# Interactive theorem proving, automated reasoning, and dynamical systems

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

April 2016

# Formal methods

“Formal methods” = logic-based methods in CS, in:

- automated reasoning
- hardware and software verification
- artificial intelligence
- databases

# Formal methods

Based on logic and formal languages:

- syntax: terms, formulas, connectives, quantifiers, proofs
- semantics: truth, validity, satisfiability, reference

They can be used for:

- finding things (SAT solvers, constraint solvers, database query languages)
- proving things (automated theorem proving, model checking)
- verifying correctness (interactive theorem proving)

# Formal verification in industry

- Intel and AMD use ITP to verify processors.
- Microsoft uses formal tools such as Boogie and SLAM to verify programs and drivers.
- Xavier Leroy has verified the correctness of a C compiler.
- Airbus uses formal methods to verify avionics software.
- Toyota uses formal methods for hybrid systems to verify control systems.
- Formal methods were used to verify Paris' driverless line 14 of the Metro.
- The NSA uses (it seems) formal methods to verify cryptographic algorithms.

# Formal verfication in mathematics

There is no sharp line between industrial and mathematical verification:

- Designs and specifications are expressed in mathematical terms.
- Claims rely on background mathematical knowledge.

Mathematics is more interesting:

- Problems are conceptually deeper, less heterogeneous.
- More user interaction is needed.

# Formal methods in mathematics

“Conventional” computer-assisted proof:

- carrying out long, difficult, computations
- proof by exhaustion

Formal methods for discovery:

- finding mathematical objects
- finding proofs

Formal methods for verification:

- verifying ordinary mathematical proofs
- verifying computations.

# Formal methods in mathematics

Questions:

- How can computers help us reason about dynamical systems?
- How can computers help make results and computations more reliable?

# Outline

- Formal methods
- Interactive theorem proving
- Automated reasoning
- Verified computation

# Interactive theorem proving

Working with a proof assistant, users construct a formal axiomatic proof.

In most systems, this proof object can be extracted and verified independently.

# Interactive theorem proving

Some systems with large mathematical libraries:

- Mizar (set theory)
- HOL (simple type theory)
- Isabelle (simple type theory)
- HOL light (simple type theory)
- Coq (constructive dependent type theory)
- ACL2 (primitive recursive arithmetic)
- PVS (classical dependent type theory)

# Interactive theorem proving

Some theorems formalized to date:

- the prime number theorem
- the four-color theorem
- the Jordan curve theorem
- Gödel's first and second incompleteness theorems
- Dirichlet's theorem on primes in an arithmetic progression
- Cartan fixed-point theorems

There are good libraries for elementary number theory, real and complex analysis, point-set topology, measure-theoretic probability, abstract algebra, Galois theory, . . .

# Interactive theorem proving

Georges Gonthier and coworkers verified the Feit-Thompson Odd Order Theorem in Coq.

- The original 1963 journal publication ran 255 pages.
- The formal proof is constructive.
- The development includes libraries for finite group theory, linear algebra, and representation theory.

The project was completed on September 20, 2012, with roughly

- 150,000 lines of code,
- 4,000 definitions, and
- 13,000 lemmas and theorems.

# Interactive theorem proving

Hales announced the completion of the formal verification of the Kepler conjecture (*Flyspeck*) in August 2014.

- Most of the proof was verified in HOL light.
- The classification of tame graphs was verified in Isabelle.
- Verifying several hundred nonlinear inequalities required roughly 5000 processor hours on the Microsoft Azure cloud.

# Interactive theorem proving

Fabian Immler is working on verifying properties of dynamical systems in Isabelle.

- Proved existence and uniqueness of solutions to ODE's (Picard-Lindelöf and variations).
- With code extraction, can compute solutions. Following Tucker, has verified enclosures for the Lorenz attractor.

# Interactive theorem proving

Johannes Hölzl, Luke Serafin, and I have verified the central limit theorem in Isabelle.

