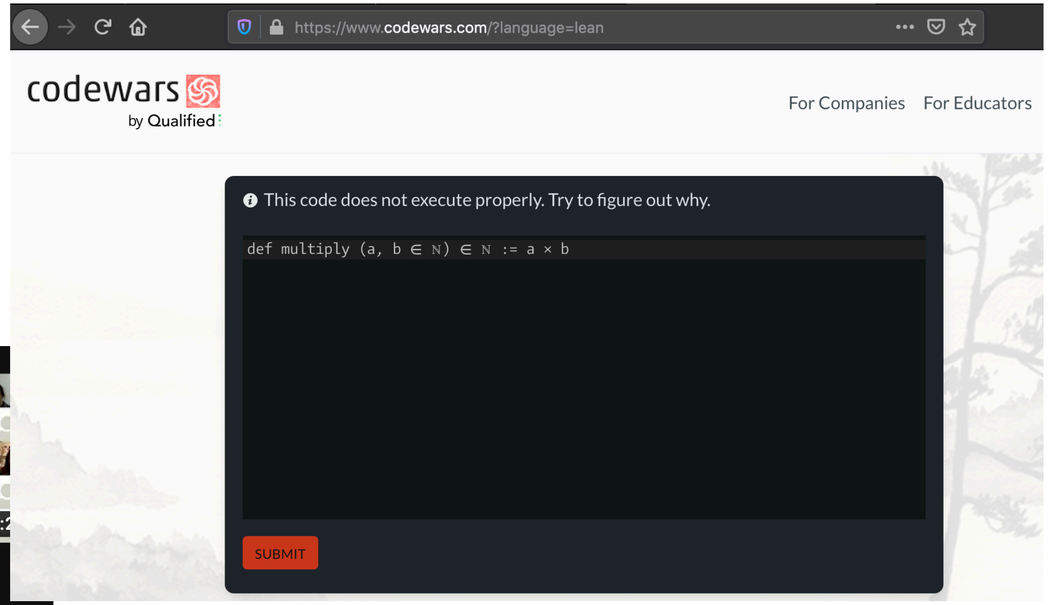
2020 Standard Edition
(October 12, 2020)

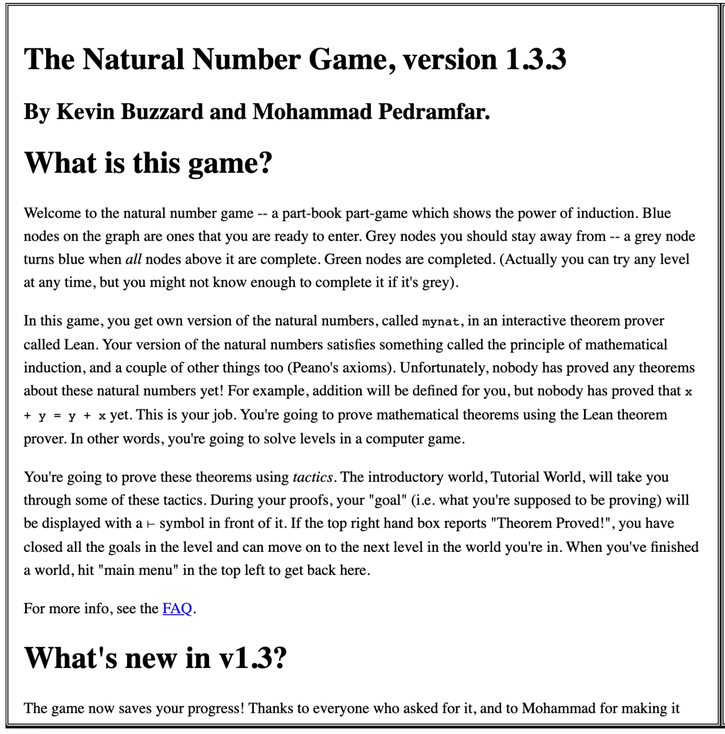
lean-forward.github.io/
logical-verification/2020

Mathematics in Lean

Online resources







The Natural Number Game, version 1.3.3

By Kevin Buzzard and Mohammad Pedramfar.

What is this game?

Welcome to the natural number game -- a part-book part-game which shows the power of induction. Blue nodes on the graph are ones that you are ready to enter. Grey nodes you should stay away from -- a grey node turns blue when *all* nodes above it are complete. Green nodes are completed. (Actually you can try any level at any time, but you might not know enough to complete it if it's grey).

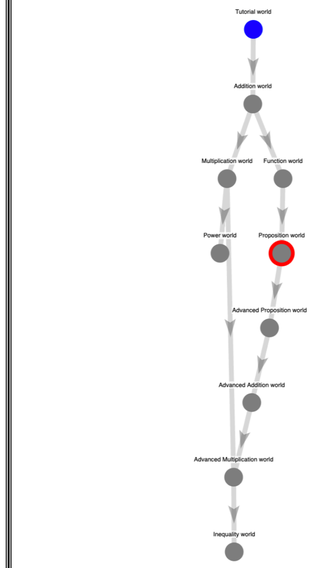
In this game, you get own version of the natural numbers, called `mynat`, in an interactive theorem prover called Lean. Your version of the natural numbers satisfies something called the principle of mathematical induction, and a couple of other things too (Peano's axioms). Unfortunately, nobody has proved any theorems about these natural numbers yet! For example, addition will be defined for you, but nobody has proved that `x + y = y + x` yet. This is your job. You're going to prove mathematical theorems using the Lean theorem prover. In other words, you're going to solve levels in a computer game.

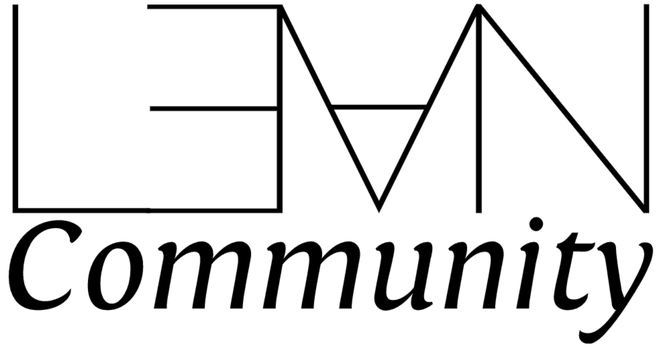
You're going to prove these theorems using *tactics*. The introductory world, Tutorial World, will take you through some of these tactics. During your proofs, your "goal" (i.e. what you're supposed to be proving) will be displayed with a ⊢ symbol in front of it. If the top right hand box reports "Theorem Proved!", you have closed all the goals in the level and can move on to the next level in the world you're in. When you've finished a world, hit "main menu" in the top left to get back here.

For more info, see the FAQ.

What's new in v1.3?

The game now saves your progress! Thanks to everyone who asked for it, and to Mohammad for making it





Lean and its Mathematical Library

The Lean theorem prover is a proof assistant developed principally by Leonardo de Moura at Microsoft Research.

The Lean mathematical library, *mathlib*, is a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant. The library also contains definitions useful for programming. This project is very active, with many regular contributors and daily activity.

The contents, design, and community organization of mathlib are described in the paper The Lean mathematical library, which appeared at CPP 2020. You can get a bird's eye view of what is in the library by reading the library overview. You can also have a look at our repository statistics to see how it grows and who contributes to it.

Try it!

You can try Lean in your web browser, install it in an isolated folder, or go for the full install. Lean is free, open source software. It works on Linux, Windows, and MacOS.

Learn to Lean!

You can learn by playing a game, following tutorials, or reading books.

Meet the community!

Lean has very diverse and active community. It gathers mostly on a Zulip chat and on GitHub. You can get involved and join the fun!

