The Lean theorem prover, for mathematicians

Robert Y. Lewis

Carnegie Mellon University
`http://andrew.cmu.edu/user/rlewis1`
`http://leanprover.github.io`

December 1, 2017

Slide credit

Some slides have been borrowed from Jeremy Avigad and Leonardo de Moura – thanks!

Outline

I want to make the following points:

- Formalizing mathematics is difficult but possible.
- There are many plausible foundations for formalized mathematics.
- Dependent type theory is one good foundational choice.
    - It is a particularly elegant and expressive language.
    - It is convenient for formalizing abstract algebraic notions.
    - It can express mathematical *processes* or *techniques* alongside *theories*.
- Formalizing mathematics is easier when we have access to common processes and techniques.

TOC

Background: Lean

Lean is a new interactive theorem prover, developed principally by Leonardo de Moura at Microsoft Research, Redmond.

Calculus of constructions with:

- Non-cumulative hierarchy of universes
- Impredicative `Prop`
- Quotient types and propositional extensionality
- Axiom of choice available

See `http://leanprover.github.io`

Contributors

Code base: Leonardo de Moura, Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Daniel Selsam

Libraries: Jeremy Avigad, Floris van Doorn, Leonardo de Moura, Robert Lewis, Gabriel Ebner, Johannes Hölzl, Mario Carneiro

Past project members: Soonho Kong, Jakob von Raumer

Lean mathematics library

The standard mathematics library contains:

- Basic algebraic structures
- Number structures through $\mathbb{C}$
- Programming data structures
- Real analysis, measure theory
- Set theory
- Tactics for reasoning with the above
- …and more to come

`https://github.com/leanprover/mathlib`

Encoding proofs

```
theorem sqrt_two_irrational {a b : ℕ} (co : coprime a b) :
  a^2 ≠ 2 * b^2 :=
assume H : a^2 = 2 * b^2,
have even (a^2),
  from even_of_exists (exists.intro H),
have even a,
  from even_of_even_pow this,
obtain (c : ℕ) (aeq : a = 2 * c),
  from exists_of_even this,
have 2 * (2 * c^2) = 2 * b^2,
  by rewrite [-H, aeq, *pow_two, mul.assoc, mul.left_comm c],
have 2 * c^2 = b^2,
  from eq_of_mul_eq_mul_left dec_trivial this,
have even (b^2),
  from even_of_exists (exists.intro _ (eq.symm this)),
have even b,
  from even_of_even_pow this,
assert 2 ∣ gcd a b,
  from dvd_gcd (dvd_of_even ‘even a‘) (dvd_of_even ‘even b‘),
have 2 ∣ 1,
  by rewrite [gcd_eq_one_of_coprime co at this]; exact this,
show false,
  from absurd ‘2 ∣ 1‘ dec_trivial
```

One language fits all

In simple type theory, we distinguish between

- types
- terms
- propositions
- proofs

Dependent type theory is flexible enough to encode them all in the same language.

It can also encode *programs*, since terms have computational meaning.

Lean as a programming language

Think of `+` as a program. An expression like `12 + 45` will *reduce* or *evaluate* to `57`.

But `+` is defined as unary addition – inefficient!

Lean implements a virtual machine which performs fast, untrusted evaluation of Lean expressions.

Lean as a programming language

There are algebraic structures that provides an interface to terminal and file I/O.

Lean's built-in package manager is implemented entirely in Lean.

Lean as a programming language

Definitions tagged with `meta` are “VM only,” and allow unchecked recursive calls.

```
meta def f : ℕ → ℕ
| n := if n=1 then 1
       else if n%2=0 then f (n/2)
       else f (3*n + 1)

#eval (list.iota 1000).map f
```

Metaprogramming in Lean

Question: How can one go about writing tactics and automation?

Various answers:

- Use the underlying implementation language (ML, OCaml, C++, . . .).
- Use a domain-specific tactic language (LTac, MTac, Eisbach, . . .).
- Use reflection (RTac).

Metaprogramming in Lean

Lean's answer: go meta, and use Lean itself.

(MTac, Idris, and now Agda do the same, with variations.)

Advantages:

- Users don't have to learn a new programming language.
- The entire library is available.
- Users can use the same infrastructure (debugger, profiler, etc.).
- Users develop metaprograms in the same interactive environment.
- Theories and supporting automation can be developed side-by-side.

Metaprogramming in Lean

The method:

- Add an extra (meta) constant: `tactic_state`.
- Reflect expressions with an `expr` type.
- Add (meta) constants for operations which act on the tactic state and expressions.
- Have the virtual machine bind these to the internal representations.
- Use a tactic monad to support an imperative style.

Definitions which use these constants are clearly marked `meta`, but they otherwise look just like ordinary definitions.

