past, present and future

Galois Inc., Oregon, August 2018

Leonardo de Moura
Microsoft Research

https://leanprover.github.io
Lean is a platform for **software verification** and **formalized mathematics**
Goals

• Proof stability
• Extensibility
• Expressivity - Dependent Type Theory
• Scalability
• de Bruijn’s principle: small trusted kernel, and 2 external type checkers

“Hack without fear”
Motivation: automated provers @ Microsoft

Testing

Software Verification

SAGE

Pex

Alive

Ivy

BOOGIE

Z3 Theorem Prover
Software verification & automated provers

• Easy to use for simple properties

• Main problems:
  • Scalability issues
  • Proof stability
  • Hard to control the behavior of automated provers

• in many verification projects:
  • Hyper-V
  • Ironclad & Ironfleet (https://github.com/Microsoft/Ironclad)
  • Everest (https://project-everest.github.io/)
Extend Lean using Lean

Metaprogramming
Domain specific automation
Domain specific languages
Whitebox automation

Access Lean internals using Lean
Simplifiers, decision procedures, type class resolution, type inference, unifiers, matchers, …
Applications

- IVy Metatheory (Ken McMillan - MSR Redmond)
- AliveInLean (Nuno Lopes - MSR Cambridge)
- Protocol Verification (Joe Hendrix, Joey Dodds, Ben Sherman, Ledah Casburn, Simon Hudon - Galois)
- Verified Machine Learning (Daniel Selsam - Stanford)
- SQL query equivalence (Shumo Chu et al - UW)
Applications (cont.)

• FormalAbstracts (Tom Hales - University of Pittsburgh)

• Lean Forward, Number Theory (Jasmin Blanchette - Vrije Universiteit)

• Mathlib (Mario Carneiro - CMU and Johannes Hölzl - Vrije Universiteit)

• Teaching
  • Logic and Automated Reasoning (Jeremy Avigad - CMU)
  • Programming Languages (Zach Tatlock - UW)
  • Foundations of Analysis (Kevin Buzzard - Imperial College)
Alive

Nuno Lopes, MSR Cambridge

Pre: isPowerOf2(C)
%s = shl C, %N
%q = zext %s
%r = udiv %x, %q

=>

%N2 = add %N, log2(C)
%N3 = zext %N2
%r = lshr %x, %N3

Re-implementation of Alive in Lean

Open source: https://github.com/Microsoft/AliveInLean

Pending issues:

• Using processes+pipes to communicate with Z3
• Simpler framework for specifying LLVM instructions
Lean Demo
Writing metaprograms/tactics/automation in Lean
Metaprogramming example

```plaintext
meta def find : expr → list expr → tactic expr
| e []        := failed
| e (h :: hs) :=
  do t ← infer_type h,
      (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
  do { ctx ← local_context,
       t ← target,
       h ← find t ctx,
       exact h }
  <|> fail "assumption tactic failed"

lemma simple (p q : Prop) (h1 : p) (h2 : q) : q :=
  by assumption
```
Reflecting expressions

inductive level
| zero : level
| succ : level → level
| max : level → level → level
| imax : level → level → level
| param : name → level
| mvar : name → level

inductive expr
| var : nat → expr
| lconst : name → name → expr
| mvar : name → expr → expr
| sort : level → expr
| const : name → list level → expr
| app : expr → expr → expr
| lam : name → binfo → expr → expr → expr → expr
| pi : name → binfo → expr → expr → expr → expr
| elet : name → expr → expr → expr → expr → expr

meta def num_args : expr → nat
| (app f a) := num_args f + 1
| e := 0
Quotations

\[
\text{example : } \text{true} \land \text{true} := \\
\text{by do apply } \text{`(and.intro trivial trivial) }
\]

\[
\text{example } (p : \text{Prop}) : p \to p \lor \text{false} := \\
\text{by do } e \leftarrow \text{intro } \text{`h, refine } \text{`(or.inl } \text{%e)}
\]

```
meta def \text{is.not} : \text{expr} \to \text{option expr}
| `(not \%a)               := \text{some a}
| `(\%a \to \text{false}) := \text{some a}
| _                       := \text{none}
```

```
meta def \text{is.not} : \text{expr} \to \text{option expr}
| (\text{app (const } `\text{not }_\) a) := \text{some a}
| (\text{pi } _ \_ a (\text{const } `\text{false }_)) := \text{some a}
| _                           := \text{none}
```
The tactic monad

```lean
meta inductive result (state : Type) (α : Type)
| success : α → state → result
| exception : option (unit → format) → option pos → state → result

meta def interaction_monad (state : Type) (α : Type) :=
state → result state α

meta def tactic := interaction_monad tactic_state
```
Extending the tactic state

```hs
def state_t (σ : Type) (m : Type → Type) [monad m] (α : Type) : Type :=
σ → m (α × σ)

meta constant smt_goal : Type
meta def smt_state := list smt_goal
meta def smt_tactic := state_t smt_state tactic

meta def eblast : smt_tactic unit := repeat (ematch; try close)

meta def collect_implied_eqs : tactic cc_state :=
focus $ using_smt $ do
  add_lemmas_from_facts, eblast,
  (done; return cc_state.mk) <|> to_cc_state
```
Superposition prover