The proof relied on Isabelle's libraries for analysis, topology, measure theory, measure-theoretic probability.

We proved:

- the portmanteau theorem (characterizations of weak convergence)
- Skorohod's theorem
- properties of characteristic functions and convolutions
- properties of the normal distribution
- the Levy uniqueness theorem
- the Levy continuity theorem

# Interactive theorem proving

```
theorem (in prob_space) central_limit_theorem:
  fixes
    X :: "nat ⇒ 'a ⇒ real" and
    μ :: "real measure" and
    σ :: real and
    S :: "nat ⇒ 'a ⇒ real"
  assumes
    X_indep: "indep_vars (λi. borel) X UNIV" and
    X_integrable: "⋀n. integrable M (X n)" and
    X_mean_0: "⋀n. expectation (X n) = 0" and
    σ_pos: "σ > 0" and
    X_square_integrable: "⋀n. integrable M (λx. (X n x)²)" and
    X_variance: "⋀n. variance (X n) = σ²" and
    X_distrib: "⋀n. distr M borel (X n) = μ"
  defines
    "S n ≡ λx. ∑i<n. X i x"
  shows
    "weak_conv_m (λn. distr M borel (λx. S n x / sqrt (n * σ²)))
        (density lborel std_normal_density)"
```

# The Lean Theorem Prover

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

It was "announced" in the summer of 2015.

It is open source, released under a permissive license, Apache 2.0.

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

# The Lean Theorem Prover

The aim is to bring interactive and automated reasoning together, and build

- an interactive theorem prover with powerful automation
- an automated reasoning tool that
    - produces (detailed) proofs,
    - has a rich language,
    - can be used interactively, and
    - is built on a verified mathematical library.

# The Lean Theorem Prover

Goals:

- Verify hardware, software, and hybrid systems.
- Verify mathematics.
- Combine powerful automation with user interaction.
- Support reasoning and exploration.
- Support formal methods in education.
- Create an eminently powerful, usable system.
- Bring formal methods to the masses.

# The Lean Theorem Prover

Notable features:

- based on a powerful dependent type theory
- written in C++, with multi-core support
- small, trusted kernel with an independent type checker
- standard and HoTT instantiations
- powerful elaborator
- can use proof terms or tactics
- Emacs mode with proof-checking on the fly
- browser version runs in javascript
- already has a respectable library
- automation is now the main focus

# The Lean Theorem Prover

```
structure semigroup [class] (A : Type) extends has_mul A :=
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))

structure monoid [class] (A : Type)
  extends semigroup A, has_one A :=
(one_mul : ∀ a, mul one a = a) (mul_one : ∀ a, mul a one = a)

definition pow {A : Type} [s : monoid A] (a : A) : ℕ → A
| 0     := 1
| (n+1) := pow n * a

theorem pow_add (a : A) (m : ℕ) : ∀ n, a^(m + n) = a^m * a^n
| 0        := by rewrite [nat.add_zero, pow_zero, mul_one]
| (succ n) := by rewrite [add_succ, *pow_succ, pow_add,
                          mul.assoc]

definition int.linear_ordered_comm_ring [instance] :
  linear_ordered_comm_ring int := ...
```

# The Lean Theorem Prover

```
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,
have 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
```

# The Lean Theorem Prover

```
theorem is_conn_susp [instance] (n : ℕ₋₂) (A : Type)
  [H : is_conn n A] : is_conn (n .+1) (susp A) :=
is_contr.mk (tr north)
begin
  apply trunc.rec, fapply susp.rec,
  { reflexivity },
  { exact (trunc.rec (λa, ap tr (merid a)) (center (trunc n A))) },
  { intro a, generalize (center (trunc n A)),
    apply trunc.rec, intro a', apply pathover_of_tr_eq,
    rewrite [transport_eq_Fr,idp_con],
    revert H, induction n with [n, IH],
    { intro H, apply is_prop.elim },
    { intros H,
      change ap (@tr n .+2 (susp A)) (merid a) = ap tr (merid a'),
      generalize a',
      apply is_conn_fun.elim n
            (is_conn_fun_from_unit n A a)
            (λx : A, trunctype.mk' n
              (ap (@tr n .+2 (susp A)) (merid a) = ap tr (merid x))),
      intros,
      change ap (@tr n .+2 (susp A)) (merid a) = ap tr (merid a),
      reflexivity } }
end
```