Metaprogramming in Lean

```
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
```

Metaprogramming in Lean

```
meta def p_not_p : list expr → list expr → tactic unit
| []         Hs := failed
| (H1 :: Rs) Hs :=
  do t ← infer_type H1,
     (do a   ← match_not t,
         H2  ← find_same_type a Hs,
         tgt ← target,
         pr  ← mk_app ‘absurd [tgt, H2, H1],
         exact pr)
     <|> p_not_p Rs Hs

meta def contradiction : tactic unit :=
do ctx ← local_context,
   p_not_p ctx ctx

lemma simple (p q : Prop) (h₁ : p) (h₂ : ¬p) : q :=
by contradiction
```

TOC

CAS and ITP

CAS strengths:

- Easy and useful
- Instant gratification
- Interactive use, exploration
- Pogrammable and extensible

CAS weaknesses:

- Focus on symbolic computation, not abstract definitions and assertions
- Not designed for reasoning
- Murky or nonexistent semantics

ITP strengths:

- Languages are expressive and well-specified
- Precise semantics
- Results are fully verified

ITP weaknesses:

- Formalization is slow
- It requires a high degree of commitment and expertise
- It doesn't promote exploration and discovery

ITP + CAS

By linking the two, we can

- Allow exploration and computation in the proof assistant, without reimplementing algorithms
- Lower the barrier for newcomers to ITP
- Loan a semantics/proof language to CAS

Many projects have attempted to connect the two: verified CAS algorithms, trusting links, verified links, ephemeral links, CAS proof languages.

Connecting Lean and Mathematica

- An extensible procedure to interpret Lean in Mathematica
- An extensible procedure to interpret Mathematica in Lean
- A link allowing Lean to evaluate arbitrary Mathematica commands, and receive the results
- Tactics for certifying results of particular Mathematica computations
- A link allowing Mathematica to execute Lean tactics and receive the results

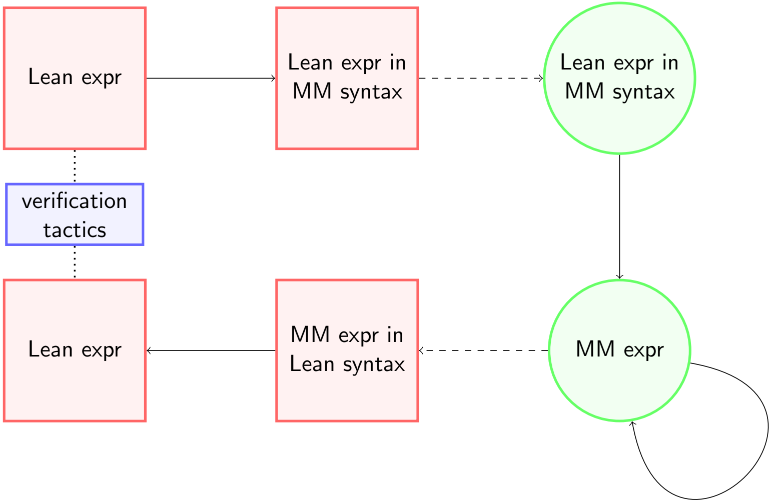
Connecting Lean and Mathematica

The idea: many declarations in Lean correspond *roughly* to declarations in Mathematica.

We can do an approximate translation back and forth and verify post hoc that the result is as expected.

Correspondences, translation rules, checking procedures are part of a mathematical theory.

Link architecture



Example: factoring polynomials

```
@[translation_rule]
meta def add_to_pexpr : app_trans_pexpr_keyed_rule :=
⟨"Plus",
 λ env args,
 do args' ← list.mfor args (pexpr_of_mmexpr env),
    return $ pexpr_fold_op ```(0) ```(has_add.add) args'⟩

meta def assert_factor (e : expr) (nm : name) : tactic unit :=
do fe ← factor e,
   pf ← eq_by_simp e fe,
   note nm pf

example (x : ℝ) : 1 - 2*x + 3*x^2 - 2*x^3 + x^4 ≥ 0 :=
begin
 assert_factor 1 - 2*x + 3*x^2 - 2*x^3 + x^4 using h,
 rewrite h,
 apply sq_nonneg
end
```

Example: sanity checking

Many computations in Mathematica are not easily certifiable, but can still be useful in interactive proofs.

`sanity_check` runs the Mathematica command `FindInstance` to search for an assignment satisfying the hypotheses and the negation of the goal. The tactic fails if an assignment is found.

```
example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0) :
      x = 0 :=
by sanity_check; admit

example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0)
      (h3 : -pi < x ∧ x < pi) : x = 0 :=
by sanity_check; admit
```

Example: sanity checking

```
meta def sanity_check_aux (hs : list expr) (xs : list
    expr) : tactic unit :=
do t ← target,
   nt ← to_expr ‘‘‘(¬ %%t),
   hs’’ ← monad.foldl (λ a b, to_expr ‘‘‘(%%a ∧ %%b))
    ‘(true) (nt::hs),
   l ← run_command_on_list
     (λ e,
    "With[{ls=Map[Activate[LeanForm[#]]&,"++e++"]},
    Length[FindInstance[ls[[1]], Drop[ls, 1]]]]")
     (hs’’::xs),
   n ← to_expr ‘‘‘(%%l : ℕ) >>= eval_expr ℕ,
   if n > 0 then
     fail "sanity check: the negation of the goal is
    consistent with hypotheses"
   else skip
```

Examples

We can use this technique for:

- factoring (numbers, polynomials, matrices)
- linear arithmetic
- computing integrals/antiderivatives
- numeric approximations
- unverified simplification
- guiding tactics
- ...