source

```
@[instance]
def division_ring.to_nontrivial (α : Type u) [s : division_ring α] :
   nontrivial α
```

source

```
@[instance]
def division_ring.to_has_inv (α : Type u) [s : division_ring α] :
   has_inv α
```

source

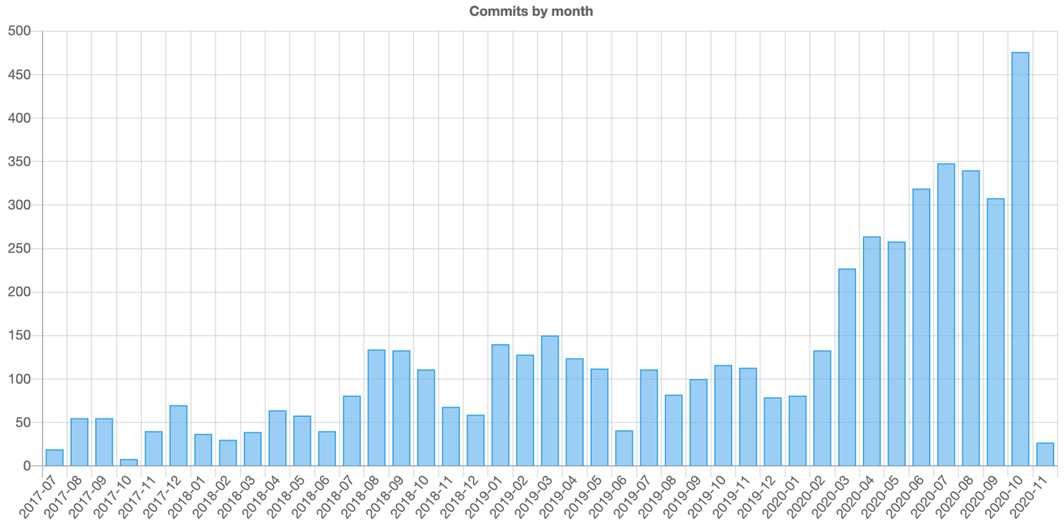
```
@[class]
structure division_ring  :
   Type u → Type u

(add : α → α → α)
(add_assoc : ∀ (a b c_1 : α), a + b + c_1 = a + (b + c_1))
(zero : α)
(zero_add : ∀ (a : α), 0 + a = a)
(add_zero : ∀ (a : α), a + 0 = a)
(neg : α → α)
(add_left_neg : ∀ (a : α), -a + a = 0)
(add_comm : ∀ (a b : α), a + b = b + a)
(mul : α → α → α)
(mul_assoc : ∀ (a b c_1 : α), (a * b) * c_1 = a * b * c_1)
(one : α)
(one_mul : ∀ (a : α), 1 * a = a)
(mul_one : ∀ (a : α), a * 1 = a)
```

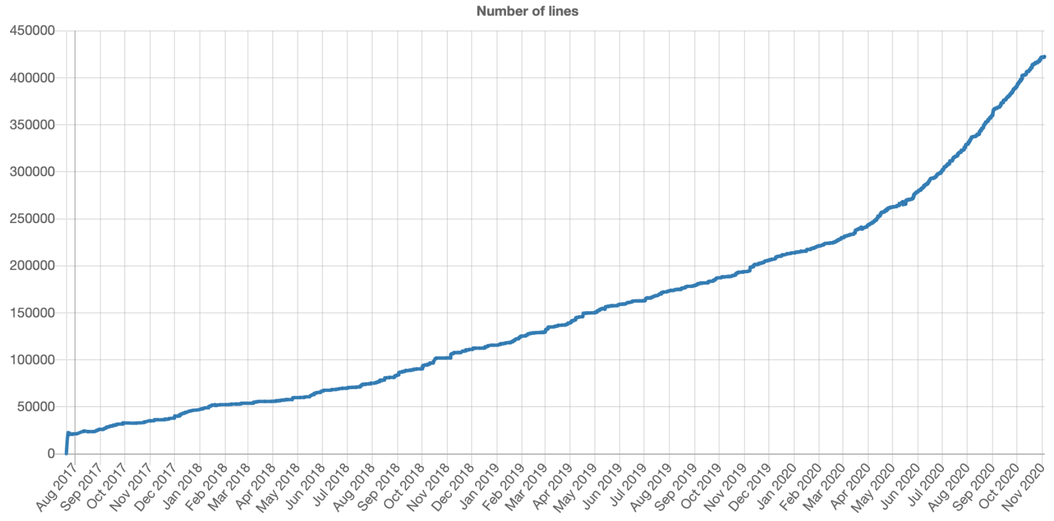
Mathlib statistics

Counts

Definitions	Theorems	Contributors
19187	41102	122

Temporal distribution





Lean Together : annual event

- Organized by the community
- More than 50 participants per edition

- 2019 - Amsterdam
- 2020 - Pittsburgh
- 2021 - Virtual (Jan 4 - 8)
  - Sebastian Ullrich and I will make the official Lean 4 release during this event

Lean in the media

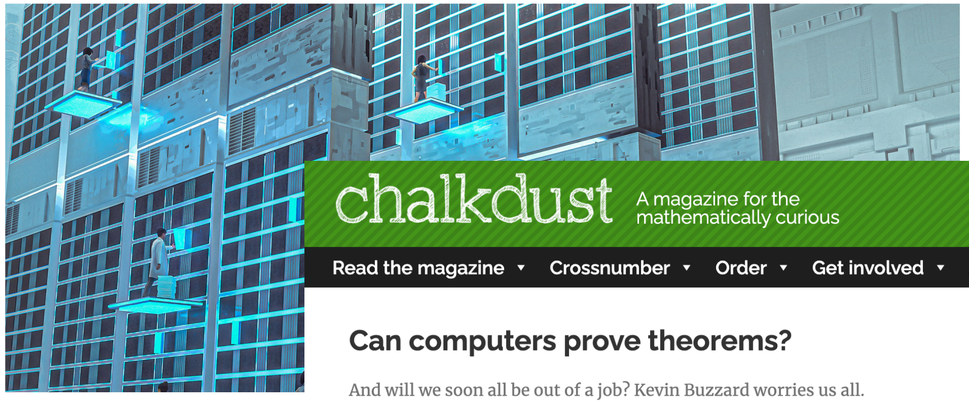
Quantamagazine Physics Mathematics Biology Computer Science All Articles

FOUNDATIONS OF MATHEMATICS

Building the Mathematical Library of the Future

18

A small community of mathematicians is using a software program called Lean to build a new digital repository. They hope it represents the future of their field.



chalkdust A magazine for the mathematically curious

Read the magazine ▾ Crossnumber ▾ Order ▾ Get involved ▾ About us ▾

Can computers prove theorems?

And will we soon all be out of a job? Kevin Buzzard worries us all.

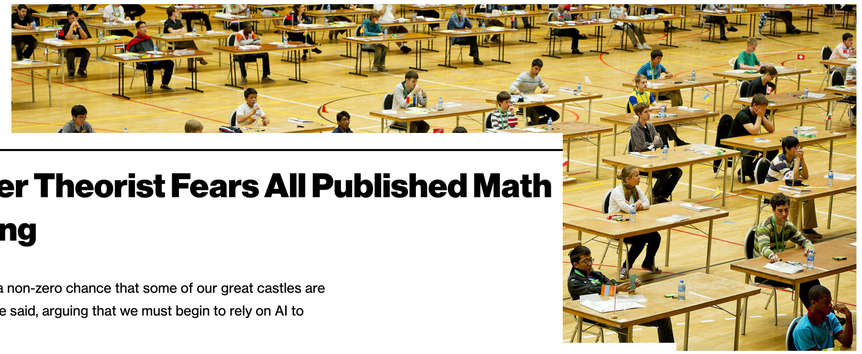
Quantamagazine Physics Mathematics Biology Computer Science All Articles

ABSTRACTIONS BLOG

At the Math Olympiad, Computers Prepare to Go for the Gold

11

Computer scientists are trying to build an AI system that can win a gold medal at the world's premier math competition.

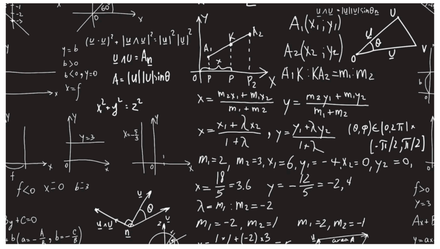


Number Theorist Fears All Published Math Is Wrong

"I think there is a non-zero chance that some of our great castles are built on sand," he said, arguing that we must begin to rely on AI to verify proofs.

MR By Mordechai Rorvig

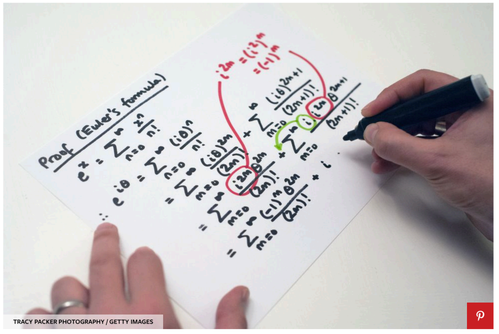
September 26, 2019, 5:30am Share Tweet Snap



What If All Published Math Is ... Wrong?

A number theorist says it's possible, and makes the case for A.I. to double-check proofs.