- 2200 lines of code

```
example {α} [monoid α] [has_inv α]: (∀ x : α, x * x⁻¹ = 1) →
    ∀ x : α, x⁻¹ * x = 1 :=
by super with mul_assoc mul_one

meta structure prover_state :=
  (active passive : rb_map clause_id derived_clause)
  (newly Derived : list derived_clause) (prec : list expr)
  (locked : list locked_clause) (sat_solver : cdcl.state)
  ...
meta def prover := state_t prover_state tactic
```
structure dlist (α : Type u) :=
  (apply : list α → list α)
  (invariant : ∀ l, apply l = apply [] ++ l)

def to_list : dlist α → list α
  | ⟨xs, _⟩ := xs []

local notation `#`\text{max} := by abstract {intros, rsimp}

/\text{-- `O(1)` Append dlists --}/
protected def append : dlist α → dlist α → dlist α
  | ⟨xs, h₁⟩ ⟨ys, h₂⟩ := ⟨xs • ys, #⟩

instance : has_append (dlist α) :=
  ⟨dlist.append⟩
transfer tactic

• Developed by Johannes Hölzl (approx. 200 lines of code)

```lean
lemma to_list_append (l₁ l₂ : dlist α) : to_list (l₁ ++ l₂) = to_list l₁ ++ to_list l₂ :=
show to_list (dlist.append l₁ l₂) = to_list l₁ ++ to_list l₂, from
by cases l₁; cases l₂; simp; rsimp

protected def rel_dlist_list (d : dlist α) (l : list α) : Prop :=
to_list d = l

protected meta def transfer : tactic unit :=
  _root_.transfer.transfer ["relator.rel_forall_of_total, "dlist.rel_eq, "dlist.rel_empty,
  "dlist.rel_singleton, "dlist.rel_append, "dlist.rel_cons, "dlist.rel_concat]

example : ∀(a b c : dlist α), a ++ (b ++ c) = (a ++ b) ++ c :=
begin
  dlist.transfer,
  intros,
  simp
end
```

• We also use it to transfer results from nat to int.
Lean to SMT2

- Goal: translate a Lean local context, and goal into SMT2 query.
- Recognize fragment and translate to low-order logic (LOL).
- Logic supports some higher order features, is successively lowered to FOL, finally SMT2.

```
lemma n_gt_0
(a : nat) : a >= 0 :=
by z3
```

```
decl n : int {n >= 0}
assert (not (n >= 0))
```

```
(declare-const n Int)
(assert (>= n 0))
(assert (not (>= n 0)))
```
mutual inductive type, term
with type : Type
| bool : type    |
| int : type     |
| var : string → type |
| fn : list type → type → type |
| refinement : type → (string → term) → type |

meta structure context :=
(type_decl : rb_map string type)
(decls : rb_map string decl)
(assertions : list term)
meta def reflect_prop_formula' : expr → smt2_m lol.term
l `(¬ %P) := lol.term.not <$> (reflect_prop_formula' P)
l `(P = Q) := lol.term.equals <$>
   (reflect_prop_formula' P) <$>
   (reflect_prop_formula' Q)
l `(P ∧ Q) := lol.term.and <$>
   (reflect_prop_formula' P) <$>
   (reflect_prop_formula' Q)
l `(P ∨ Q) := lol.term.or <$>
   (reflect_prop_formula' P) <$>
   (reflect_prop_formula' Q)
l `(P < Q) := reflect_ordering lol.term.lt P Q
l ...
l `(true) := return $ lol.term.true
l `(false) := return $ lol.term.false
l e := ...
Coinductive predicates

- Developed by Johannes Hölzl (approx. 800 lines of code)
- Uses impredicativity of Prop
- No kernel extension is needed

```plaintext
coinductive all_stream {α : Type u} (s : set α) : stream α → Prop
| step {} : ∀{a : α} {ω : stream α}, a ∈ s → all_stream ω → all_stream (a :: ω)

coinductive alt_stream : stream bool → Prop
| tt_step : ∀{ω : stream bool}, alt_stream (ff :: ω) → alt_stream (tt :: ff :: ω)
| ff_step : ∀{ω : stream bool}, alt_stream (tt :: ω) → alt_stream (ff :: tt :: ω)
```
Ring solver

