Two Tools for Formalizing Mathematical Proofs

Robert Y. Lewis

Carnegie Mellon University

January 5, 2018

Motivation

Formalizing mathematics is hard.

How can we make it easier?

TOC

Interactive theorem proving

We want a language (and an implementation of this language) for:

- Defining mathematical objects
- Stating properties of these objects
- Showing that these properties hold
- Checking that these proofs are correct
- Automatically generating these proofs

This language should be:

- Expressive
- User-friendly
- Computationally efficient

Challenges in ITP

Formalizing mathematics has many challenges:

- It often requires expert knowledge of both the mathematics and the logic.
- Natural language mathematics is often vague or ambiguous.
- "Trivial" proof steps are not necessarily trivial to a computer.
- Processes and techniques that are informally valid—e.g. diagrams, computer programs—can be difficult to formally justify.

Contributions

This dissertation contributes two tools for the Lean proof assistant that target these latter points.

- A connection between Lean and Mathematica
- A method for proving nonlinear inequalities

Broader ideas that connect these tools: partial formalization and the formalization of mathematical process.

Partial formalization

Traditionally, formalization projects have targeted "gapless" proofs.

This makes proofs much harder to finish, and automation much harder to write.

The tools in this dissertation allow partial formalization, to increase speed and/or scope.

Formalizing mathematical processes

There is more to mathematics than its raw output: methods of reasoning, heuristics, metamathematical info.

Lean's *metaprogramming* framework allows these to be expressed in the same language and same environment as definitions, theorems, proofs.

New theories can describe how to incorporate themselves into existing tools.

TOC

Background: Lean

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

Calculus of inductive constructions with:

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

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

Some slides in this section are borrowed from Jeremy Avigad and Leonardo de Moura — thanks!

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

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?

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

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

```
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
```

TOC

CAS and ITP

CAS strengths:

- Easy and useful
- Instant gratification
- Interactive use, exploration
- Programmable 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