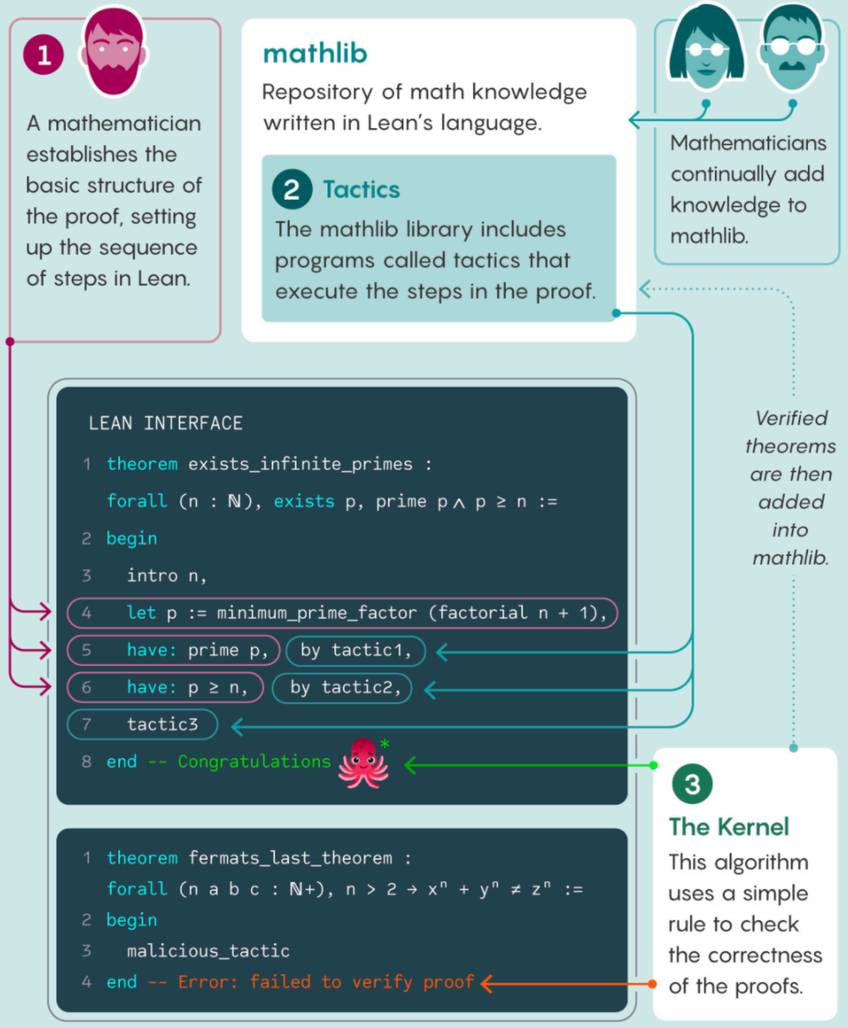
// BY DAVID GROSSMAN SEP 27, 2019



TRACY PACKER PHOTOGRAPHY / GETTY IMAGES

Anatomy of a Proof Assistant

The proof assistant Lean helps mathematicians prove theorems by checking their work and automating some of the steps in a proof.



* The octopus has somehow become the go-to emoji for happiness in the Lean community.

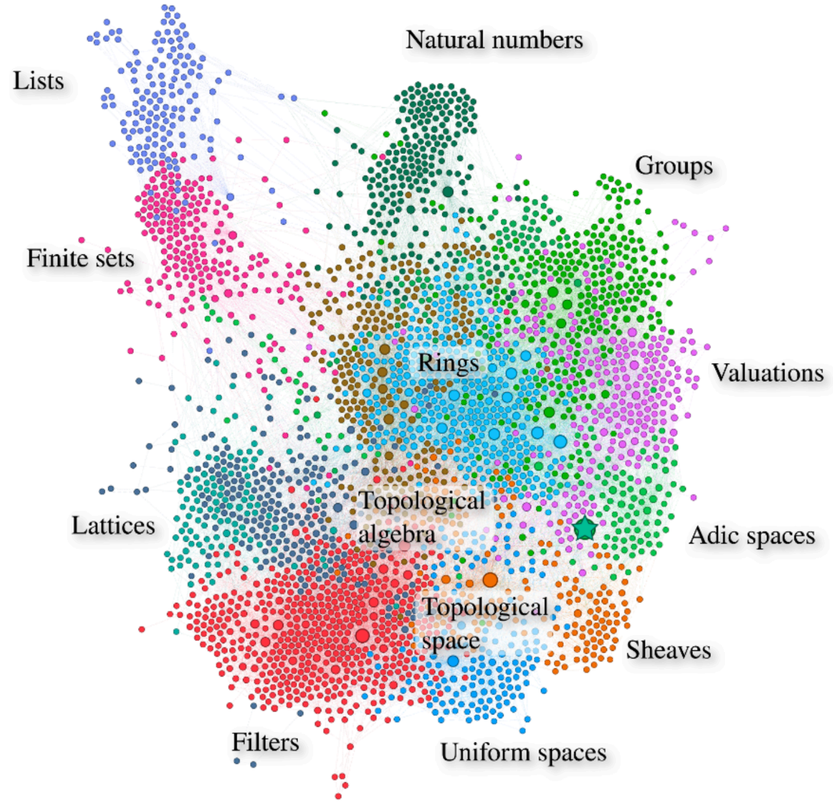
Lean perfectoid spaces

by Kevin Buzzard, Johan Commelin, and Patrick Massot

What is it about?

We explained Peter Scholze's definition of perfectoid spaces to computers, using the Lean theorem prover, mainly developed at Microsoft Research by Leonardo de Moura. Building on earlier work by many people, starting from first principles, we arrived at

```
-- We fix a prime number p
parameter (p : primes)

/-- A perfectoid ring is a Huber ring that is complete, uniform,
that has a pseudo-uniformizer whose p-th power divides p in the power bounded subring,
and such that Frobenius is a surjection on the reduction modulo p.-/
structure perfectoid_ring (R : Type) [Huber_ring R] extends Tate_ring R : Prop :=
(complete  : is_complete_hausdorff R)
(uniform   : is_uniform R)
(ramified  : ∃ ϖ : pseudo_uniformizer R, ϖ^p | p in R°)
(Frobenius : surjective (Frob R°/p))

/-
CLVRS ("complete locally valued ringed space") is a category
whose objects are topological spaces with a sheaf of complete topological rings
and an equivalence class of valuation on each stalk, whose support is the unique
maximal ideal of the stalk; in Wedhorn's notes this category is called 𝒱.
A perfectoid space is an object of CLVRS which is locally isomorphic to Spa(A) with
A a perfectoid ring. Note however that CLVRS is a full subcategory of the category
`PreValuedRingedSpace` of topological spaces equipped with a presheaf of topological
rings and a valuation on each stalk, so the isomorphism can be checked in
PreValuedRingedSpace instead, which is what we do.
-/

/-- Condition for an object of CLVRS to be perfectoid: every point should have an open
neighbourhood isomorphic to Spa(A) for some perfectoid ring A.-/
def is_perfectoid (X : CLVRS) : Prop :=
∀ x : X, ∃ (U : opens X) (A : Huber_pair) [perfectoid_ring A],
  (x ∈ U) ∧ (Spa A ≅ U)

/-- The category of perfectoid spaces.-/
def PerfectoidSpace := {X : CLVRS // is_perfectoid X}

end
```



mathoverflow

Home

Questions

Tags

What are "perfectoid spaces"?

Asked 9 years, 5 months ago Active 1 year, 5 months ago Viewed 49k times

59

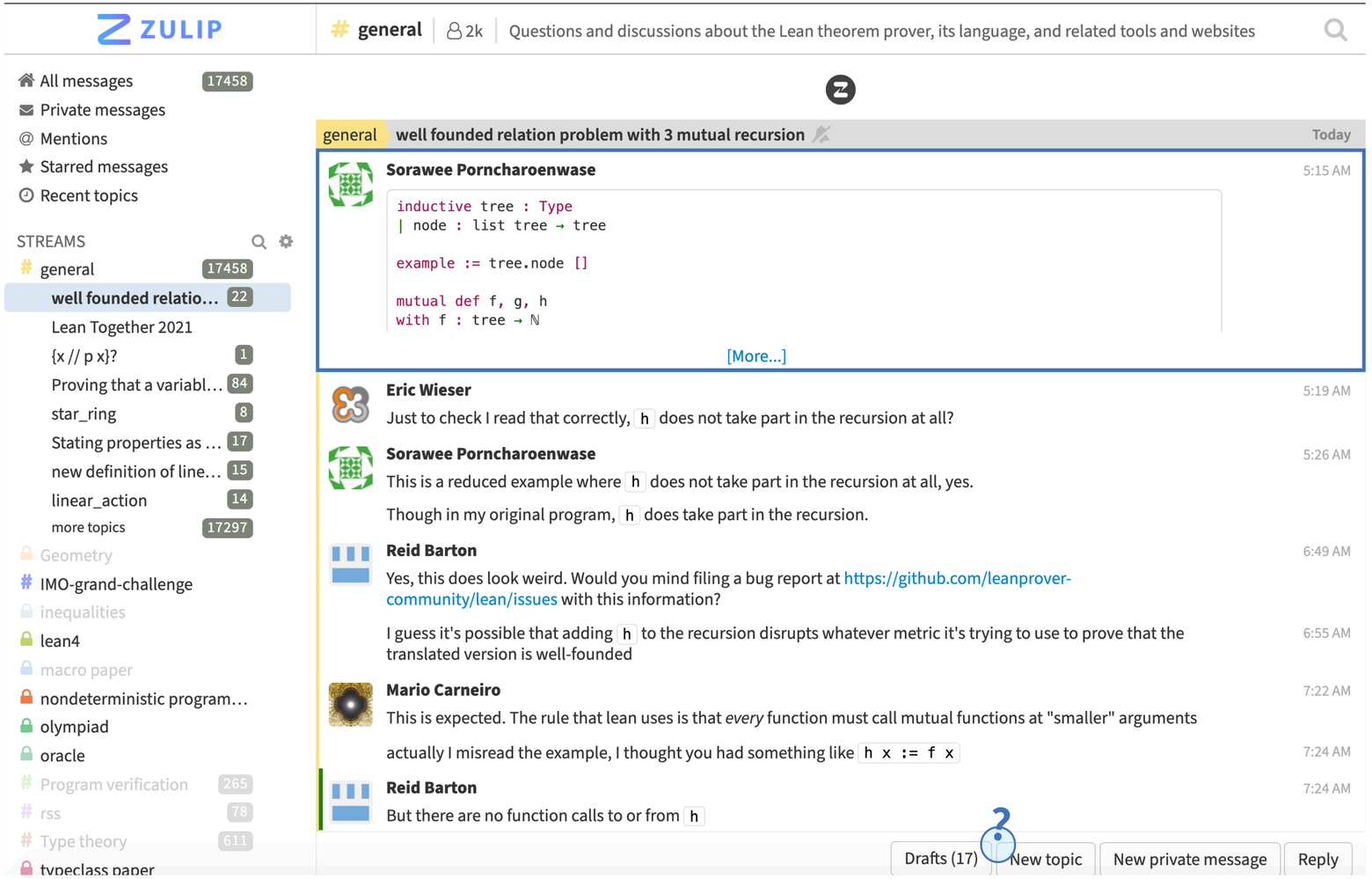
Here is a completely different kind of answer to this question.