- Developed by Mario Carneiro (approx. 500 lines of code)
- [https://github.com/leanprover/mathlib/blob/master/tactic/ring.lean](https://github.com/leanprover/mathlib/blob/master/tactic/ring.lean)
- ring2 uses computational reflection

```lean
import tactic.ring

theorem ex1 (a b c d : int) : (a + 0 + b) * (c + d) = b*d + c*b + a * c + d * a := by ring

theorem ex2 (α : Type) [comm_ring α] (a b c d : α) | | : (a + 0 + b) * (c + d) = b*d + c*b + a * c + d * a := by ring
```
Fourier-Motzkin elimination

- Linear arithmetic inequalities
- Developed here
- [https://github.com/GaloisInc/lean-protocol-support/tree/master/galois/arith](https://github.com/GaloisInc/lean-protocol-support/tree/master/galois/arith)
Lean 3.x limitations

- Lean programs are compiled into byte code
- Lean expressions are foreign objects in the Lean VM
- 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
- Users cannot extend the equation compiler (e.g., support for quotient types)
Lean 4

- Leo and Sebastian Ullrich (and soon Gabriel Ebner)

- Implement Lean in Lean
  - parser, elaborator, equation compiler, code generator, tactic framework and formatter

- New intermediate representation (defined in Lean) can be translated into C++ (and LLVM IR)

- Only runtime, kernel and basic primitives are implemented in C++

- Users may want to try to prove parts of the Lean code generator or implement their own kernel in Lean

- Foreign function interface (invoke external tools)
Lean 4 architecture

- Server
- VS Code & Emacs
- Tactic Framework
- Parser & Macro Expander
- Equation Compiler
- Elaborator
- Elaboration primitives
- Compiler
- Kernel
- Runtime / Interpreter
- IR
- C / LLVM backends
Parser

- Implemented in Lean
- Fully extensible
- Design your own domain specific language
- Error recovery, documentation, printer, ... for free

```lean
@[irreducible, derive monad alternative monad_reader monad_state monad_parsec monad_except]
def read_m := rec_t syntax $ reader_t reader_config $ state_t reader_state $ parsec syntax

structure reader :=
(read : read_m syntax)
(tokens : list token_config := [])
```

```lean
def open_export.reader : reader :=
[ident,
 ["as", ident]?,
 [try ["(" , ident], ident*, ")"]?,
 [try ["(" , "renaming"] , [ident, "->" , ident]+ , ")"]?,
 ["(" , "hiding" , ident+ , ")"]?]
]+

def open.reader : reader :=
node «open» ["open", open_export.reader]
```
Syntax Objects

structure syntax_ident :=
(info : option source_info) (name : name) (msc : option macro_scope_id) (res : option resolved)

inductive atomic_val
| string (s : string)
| name (n : name)

structure syntax_atom :=
(info : option source_info) (val : atomic_val)

structure syntax_node (syntax : Type) :=
(macro : name) (args : list syntax)

inductive syntax
| ident (val : syntax_ident)

/* any non-ident atom */
| atom (val : syntax_atom)
| node (val : syntax_node syntax)

Macros can be expanded and/or elaborated.
Users can define new readers and macros.
Kernel expressions

Elaborator converts syntax objects into expressions.

```
inductive expr
| bvar : nat → expr              -- bound variables
| fvar : name → expr             -- free variables
| mvar : name → expr → expr      -- (temporary) meta variables
| sort : level → expr            -- Sort
| const : name → list level → expr -- constants
| app : expr → expr → expr       -- application
| lam : name → binder_info → expr → expr → expr -- lambda abstraction
| pi : name → binder_info → expr → expr → expr -- Pi
| elet : name → expr → expr → expr → expr -- let expressions
| lit : literal → expr           -- literals
| mdata : kvmap → expr → expr    -- metadata
| proj : nat → expr → expr       -- projection
```
Compiler - code generator

- Implemented Lean
- External contributors can prove the new compiler is correct
- Code specialization and monomorphization
- Target is the new IR also defined in Lean
- Users can select theorems as optimization rules

@simp lemma map_map (g : β → γ) (f : α → β) (l : list α) : map g (map f l) = map (g ∘ f) l :=
by induction l; simp [*]
Runtime

Strict, GC based on reference counting, destructive updates for unshared objects, support for unboxed values.

