A heuristic method for formally verifying real inequalities

Robert Y. Lewis

Vrije Universiteit Amsterdam

June 27, 2018

Motivation

Specific topic for today: verifying systems of nonlinear inequalities over $\mathbb{R}$

$$0 \leq n,\ n < (K/2)x,\ 0 < C,\ 0 < \varepsilon < 1 \models \left(1 + \frac{\varepsilon}{3(C+3)}\right) \cdot n < Kx$$

$$0 < x < y \models (1 + x^2)/(2 + y)^{17} < (1 + y^2)/(2 + x)^{10}$$

$$0 < x < y \models (1 + x^2)/(2 + exp(y)) \geq (2 + y^2)/(1 + exp(x))$$

Motivation

More general theme: connections between automation and formalization in the development of formal libraries

Motivation

Post hoc view: mathematics is a collection of definitions, theorems, and proofs. (Maybe algorithms with proofs of correctness.)

In practice: mathematics includes methods of reasoning, heuristics, metamathematical info.

Formalization focuses on the former.

Motivation

Can a formal library describe methods, processes, techniques?

Can these processes be used in the formal library?

What sort of language is needed to describe them?

LEAN

THEOREM PROVER

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

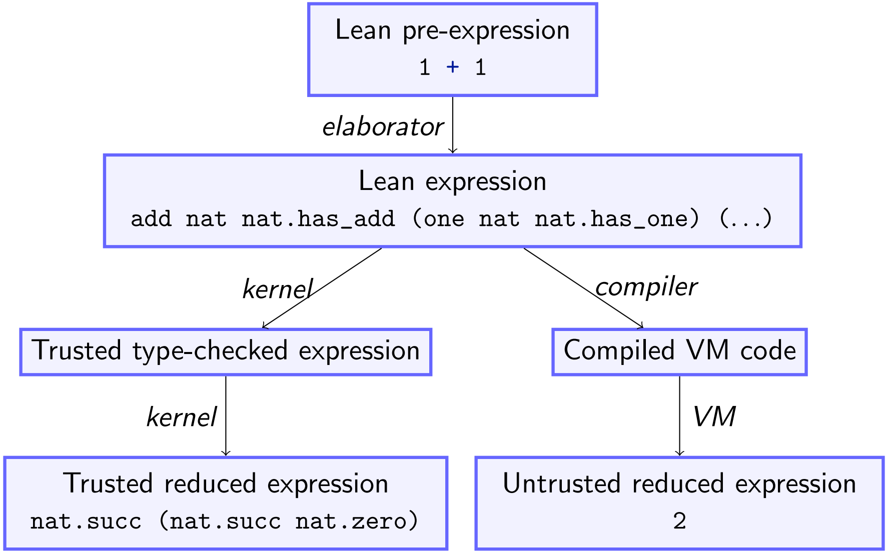
Think of `+` as a program. An expression like `3+5` will *reduce* or *evaluate* to `8`.

But:

- `3` is defined as `succ(succ(succ(zero)))`
- `+` is defined as unary addition

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

Expression evaluation



Lean pre-expression
1 + 1
elaborator
Lean expression
add nat nat.has_add (one nat nat.has_one) (...)
kernel
compiler
Trusted type-checked expression
Compiled VM code
kernel
VM
Trusted reduced expression
nat.succ (nat.succ nat.zero)
Untrusted reduced expression
2

The Lean VM

- The VM can evaluate anything in the Lean library, as long as it is not `noncomputable`.
- It substitutes native nats, ints, arrays.
- It has a profiler and debugger.
- The VM is ideal for non-trusted execution of code.

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.

Ebner, Ullrich, Roesch, Avigad, de Moura. *A Metaprogramming Framework for Formal Verification*, ICFP 2017.

Metaprogramming in Lean

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 strategy: expose internal data structures as `meta` declarations, and insert these internal structures during evaluation.

```
meta constant expr : Type
meta constant environment : Type
meta constant tactic_state : Type
meta constant to_expr : expr → tactic expr
```

Tactic proofs

```
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
```

Nonlinear inequalities

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

- 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]

The algorithm has been redesigned to produce proof terms, and has been implemented in Lean.[2]

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

[2] Lewis. *Two Tools for Formalizing Mathematical Proofs*. Dissertation, 2018.