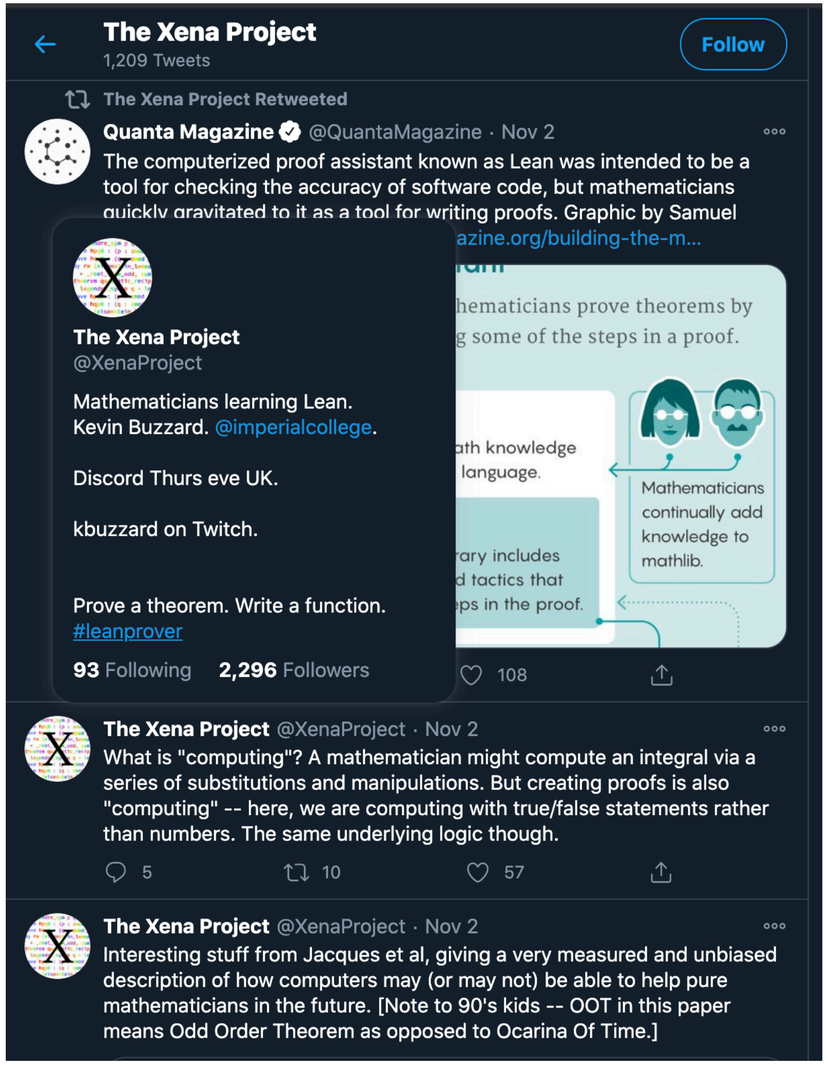
A *perfectoid space* is a term of type `PerfectoidSpace` in the Lean theorem prover.

Here's a quote from the source code:

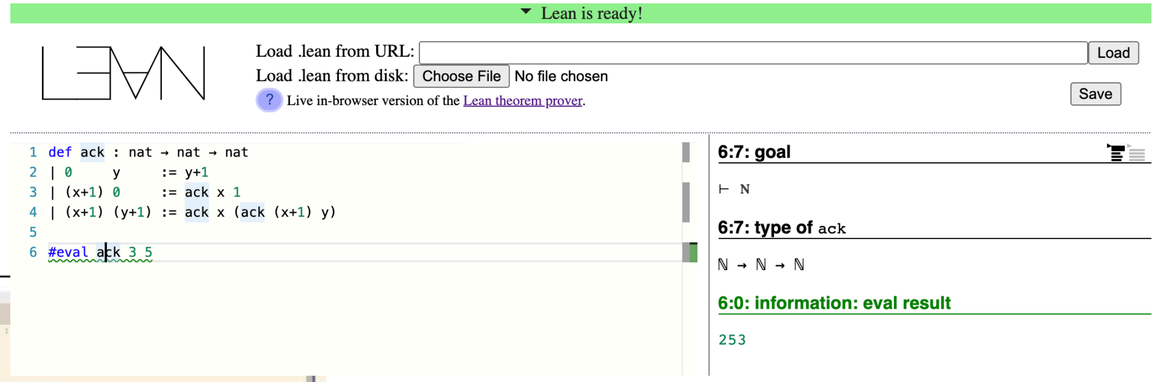
```
structure perfectoid_ring (R : Type) [Huber_ring R] extends Tate_ring R : Prop :=
(complete  : is_complete_hausdorff R)
```