# Outline

- Formal methods
- Interactive theorem proving
- Automated reasoning
- Verified computation

# Automated reasoning

Ideal: given an assertion, $\varphi$, either

- provide a proof that $\varphi$ is true (or valid), or
- give a counterexample

Dually: given some constraints either

- provide a solution, or
- prove that there aren't any.

In the face of undecidability:

- search for proofs
- search for solutions

# Automated reasoning

Some fundamental distinctions:

- Domain-general methods vs. domain-specific methods
- Decision procedures vs. search procedures
- “Principled” methods vs. heuristics

# Automated reasoning

Domain-general methods:

- Propositional theorem proving
- First-order theorem proving
- Equational reasoning
- Higher-order theorem proving
- Nelson-Oppen “combination” methods

Domain-specific methods:

- Linear arithmetic (integer, real, or mixed)
- Nonlinear real arithmetic (real closed fields, transcendental functions)
- Algebraic methods (such as Gröbner bases)

# Automated reasoning

Formal methods in analysis:

- domain specific: reals, integers
- can emphasize either peformance or rigor
- can get further in restricted domains

Methods:

- quantifier elimination for real closed fields
- linear programming, semidefinite programming
- combination methods
- numerical methods
- heuristic symbolic methods

# Symbolic methods

Decision procedures for real closed fields represent a symbolic approach.

Problems:

- Complexity overwhelms.
- Polynomials may not be expressive enough.
- Undecidability sets in quickly.

How can we integrate numeric approaches?

- Calculations are only approximate.
- We want an exact guarantee.

# Numeric methods

Sicun Gao, Ed Clarke, and I proposed a framework that offers:

- More flexibility: arbitrary computable functions
- A restriction: quantification only over bounded domains
- A compromise: approximate decidability (but with an exact guarantee)

This provides a general framework for thinking about verification problems.

# Numeric methods

Choose a language with 0, $+$, $-$, $<$, $\leq$, $|\cdot|$, and symbols for *any* computable functions you want.

Fix a "tolerance" $\delta > 0$. We defined:

- $\varphi^{+\delta}$, a slight strengthening of $\varphi$
- $\varphi^{-\delta}$, a slight weakening of $\varphi$

such that whenever $\delta' \geq \delta \geq 0$, we have

$$\varphi^{+\delta'} \to \varphi^{+\delta} \to \varphi \to \varphi^{-\delta} \to \varphi^{-\delta'}.$$

# Numeric methods

Say a formula $\varphi$ is *bounded* if every quantifier is of the form $\forall x \in [s, t]$ or $\exists x \in [s, t]$ .

**Theorem.** There is an algorithm which, given any bounded formula $\varphi$, correctly returns on of the following two answers:

- $\varphi$ is true
- $\varphi^{+\delta}$ is false.

For verification problems, think of the first answer as "the system is safe," and the second as "a small perturbation of the system is unsafe."

Note that there is a grey area where either answer is allowed.

# Numeric methods

This is a theoretical result. The practical goal is to implement such an algorithm.

Gao, Clarke, Soonho Kong, and others are developing a tool, *dReal*:

- It focuses on the existential / universal fragment.
- It uses an SMT ("satisfiability modulo theories") framework.
- It uses IBEX for interval constraint propagation.
- It uses the CAPD library to compute numerical enclosures for ODE's.