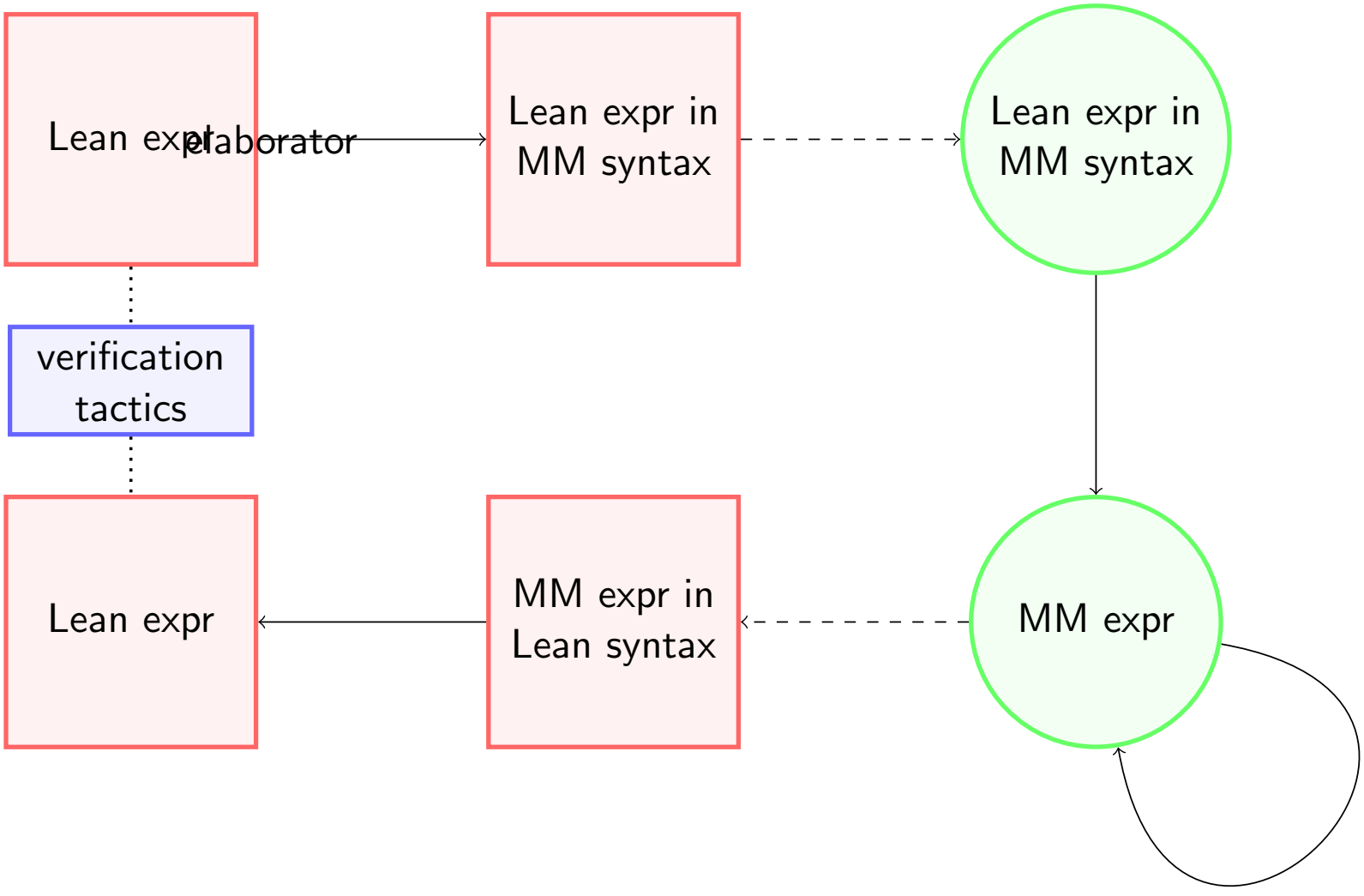
Connecting Lean and Mathematica

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

$$0 < x < y,\ u < v$$
$$\Longrightarrow$$
$$2u + \exp(1 + x + x^4) < 2v + \exp(1 + y + y^4)$$

- 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.

Implementations

A prototype version of this system was implemented in Python [1] [2].

For this dissertation, the algorithm has been redesigned to produce proof terms, and has been implemented in Lean.

[1] Avigad, Lewis, and Roux. *A heuristic prover for real inequalities*. Journal of Automated Reasoning, 2016

[2] Lewis. *Polya: a heuristic procedure for reasoning with real inequalities*. MS thesis

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 : ∏ {lhs rhs}, eq_data lhs rhs → diseq_data
    lhs rhs → contrad
| ineqs : ∏ {lhs rhs}, ineq_info lhs rhs → ineq_data lhs
    rhs → contrad
| sign : ∏ {e}, sign_data e → sign_data e → contrad
| strict_ineq_self : ∏ {e}, ineq_data e e → contrad
| sum_form : ∏ {sfc}, sum_form_proof sfc → contrad
```

Polya: producing proof terms

```
meta inductive ineq_proof  : expr → expr → ineq → Type
meta inductive eq_proof    : expr → expr → ℚ → Type
meta inductive diseq_proof : expr → expr → ℚ → Type
meta inductive sign_proof  : expr → gen_comp → Type

#check ineq_proof.adhoc
/-
ineq_proof.adhoc : Π (lhs rhs : expr) (i : ineq),
   tactic expr → ineq_proof lhs rhs i
-/
```

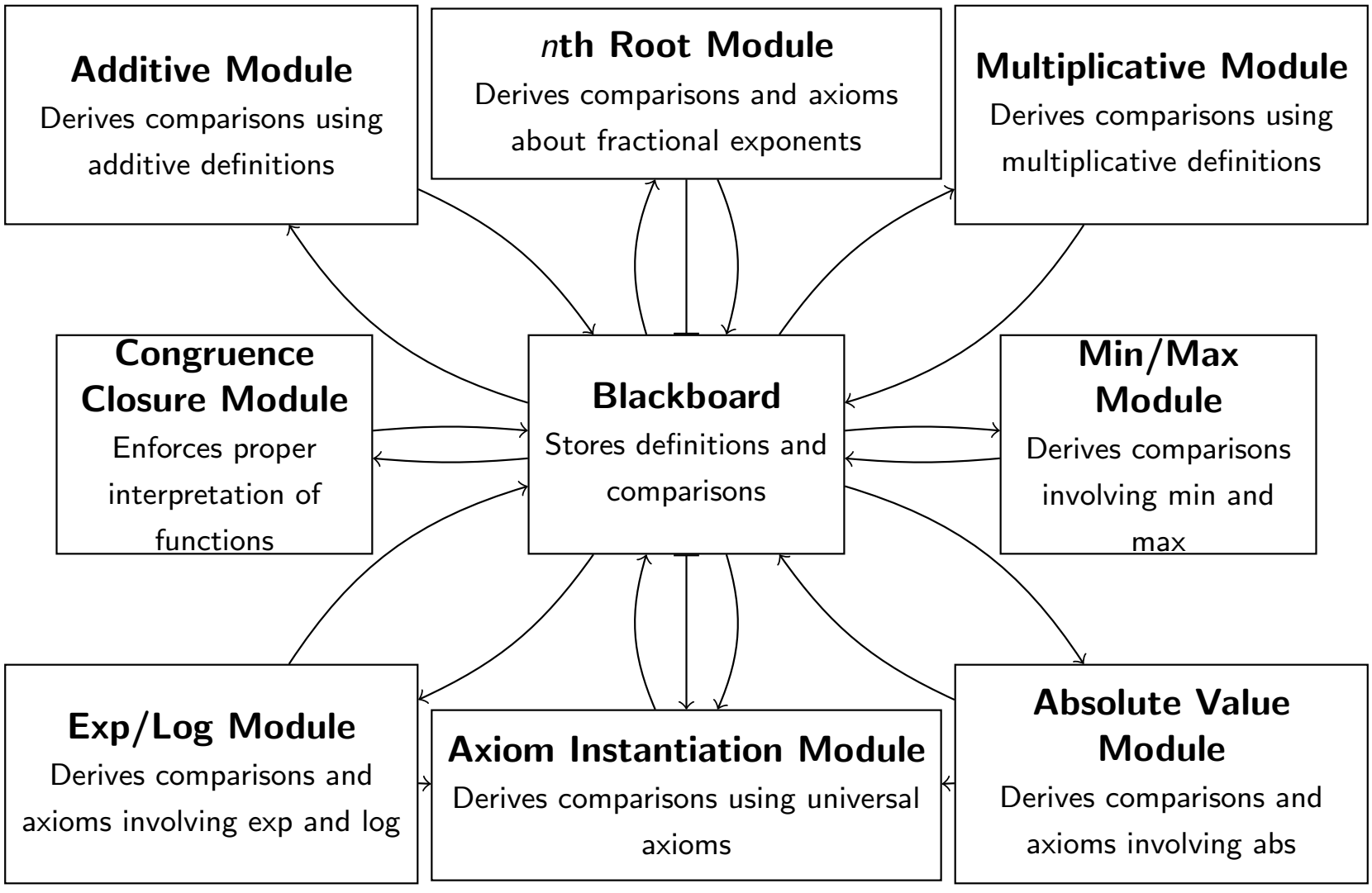
Polya: producing proof terms

Proof terms are assembled by traversing the proof trace tree.

Some steps, mostly related to normalization of algebraic terms, are currently axiomatized.

This architecture separates *search* from *reconstruction*.

Polya: computational structure



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.

Currently implemented in the Lean version: additive and multiplicative arithmetic modules.

Examples