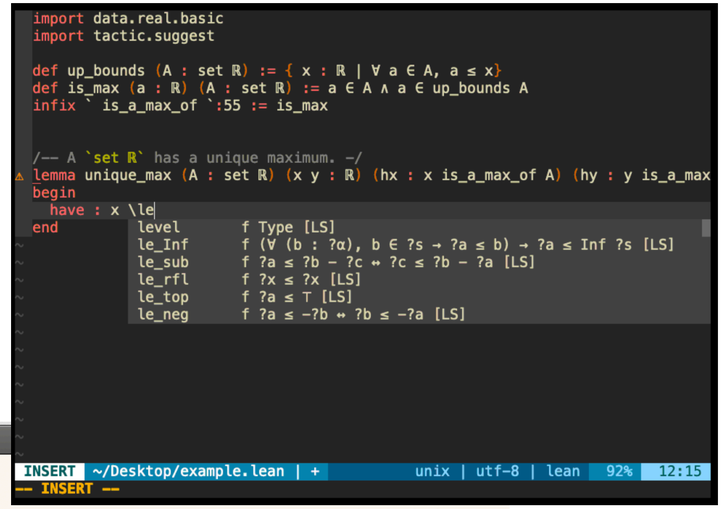
Social media

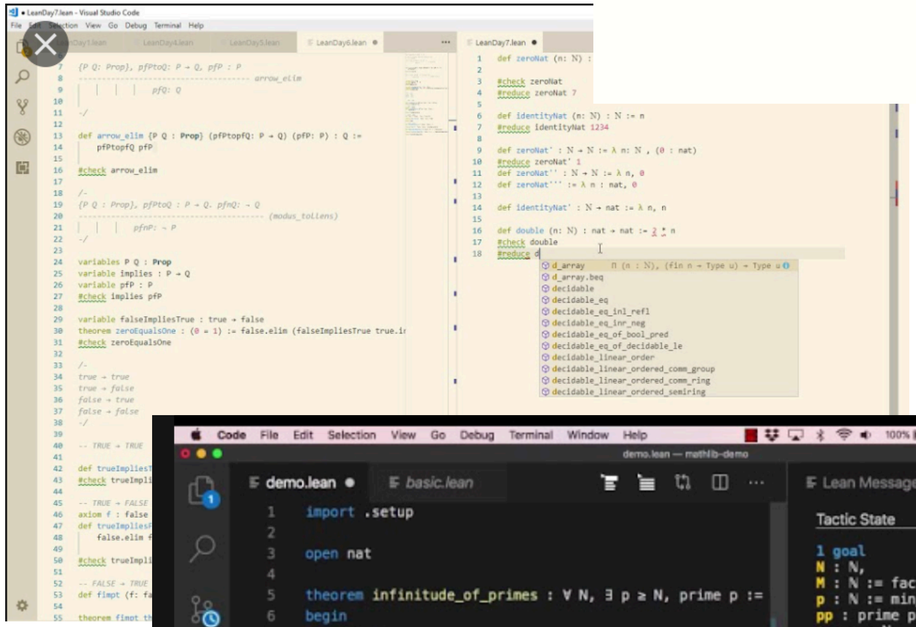


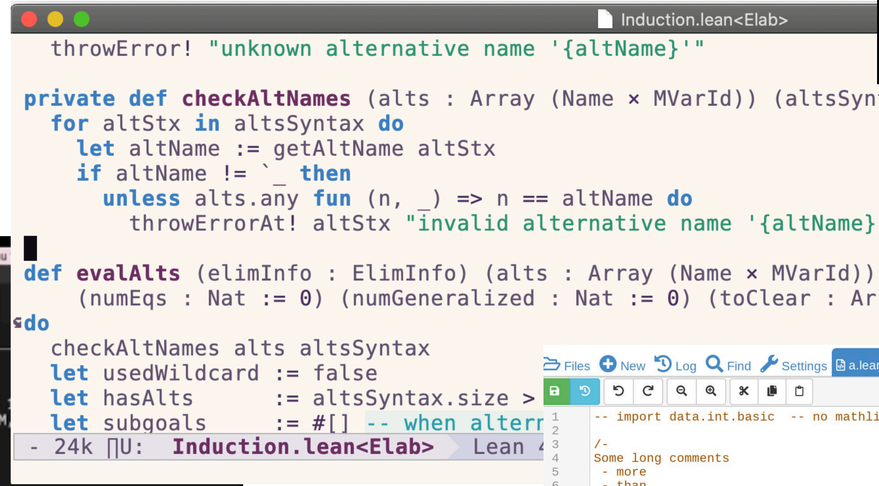


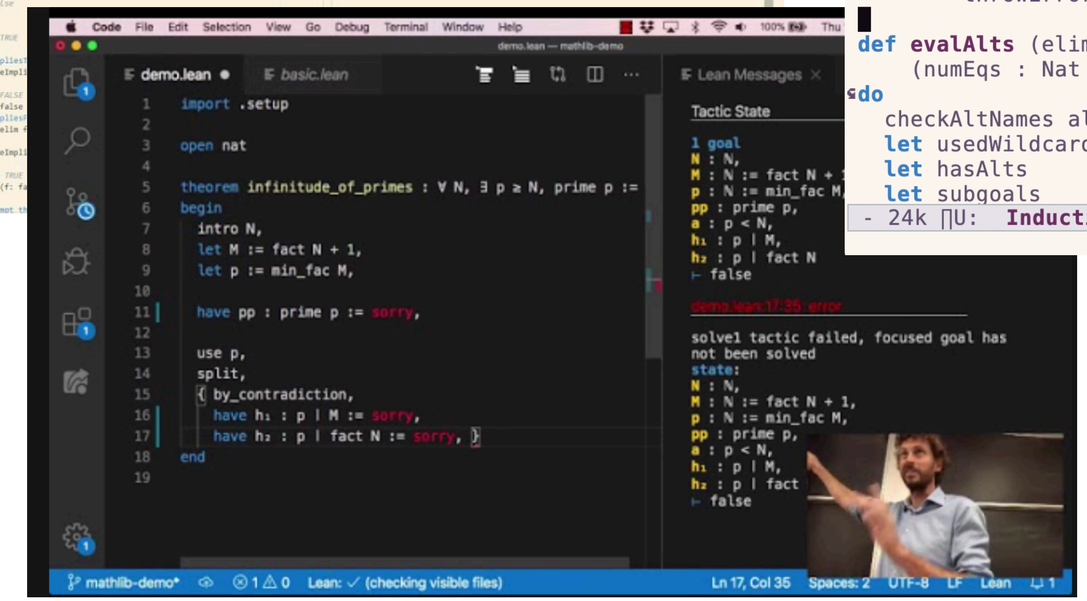
IDEs

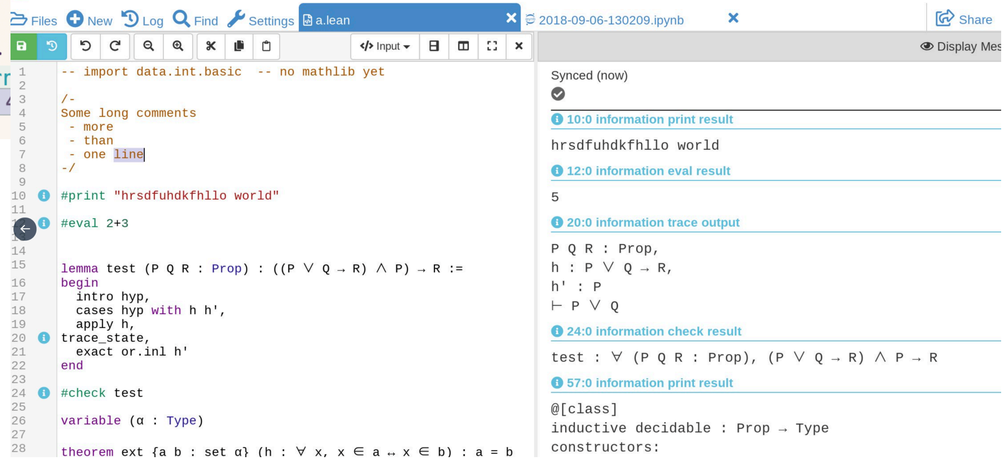

Lean is ready!
Load .lean from URL:
Load
Load .lean from disk:
Choose File
No file chosen
Live in-browser version of the Lean theorem prover.
Save
def ack : nat → nat → nat
| 0 y := y+1
| (x+1) 0 := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)
#eval ack 3 5
6:7: goal
⊢ ℕ
6:7: type of ack
ℕ → ℕ → ℕ
6:0: information: eval result
253


import data.real.basic
import tactic.suggest
def up_bounds (A : set ℝ) := { x : ℝ | ∀ a ∈ A, a ≤ x}
def is_max (a : ℝ) (A : set ℝ) := a ∈ A ∧ a ∈ up_bounds A
infix ` is_a_max_of `:55 := is_max
/-- A `set ℝ` has a unique maximum. -/
lemma unique_max (A : set ℝ) (x y : ℝ) (hx : x is_a_max_of A) (hy : y is_a_max
begin
have : x \le
end
level f Type [LS]
le_Inf f (∀ (b : ?α), b ∈ ?s → ?a ≤ b) → ?a ≤ Inf ?s [LS]
le_sub f ?a ≤ ?b - ?c ↔ ?c ≤ ?b - ?a [LS]
le_rfl f ?x ≤ ?x [LS]
le_top f ?a ≤ T [LS]
le_neg f ?a ≤ -?b ↔ ?b ≤ -?a [LS]
INSERT ~/Desktop/example.lean | + unix | utf-8 | lean 92% 12:15
-- INSERT --


• LeanDay7.lean - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
LeanDay1.lean LeanDay4.lean LeanDay5.lean LeanDay6.lean LeanDay7.lean
{P Q: Prop}, pfPtoQ: P → Q, pfP : P
arrow_elim
pfQ: Q
def arrow_elim {P Q : Prop} (pfPtopfQ: P → Q) (pfP: P) : Q :=
pfPtopfQ pfP
#check arrow_elim
{P Q : Prop}, pfPtoQ : P → Q, pfnQ: ¬ Q
(modus_tollens)
pfnP: ¬ P
variables P Q : Prop
variable implies : P → Q
variable pfP : P
#check implies pfP
variable falseImpliesTrue : true → false
theorem zeroEqualsOne : (0 = 1) := false.elim (falseImpliesTrue true.ir
#check zeroEqualsOne
true → true
true → false
false → true
false → false
-- TRUE → TRUE
-- TRUE → FALSE
axiom f : false
-- FALSE → TRUE
def fimpt (f: fa
theorem fimpt th
def zeroNat (n: N) :
#check zeroNat
#reduce zeroNat 7
def identityNat (n: N) : N := n
#reduce identityNat 1234
def zeroNat' : N → N := λ n: N , (0 : nat)
#reduce zeroNat' 1
def zeroNat'' : N → N := λ n, 0
def zeroNat''' := λ n : nat, 0
def identityNat' : N → nat := λ n, n
def double (n: N) : nat → nat := λ x * n
#check double
#reduce d
d_array
d_array.beq
decidable
decidable_eq
decidable_eq_inl_refl
decidable_eq_inr_neg
decidable_eq_of_bool_pred
decidable_eq_of_decidable_le
decidable_linear_order
decidable_linear_ordered_comm_group
decidable_linear_ordered_comm_ring
decidable_linear_ordered_semiring