See `https://dreal.github.io`

# A heuristic symbolic method

Consider the following implication:

$$0 < x < y, \; u < v$$

$$\Longrightarrow$$

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

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

# A heuristic symbolic method

Robert Lewis and I (initially with Cody Roux) have developed a new approach, that:

- verifies inequalities on which other procedures fail
- extends beyond the language of RCF
- is amenable to producing proof terms
- captures natural patterns of inference

But:

- It is not complete.
- It not guaranteed to terminate.

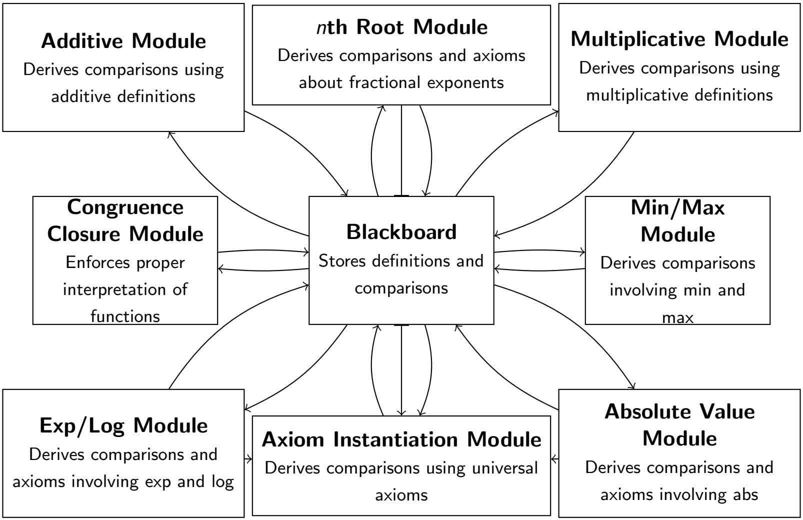
It is designed to complement other procedures.

# A heuristic symbolic method

The main ideas:

- Use forward reasoning (guided by the structure of the problem).
- Show "hypotheses $\Rightarrow$ conclusion" by negating the conclusion and deriving a contradiction.
- As much as possible, put terms in canonical "normal forms," e.g. to recognize that $3(x + y)$ is a multiple of $2y + 2x$.
- Derive relationships between "terms of interest," including subterms of the original problem.
- Different modules contribute bits of information, based on their expertise.

# Computational structure



# A heuristic symbolic method

We have a prototype Python implementation, *Polya*.

The code is open-source and available online.

- An associated paper.
- Rob's MS thesis.
- Slides from Rob's talks (from which I have borrowed).

We are planning to implement this in Lean.

# Outline

- Formal methods
- Interactive theorem proving
- Automated reasoning
- Verified computation

# Verified computation

Important computational proofs have been verified:

- The four color theorem (Gonthier, 2004)
- The Kepler conjecture (Hales et al., 2014)
- Various aspects of Kenzo (computation in algebraic topology) have been verified:
  - in ACL2 (simplicial sets, simplicial polynomials)
  - in Isabelle (the basic perturbation lemma)
  - in Coq/SSReflect (effective homology of bicomplexes, discrete vector fields)

# Verified computation

Some approaches:

- rewrite the computations to construct proofs as well (Flyspeck: nonlinear bounds)
- verify certificates (e.g. proof sketches, duality in linear programming)
- verify the algorithm, and then execute it with a specialized (trusted) evaluator (Four color theorem)
- verify the algorithm, extract code, and run it (trust a compiler or interpreter) (Flyspeck: enumeration of tame graphs)

# Verified computation

In dynamical systems:

- Immler and Hölzl have obtained verified solutions to ODE's.
- Immler has obtained verified algorithms for geometric zonotopes (for convex hull calculutions).
- He has verified and enclosure for the Lorenz attractor.

The formalizations are carried out in Isabelle, and Standard ML code is extracted.

# Conclusions

- Computers change the kinds of proofs that we can discover and verify.
- In the long run, formal methods *will* play an important role in mathematics.
- It will take clever ideas, and hard work, to understand how to use them effectively.