```
example
  (h1 : u > 0) (h2 : u < v) (h3 : z > 0) (h4 : z + 1 < w)
  (h5 : (u + v + z)^3 ≥ (u + v + w + 1)^5) : false :=
by polya h1 h2 h3 h4 h5

example
  (h1 : x > 0) (h2 : x < 3*y) (h3 : u < v) (h4 : v < 0)
  (h5 : 1 < v^2) (h6 : v^2 < x)
  (h7 : u*(3*y)^2 + 1 ≥ x^2*v + x) : false :=
by polya h1 h2 h3 h4 h5 h6
```

Proof sketches

```
example  (h1 : x > 0) (h2 : x < 1*1)
  (h3 : (1 + (-1)*x)^(-1) ≤ (1 + (-1)*x^2)^(-1)) : false

/-
false : contradictory inequalities
  1 ≤ 1*x^2 : by multiplicative arithmetic
    x^2 ≥ 1*x : by linear arithmetic
      1 * 1 + (-1) * x^2 ≤ 1*1 * 1 + (-1) * x
        : by multiplicative arithmetic
        (1 * 1 + (-1) * x)^-1 ≤ 1*(1 * 1 + (-1) * x^2)^-1 : hypothesis
        1 = 1 * ((1 * 1 + (-1) * x)^-1^-1 * (1 * 1 + (-1) * x)^-1)
          : by definition
        1 = 1 * ((1 * 1 + (-1) * x^2)^-1^-1 * (1 * 1 + (-1) * x^2)^-1)
          : by definition
    1 = 1 * (x^2^-1 * x^2) : by definition
  1 > 1*x^2 : by multiplicative arithmetic
    1 = 1 * (x^2^-1 * x^2) : by definition
    1 <1 * x^-1 : rearranging
      x < 1*1 : hypothesis
  x^2 > 0 : inferred from other sign data
-/
```

KeYmaera benchmarks

We have tested Polya on a collection of 4442 benchmark problems generated by KeYmaera, a verification tool for hybrid systems. With a three-second timeout:

- Python version of Polya: 96%, 6 minutes
- Lean version of Polya: 72%, 90 minutes
- Mathematica: 99%, 120 minutes

Conclusion

- Lean's metaprogramming framework allows us to develop theories and automation in sync.
- The automation can be an essential part of a theory.
- Tools that accomplish "standard" mathematical tasks will help encourage mathematicians to use proof assistants.
- Lean's metaprogramming framework is powerful enough to implement these tools.

Conclusion

Thanks for listening!