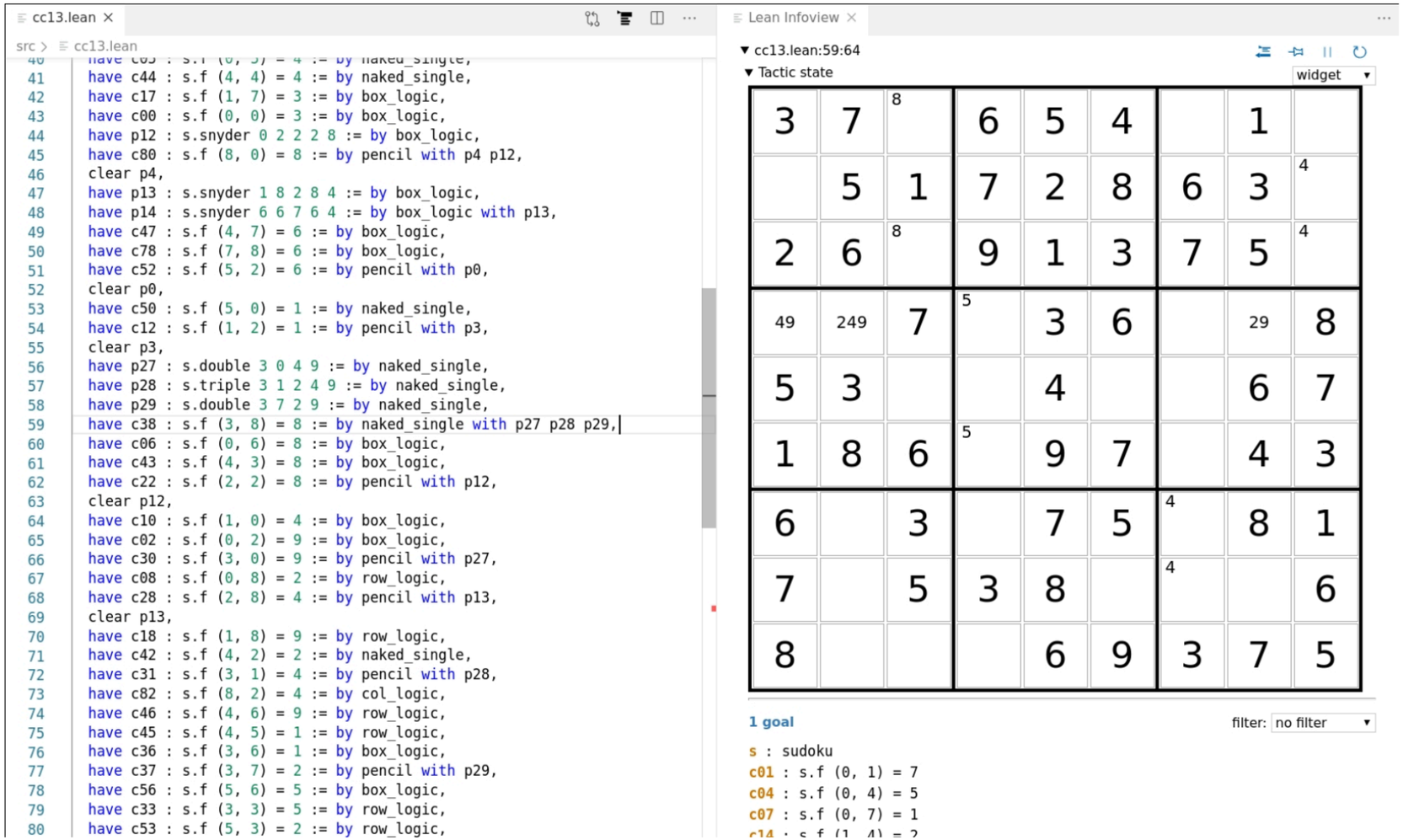

Induction.lean<Elab>
throwError! "unknown alternative name '{altName}'"
private def checkAltNames (alts : Array (Name × MVarId)) (altsSyntax : Array Syntax) : TacticM Unit :=
for altStx in altsSyntax do
let altName := getAltName altStx
if altName != `_ then
unless alts.any fun (n, _) => n == altName do
throwErrorAt! altStx "invalid alternative name '{altName}'"
def evalAlts (elimInfo : ElimInfo) (alts : Array (Name × MVarId)) (altsSyntax : Array Syntax)
(numEqs : Nat := 0) (numGeneralized : Nat := 0) (toClear : Array FVarId := #[]) : TacticM Unit :=
do
checkAltNames alts altsSyntax
let usedWildcard := false
let hasAlts := altsSyntax.size >
let subgoals := #[] -- when alterr
- 24k ∏U: Induction.lean<Elab> Lean


Code File Edit Selection View Go Debug Terminal Window Help
demo.lean — mathlib-demo
demo.lean
basic.lean
import .setup
open nat
theorem infinitude_of_primes : ∀ N, ∃ p ≥ N, prime p :=
begin
intro N,
let M := fact N + 1,
let p := min_fac M,
have pp : prime p := sorry,
use p,
split,
{ by_contradiction,
have h₁ : p ∣ M := sorry,
have h₂ : p ∣ fact N := sorry, },
end
Lean Messages
Tactic State
1 goal
N : ℕ,
M : ℕ := fact N +
p : ℕ := min_fac M,
pp : prime p,
a : p < N,
h₁ : p ∣ M,
h₂ : p ∣ fact N
⊢ false
demo.lean:17:35: error
solve1 tactic failed, focused goal has not been solved
state:
N : ℕ,
M : ℕ := fact N + 1,
p : ℕ := min_fac M,
pp : prime p,
a : p < N,
h₁ : p ∣ M,
h₂ : p ∣ fact
⊢ false
mathlib-demo* 1 ⚠ 0 Lean: ✓ (checking visible files) Ln 17, Col 35 Spaces: 2 UTF-8 LF Lean 1


Files New Log Find Settings a.lean 2018-09-06-130209.ipynb Share
Input
Display Mes
-- import data.int.basic -- no mathlib yet
/-
Some long comments
- more
- than
- one line
-/
#print "hrsdfuhdkfhllo world"
#eval 2+3
lemma test (P Q R : Prop) : ((P ∨ Q → R) ∧ P) → R :=
begin
intro hyp,
cases hyp with h h',
apply h,
trace_state,
exact or.inl h'
end
#check test
variable (α : Type)
theorem ext {a b : set α} (h : ∀ x, x ∈ a ↔ x ∈ b) : a = b
Synced (now)
10:0 information print result
hrsdfuhdkfhllo world
12:0 information eval result
5
20:0 information trace output
P Q R : Prop,
h : P ∨ Q → R,
h' : P
⊢ P ∨ Q
24:0 information check result
test : ∀ (P Q R : Prop), (P ∨ Q → R) ∧ P → R
57:0 information print result
@[class]
inductive decidable : Prop → Type
constructors:

Fun IDE extensions

Playing sudoku in the Lean theorem prover



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.
- Trace messages are just strings.

It’s been a long time coming …

“We should really refactor the elaborator as well”

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

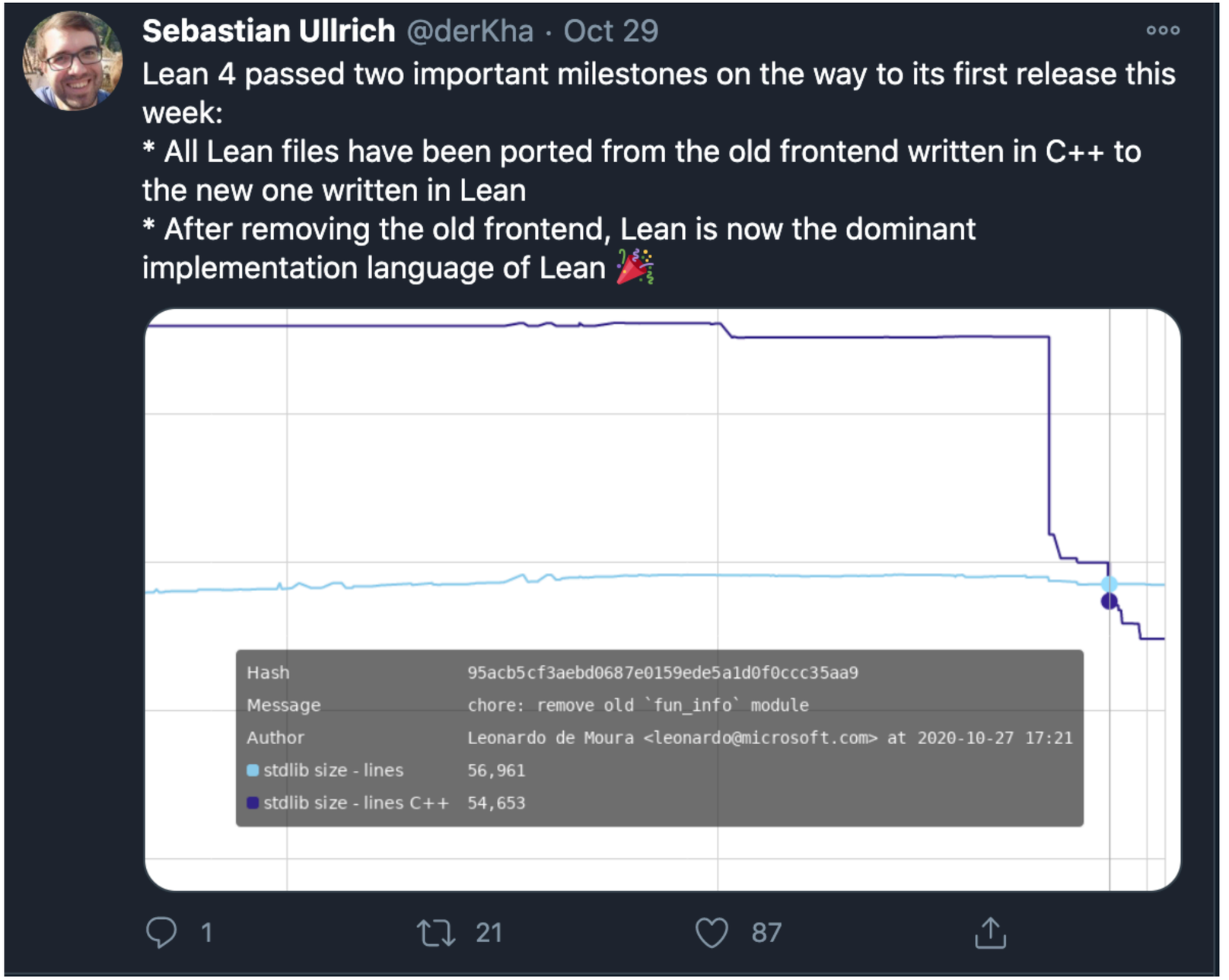
“We first need a capable Lean compiler for that …”