/* IR Instructions */

inductive instr
| assign (x : var) (ty : type) (y : var) -- x : ty := y
| assign_lit (x : var) (ty : type) (lit : literal) -- x : ty := lit
| assign_unop (x : var) (ty : type) (op : assign_unop) (y : var) -- x : ty := op y
| assign_binop (x : var) (ty : type) (op : assign_binop) (y z : var) -- x : ty := op y z
| unop (op : unop) (x : var) -- op x

/* Constructor objects */

| cnstr (o : var) (tag : tag) (nobjs : uint16) (ssz : usize) -- Create constructor object
| set (o : var) (i : uint16) (x : var) -- Set object field: set o i x
| get (x : var) (o : var) (i : uint16) -- Get object field: x := get o i
| sset (o : var) (d : usize) (v : var) -- Set scalar field: sset o d v
| sget (x : var) (ty : type) (o : var) (d : usize) -- Get scalar field: x : ty := sget o d

/* Closures */

| closure (x : var) (f : fnid) (ys : list var) -- Create closure: x := closure f ys
| apply (x : var) (ys : list var) -- Apply closure: x := apply ys

/* Arrays */

| array (a sz c : var) -- Create array of objects with size `sz` and capacity `c`
| sarray (a : var) (ty : type) (sz c : var) -- Create scalar array
| array_write (a i v : var) -- (scalar) Array write

/inductive unop
| inc_ref | dec_ref | dec_sref | inc | dec
| free | dealloc
| array_pop | sarray_pop

/inductive assign_unop
| not | neg | ineg | nat2int | is_scalar | is_shared | is_null | cast | box | unbox
| array_copy | sarray_copy | array_size | sarray_size | string_len
| succ | tag | tag_ref
Code generation hints

- Support for low-level tricks used in SMT and ATP. Example: pointer equality

```python
def use_ptr_eq {α : Type u} {a b : α}
  (c : unit → {r : bool // a = b → r = tt})
  : {r : bool // a = b → r = tt} :=
c ()
```

Given `@use_ptr_eq _ a b c`, compiler generates

```c
if (addr_of(a) == addr_of(b)) return true;
else return c();
```
Structured trace messages

- Why did my tactic/solver fail?
- Lean 3 has support for trace messages, but they are just a bunch of strings.
- Lean 4 will provide structured trace messages and APIs for browsing them.
- Traces will be generated on demand (improved discoverability).

```lean
inductive trace
| mk (msg : message) (subtraces : list trace)

def trace_map := rbmap pos trace (<)

structure trace_state :=
  (opts : options)
  (roots : trace_map)
  (cur_pos : option pos)
  (cur_traces : list trace)

def trace_t (m : Type → Type u) := state_t trace_state m

class monad_tracer (m : Type → Type u) :=
  (trace_root {α} : pos → name → message → thunk (m α) → m α)
  (trace_ctx {α} : name → message → thunk (m α) → m α)
```
Better support for proofs by reflection

- Define an inductive datatype `form` that captures a class of formulas.

- Implement a decision procedure `dec_proc` for this class.

- Prove: \( \forall (s : \text{form}) \ ctx, \ dec\_proc \ s = \text{tt} \rightarrow \text{denote} \ s \ ctx \)

- The type checker has to reduce `(dec_proc s)`. This is too inefficient in Lean 3.

- In Lean 4, we allow users to use the compiler + IR interpreter to reduce `(dec_proc s)`.

- We still need to use the symbolic reduction engine to show that the current goal and `(denote s ctx)` are definitionally equal.

- Disadvantages: increases the size of the TCB, external type checkers will probably timeout in proofs using this feature.
New application scenarios
Automated reasoning framework

- Many users use Python + SMT solver to developing automated reasoning engines (e.g., Alive is implemented in Z3Py).

- Lean 3 interpreter is already faster than Python.

- FFI in Lean 4 will provide (efficient) access to external SAT & SMT solvers and ATP.

- Many goodies not available in the Python + SMT framework:
  - Simplifiers.
  - Efficient symbolic simulation.
  - Custom automation.
  - Parsing framework + integration with IDEs (VS Code, Emacs).
Domain Specific Languages

- Users can define and reason about their DSLs.

- Code reuse:
  - Compiler infrastructure.
  - Parsing framework.
  - Elaborator.
  - IDE integration.
Lean as a general purpose programming language

- Lean is an extensible system: parser, elaborator, compiler, etc.
- User certified optimizations as conditional rewriting rules.
- New backends for the Lean 4 IR can be implemented in Lean.
- Foreign function interface.
- leanpkg - package management tool implemented in Lean.
Conclusion

- Users can create their own automation, extend and customize Lean
- Domain specific automation
- Internal data structures and procedures are exposed to users
- Whitebox automation
- Lean 4 automation written in Lean will be much more efficient
- Lean 4 will be more extensible
- New application domains
  - Lean 4 as a more powerful Z3Py
  - Lean 4 as a platform for developing domain specific languages