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) ineq_info)
(diseqs : hash_map (expr×expr) diseq_info)
(signs : hash_map expr sign_info)
(exprs : rb_set (expr × expr_form))
(contr : contrad)
(changed : bool)
```

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: 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
-/
```

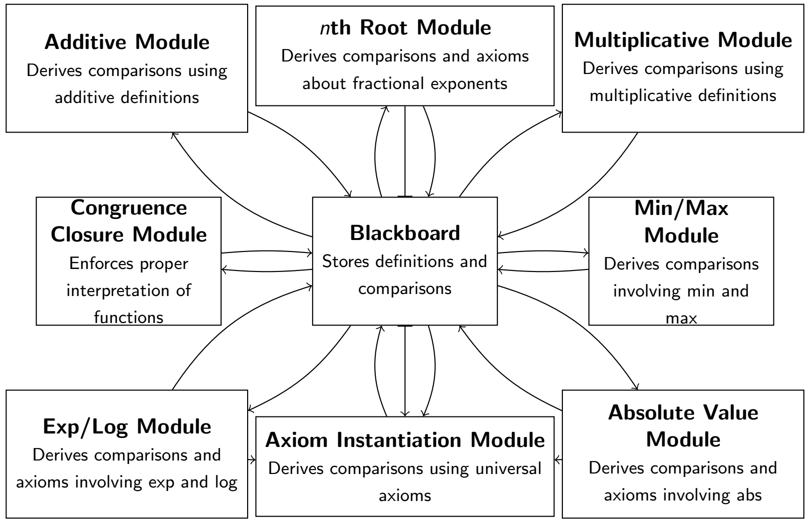
Polya: producing proof terms

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: compositional 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 (possibly external) computational processes.

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

Additive module

Given additive equations $\{t_i = \sum c_j \cdot t_{k_j}\}$ and atomic comparisons $\{t_i \bowtie c \cdot t_j\}$ and $\{t_i \bowtie 0\}$, produce a list of new comparisons, with justifications.

Method: Fourier-Motzkin elimination.

Multiplicative arithmetic can be handled similarly. (But: minor challenges for proof production.)

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, eliminate t_3:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned}$$

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, eliminate t_3:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned} \quad \Longrightarrow \quad 7t_1 + 3t_2 > 0$$

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, eliminate t_3:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 7t_1 + 3t_2 &> 0 \\ 10t_1 + t_2 &\geq 0 \end{aligned}$$

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, eliminate t_3:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned} \implies \begin{aligned} 7t_1 + 3t_2 &> 0 \\ 10t_1 + t_2 &\geq 0 \\ 4t_1 - t_2 &> 0 \end{aligned}$$

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, find the strongest pair:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned} \implies \begin{aligned} 7t_1 + 3t_2 &> 0 \\ 10t_1 + t_2 &\geq 0 \\ 4t_1 - t_2 &> 0 \end{aligned} \implies \begin{aligned} t_1 &> -\frac{3}{7}t_2 \\ t_1 &\geq -\frac{1}{10}t_2 \\ t_1 &> \frac{1}{4}t_2 \end{aligned}$$

Fourier-Motzkin additive module

To find comparisons between t_1 and t_2, find the strongest pair:

$$\begin{aligned} 3t_1 + 2t_2 - t_3 &> 0 \\ 4t_1 + t_2 + t_3 &\geq 0 \\ 2t_1 - t_2 - 2t_3 &\geq 0 \\ -2t_2 - t_3 &> 0 \end{aligned} \implies \begin{aligned} 7t_1 + 3t_2 &> 0 \\ 10t_1 + t_2 &\geq 0 \\ 4t_1 - t_2 &> 0 \end{aligned} \implies \begin{aligned} t_1 &> -\frac{3}{7}t_2 \\ t_1 &\geq -\frac{1}{10}t_2 \\ t_1 &> \frac{1}{4}t_2 \end{aligned}$$

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

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

example
  (h1 : 0 ≤ n) (h2 : n < (1/2)*K*x) (h3 : 0 < C) (h4 : 0 < eps)
  (h5 : eps < 1) (h6 : 1 + (1/3)*eps*(C+3)^(-1)*n < K*x) : false :=
by polya
```

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
-/
```

Interactive mode

```
example (h1 : x > 0) (h2 : x < y) (h3 : 0 < u)
  (h4 : u < v) (h5 : 0 < w + z) (h6 : w + z < r - 1)
  (h7 : u + (1+x)^2*(2*w + 2*z + 3) < 2*v + (1+y)^2 * (2*r + 1)) : false :=
begin [polya_tactic]
  add_hypotheses h1 h2 h3 h4 h5 h6 h7,
  trace_exprs,
  additive,
  multiplicative,
  trace_state,
  trace_contr,
  reconstruct
end
```

Conclusions

- 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.

Thanks for listening!