Thus the Lean 4 project was born.

Two developers: Sebastian Ullrich (KIT) and Leo

LEAN 4

- Implement Lean in Lean
  - Parser, elaborator, compiler, tactics and formatter (all user extensible)
  - Hygienic macro system
  - Only the runtime and basic primitives are implemented in C
- Runtime has support for boxed and unboxed data
- Runtime uses reference counting for GC and performs destructive updates when RC = 1
- Compiler generates C code. We can mix byte code and compiled code
- (Safe) support for low-level tricks such as pointer equality
- A better value proposition: use proofs for obtaining more efficient code


Sebastian Ullrich @derKha · Oct 29
Lean 4 passed two important milestones on the way to its first release this week:
* All Lean files have been ported from the old frontend written in C++ to the new one written in Lean
* After removing the old frontend, Lean is now the dominant implementation language of Lean
Hash 95acb5cf3aebd0687e0159ede5a1d0f0ccc35aa9
Message chore: remove old `fun_info` module
Author Leonardo de Moura <leonardo@microsoft.com> at 2020-10-27 17:21
stdlib size - lines 56,961
stdlib size - lines C++ 54,653
1
21
87

Extensible parser

- No tokenizer
- Parsers combinators
- Pratt's parsers: leading and trailing parsers
  - New twist: parsers, not tokens, have precedences

```
syntax:100 "-" term:100     : term
syntax:65 term "-" term:66  : term
syntax:65 term "+" term:66  : term
syntax:67 term "::" term:67 : term

infixl:65 "+"  => add
infixr:67 "::" => List.cons
```

- Automatically generated formatters, parenthesizers

```
-- 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 `_bigop` with the different kinds of index
macro_rules
| `(_big [$op, $idx] ($i:ident <- $r | $p) $F) => `(bigop $idx $r (fun $i:ident => ($i:ident, $op, $p, $F)))
| `(_big [$op, $idx] ($i:ident <- $r) $F) => `(bigop $idx $r (fun $i:ident => ($i:ident, $op, true, $F)))
| `(_big [$op, $idx] ($lower:term ≤ $i:ident < $upper) $F) => `(bigop $idx (index_iota $lower $upper) (fun $i:ident => ($i:ident, $op, true, $F)))
| `(_big [$op, $idx] ($lower:term ≤ $i:ident < $upper | $p) $F) => `(bigop $idx (index_iota $lower $upper) (fun $i:ident => ($i:ident, $op, $p, $F)))

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

-- We can already use `Sum` with the different kinds of index.
#check Sum (i <- [0, 2, 4] | i != 2) i
#check Sum (10 ≤ i < 20 | i != 5) i+1
#check Sum (10 ≤ i < 20) i+1

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

-- The examples above now also work for `Prod`
#check Prod (i <- [0, 2, 4] | i != 2) i
#check Prod (10 ≤ i < 20 | i != 5) i+1
#check Prod (10 ≤ i < 20) i+1
```

A unified frontend system

Notations

```
notation "∃" b "," P ⇒ Exists (fun b ⇒ P)
```

⇓

Macros $\quad Surf \to Surf$

```
macro "∃" b:term "," P:term : term ⇒
`(Exists (fun $b ⇒ $P))
```

⇓

Elaborators $\quad Surf \to Core$

```
elab "∃" b:term "," P:term : term ⇒
`(Exists (fun $b ⇒ $P)) >>= elabTerm
```

Equal Hygiene guarantees for all levels

Hygiene

```
notation "const" e ⇒ fun x ⇒ e
```

"Of course" *e* may not capture *x*

```
macro "elab" ... ⇒ do
  ...;
  `(@[$elabAttr] def myElaborator (stx : Syntax) : $type := match_syntax stx with ...)
```

"Of course" *myElaborator* may not be captured from outside

String interpolation: a micro DSL

```
-- 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: a micro DSL

```
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}"
```

“do” notation : another DSL

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”

“do” notation : another DSL

```
def sum (xs : Array Nat) : IO Nat := do
  let 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
```

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

“do” notation : another DSL

```
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
```

Type-directed macros

```
@[builtinTermElab anonymousCtor]
def elabAnonymousCtor : TermElab :=
  fun stx expectedType? =>
    match_syntax stx with
    | `(⟨$args*⟩) => do
      tryPostponeIfNoneOrMVar expectedType?
      match expectedType? with
      | some expectedType =>
        let expectedType ← whnf expectedType
        matchConstInduct expectedType.getAppFn
          (fun _ => throwError! "invalid constructor ⟨...⟩, expected type must be an inductive type {indentExpr expectedType}")
          (fun ival us => do
            match ival.ctors with
            | [ctor] =>
              let newStx ← `($(mkCIdentFrom stx ctor) $(args.getSepElems)*)
              withMacroExpansion stx newStx $ elabTerm newStx expectedType?
            | _ =>
              throwError! "invalid constructor ⟨...⟩, expected type must be an inductive type with only one constructor {indentExpr expectedType}")
      | none => throwError "invalid constructor ⟨...⟩, expected type must be known"
    | _ => throwUnsupportedSyntax
```

Tactic/synthesis framework: another DSL

Go to tactic/synthesis mode

```
variables {α : Type u} {β : Type v}
variables {ra  : α → α → Prop} {rb  : β → β → Prop}

def lexAccessible (aca : (a : α) → Acc ra a) (acb : (b : β) → Acc rb b) (a : α) (b : β) : Acc (Lex ra rb) (a, b) := by
  induction (aca a) generalizing b
  | intro xa aca iha =>
    induction (acb b)
    | intro xb acb ihb =>
      apply Acc.intro (xa, xb)
      intro p lt
      cases lt
      | left  a₁ b₁ a₂ b₂ h => apply iha a₁ h
      | right a b₁ b₂ h     => apply ihb b₁ h
```

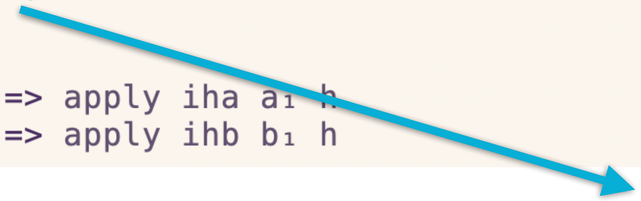
Construct a lambda

Construct an application

Tactic/synthesis framework: another DSL

It is interactive: we can visualize intermediate states

```
variables {α : Type u} {β : Type v}
variables {ra  : α → α → Prop} {rb  : β → β → Prop}

def lexAccessible (aca : (a : α) → Acc ra a) (acb : (b : β) → Acc rb b) (a : α) (b : β) : Acc (Lex ra rb) (a, b) := by
  induction (aca a) generalizing b
  | intro xa aca iha =>
    induction (acb b)
    | intro xb acb ihb =>
      apply Acc.intro (xa, xb)
      intro p lt
      cases lt
      | left  a₁ b₁ a₂ b₂ h => apply iha a₁ h
      | right a b₁ b₂ h     => apply ihb b₁ h
```