TOC

Nonlinear inequalities

$$\begin{gathered} 0 < x < y,\ u < v \\ \Longrightarrow \\ 2u + \exp(1 + x + x^4) < 2v + \exp(1 + y + y^4) \end{gathered}$$

- This inference is not contained in linear arithmetic or real closed fields.
- This inference is tight: symbolic or numeric approximations to exp are not useful.
- Backchaining using monotonicity properties suggests many equally plausible subgoals.
- But, the inference is completely straightforward.

A new method

We propose and implement a method based on this type of heuristically guided forward reasoning. Our method:

- Verifies inequalities on which other procedures fail.
- Can produce fairly direct proof terms.
- Captures natural, human-like inferences.
- Performs well on real-life problems.

- Is not complete.
- Is not guaranteed to terminate.

Polya: modules and database

Any comparison between canonical terms can be expressed as $t_i \bowtie 0$ or $t_i \bowtie c \cdot t_j$, where $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$. This is in the common language of addition and multiplication.

A central database (the blackboard) stores term definitions and comparisons of this form.

Modules use this information to learn and assert new comparisons.

The procedure has succeeded in verifying an implication when modules assert contradictory information.

Polya data types

```
meta structure blackboard : Type :=
(ineqs : hash_map (expr×expr) (λ p, ineq_info p.1 p.2))
(diseqs : hash_map (expr×expr) (λ p, diseq_info p.1 p.2))
(signs : hash_map expr sign_info)
(exprs : rb_set (expr × expr_form))
(contr : contrad)
(changed : bool := ff)
```

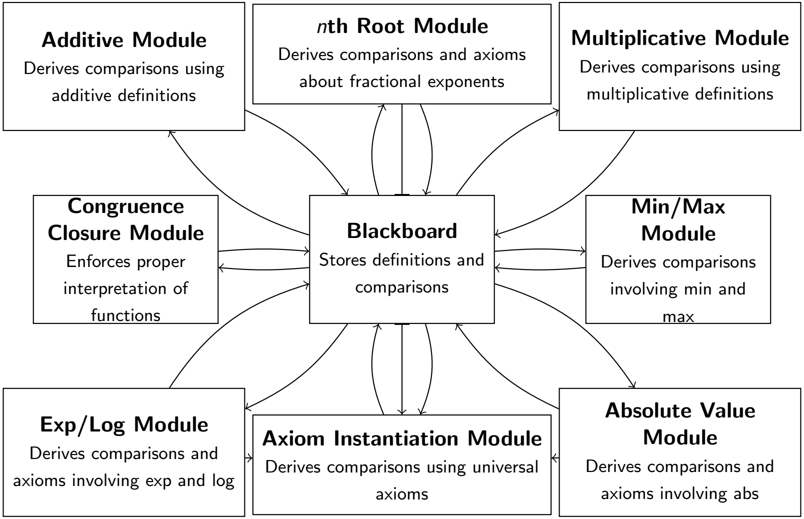
Polya: producing proof terms

Every piece of information asserted to the blackboard must be tagged with a *justification*.

We define a datatype of justifications in Lean, and a metaprogram that will convert a justification into a proof term.

```
meta inductive contrad
| none : contrad
| eq_diseq : Π {lhs rhs}, eq_data lhs rhs → diseq_data
    lhs rhs → contrad
| ineqs : Π {lhs rhs}, ineq_info lhs rhs → ineq_data lhs
    rhs → contrad
| sign : Π {e}, sign_data e → sign_data e → contrad
| strict_ineq_self : Π {e}, ineq_data e e → contrad
| sum_form : Π {sfc}, sum_form_proof sfc → contrad
```

Polya: computational structure



Additive Module
Derives comparisons using additive definitions
***n*th Root Module**
Derives comparisons and axioms about fractional exponents
Multiplicative Module
Derives comparisons using multiplicative definitions
Congruence Closure Module
Enforces proper interpretation of functions
Blackboard
Stores definitions and comparisons
Min/Max Module
Derives comparisons involving min and max
Exp/Log Module
Derives comparisons and axioms involving exp and log
Axiom Instantiation Module
Derives comparisons using universal axioms
Absolute Value Module
Derives comparisons and axioms involving abs

Theory modules

Each module looks specifically at terms with a certain structure. E.g. a trigonometric module looks only at applications of `sin`, `cos`, etc.

Theory modules can be developed alongside the mathematical theory. Intuition: “when I see a term of this shape, this is what I immediately know about it, and why.”

Modules can interact with other computational processes, e.g. Mathematica.

Conclusions

- Dependent type theory is a powerful language for formalizing mathematics.
- It is also a powerful programming language.
- The line between these is blurry: mathematical reasoning techniques are part of mathematics.
- We can create a more powerful proof assistant by combining the two.

Thanks for listening!