```
unsolved goals
case intro.intro
α : Type u
β : Type v
ra : α → α → Prop
rb : β → β → Prop
aca✝ : (a : α) → Acc ra a
acb✝ : (b : β) → Acc rb b
a xa : α
aca : (y : α) → ra y xa → Acc ra y
iha : (y : α) → (a : ra y xa) → (b : β) → Acc (Lex ra rb) (y, b)
b xb : β
acb : (y : β) → rb y xb → Acc rb y
ihb : (y : β) → (a : rb y xb) → Acc (Lex ra rb) (xa, y)
⊢ Acc (Lex ra rb) (xa, xb)
```

Lifting notation to another DSL

```
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
```

The tactic framework is implemented in Lean itself

```
def cases (mvarId : MVarId) (majorFVarId : FVarId) (givenNames : Array (List Name)) (useUnusedNames : Bool) : MetaM (Array CasesSubgoal) :=
  withMVarContext mvarId do
    checkNotAssigned mvarId `cases
    let context? ← mkCasesContext? majorFVarId
    match context? with
    | none     => throwTacticEx `cases mvarId "not applicable to the given hypothesis"
    | some ctx =>
      if ctx.inductiveVal.nindices == 0 then
        inductionCasesOn mvarId majorFVarId givenNames useUnusedNames ctx
      else
        let s₁ ← generalizeIndices mvarId majorFVarId
        trace[Meta.Tactic.cases]! "after generalizeIndices\n{MessageData.ofGoal s₁.mvarId}"
        let s₂ ← inductionCasesOn s₁.mvarId s₁.fvarId givenNames useUnusedNames ctx
        let s₂ ← elimAuxIndices s₁ s₂
        unifyCasesEqs s₁.numEqs s₂
```

Users can add their own primitives

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 we cannot prove False in Lean.
  - Theorems proved in Lean 4 may still be checked by reference checkers.
  - Unsafe functions are ignored by reference checkers.
- Allow developers to provide an unsafe version for any (opaque) function whose type is inhabited.
- Examples:
  - Primitives implemented in C

```
@[extern "lean_uint64_mix_hash"]
constant mixHash64 (u₁ u₂ : UInt64) : UInt64 := 0
```

  - Sealing unsafe features

```
unsafe def setStateUnsafe {σ : Type} (ext : EnvExtension σ) (env : Environment) (s : σ) : Environment :=
{ env with extensions := env.extensions.set! ext.idx (unsafeCast s) }

@[implementedBy setStateUnsafe]
constant setState {σ : Type} (ext : EnvExtension σ) (env : Environment) (s : σ) : Environment := env
```

The partial keyword

- General recursion is a major convenience.
  - Some functions in our implementation may not terminate or cannot be shown to terminate in Lean, and we want to avoid an artificial “fuel” argument.
  - In many cases, the function terminates, but we don’t want to “waste" time proving it.

```
partial def whnfImpl : Expr → MetaM Expr
```

- A **partial** definition is just syntax sugar for the **unsafe** + **implementedBy** idiom.
- Future work: allow users to provide termination later, and use meta programming to generate a safe and non-opaque version of a partial function.

Proofs for performance and profit

- A better value proposition: use proofs for obtaining more efficient code.
- Example: skip runtime array bounds checks

```
def get (a : Array α) (i : Nat) (h : i < a.size) : α
```

- Example: pointer equality (join work with Daniel Selsam and Simon Hudon)

```
def withPtrEq (x y : α) (k : Unit → Bool)
  (h : x = y → k () = true) : Bool := k ()
```

The definition is called a reference implementation

The compiler generates:

```
def withPtrEq (x y : α) (k : Unit → Bool)
  (h : x = y → k () = true) : Bool :=
if ptrAddr x = ptrAddr y
  then true
  else k ()
```

The return of reference counting

- Most compilers for functional languages (OCaml, GHC, …) use tracing GC
- RC is simple to implement.
- Easy to support multi-threading programs.
- Destructive updates when reference count = 1.
  - It is a known optimization for big objects (e.g., arrays).
  - We demonstrate it is also relevant for small objects.
- In languages like Coq and **Lean**, we do not have cycles.
- Easy to interface with C, C++ and Rust.

Paper: "Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming", IFL 2019

Resurrection hypothesis

Many objects die just before the creation of an object of the same kind.

Examples:

- List.map : List a -> (a -> b) -> List b
- Compiler applies transformations to expressions.
- Proof assistant rewrites/simplifies formulas.
- Updates to functional data structures such as red black trees.
- List zipper

$$goForward\,([],\,bs) = ([],\,bs)$$
$$goForward\,(x\,:xs,\,bs) = (xs,\,x\,:bs)$$

New idioms

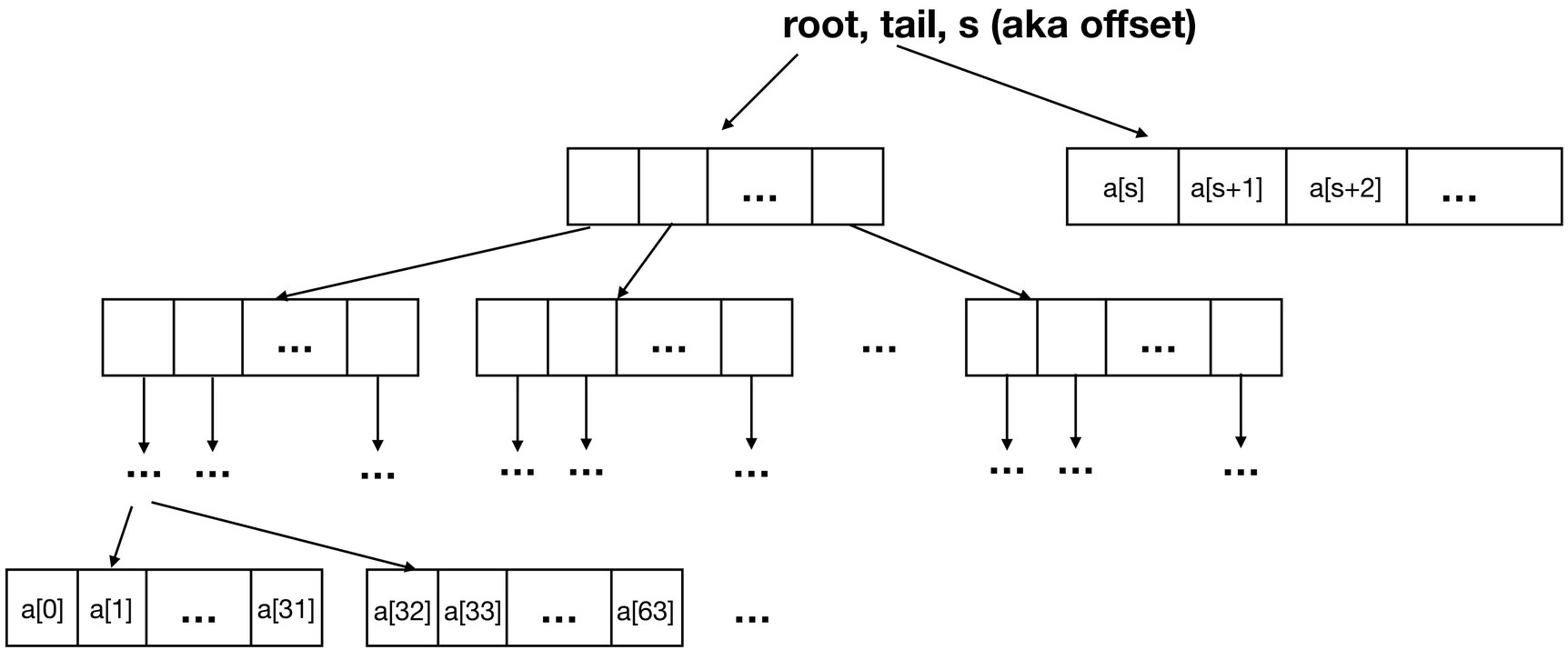
```
structure ParserState :=
  (stxStack : Array Syntax := #[])
  (pos      : String.Pos := 0)
  (cache    : ParserCache)
  (errorMsg : Option Error := none)
```

```
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 }
```

Comparison with linear/uniqueness types

- Values of types marked as linear/unique can be destructively updated.
- Compiler statically checks whether values are being used linearly or not.
- Pros: no runtime checks; compatible with tracing GCs.
- Cons: awkward to use; complicates a dependent type system even more.
- Big cons: all or nothing. A function *f* that takes non-shared values most of the time cannot perform destructive updates.

Persistent Arrays



Reusing big and small objects.
Persistent arrays will often be shared.

Moving forward

- Interactive compilation: a “tactic framework for guiding the compiler”
- Custom automation for the IMO grand challenge
- Moving the simplifier step in the compiler to Lean
- The new language manual
- Pushing/Extending RC (join work with Daan Leijen)
- Typed macro system (Sebastian Ullrich)
- Rust integration (Sebastian Ullrich)

Conclusion

- We implemented Lean4 in Lean.
- Users will be able and customize all modules of the 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
- Official release: Lean Together meeting in January 2021