Formal verification, interactive theorem proving, and automated reasoning

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

January 2014

Formal verification

Formal methods can be used to verify correctness:

- verifying that a circuit description, an algorithm, or a network or security protocol meets its specification; or
- verifying that a mathematical statement is true.

Two approaches:

- Model checking: reduce to a finite state space, and test exhaustively.
- Interactive theorem proving: construct a formal axiomatic proof of correctness.

Formal verification in industry

Formal methods are becoming common:

- 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 verification 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.

I will focus, however, on verifying mathematics.

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

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)

Tactic-style proof scripts

```
lemma prime_factor_nat: "n ~= (1::nat) ==>
    EX p. prime p & p dvd n"
  apply (induct n rule: nat_less_induct)
  apply (case_tac "n = 0")
  using two_is_prime_nat apply blast
  apply (case_tac "prime n")
  apply blast
  apply (subgoal_tac "n > 1")
  apply (frule (1) not_prime_eq_prod_nat)
  apply (auto intro: dvd_mult dvd_mult2)
done
```

Declarative proof scripts

```
proof (induct n rule: less_induct_nat)
  fix n :: nat
  assume "n ~= 1" and
    ih: "ALL m < n. m ~= 1 --> (EX p. prime p & p dvd m)"
  then show "EX p. prime p & p dvd n"
  proof -
  { assume "n = 0"
    moreover note two_is_prime_nat
    ultimately have ?thesis by auto }
  moreover
  { assume "prime n" then have ?thesis by auto }
  moreover
  { assume "n ~= 0" and "~prime n"
    with ‘n ~= 1‘ have "n > 1" by auto
    with ‘~prime n‘ and not_prime_eq_prod_nat obtain m k where
      "n = m * k" and "1 < m" and "m < n" by blast
    with ih obtain p where "prime p" and "p dvd m" by blast
    with ‘n = m * k‘ have ?thesis by auto }
  ultimately show ?thesis by blast
```

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

Thomas Hales' formal verification of the Kepler conjecture (*Flyspeck*) in HOL light (and Isabelle) is nearing completion.

- Three essential uses of computation
  - enumerating tame hypermaps
  - proving nonlinear inequalities
  - showing infeasibility of linear programs
- The formalization led to even stronger results.

Interactive theorem proving

Vladimir Voevodsky has launched a project to develop "univalent foundations."

- Constructive dependent type theory has natural homotopy-theoretic interpretations (Voevodsky, Awodey and Warren).
- Rules for equality characterize equivalence "up to homotopy."
- One can consistently add an axiom to the effect that "isomorphic structures are identical."

This makes it possible to reason "homotopically" in systems like Coq and Agda.

Interactive theorem proving

Rigor and correctness are important to mathematics. Formal verification is a technology that can help.

Compare to TeX, computer algebra systems, numeric simulation, MathSciNet, Google.

Interactive theorem proving is not "ready for prime time."

- There is a steep learning curve.
- Verification can be time consuming and painful.

Interactive theorem proving

Short term wins:

- verifying delicate, technical calculations by hand
- verifying computation

Long term: we need better

- libraries (and means to translate between them)
- automation (decision procedures, search procedures)
- ways to incorporate and verify computations
- ways to express mathematical knowledge and expertise

Interactive theorem proving

Formal verification raises mathematical, logical, computational, and conceptual questions that are interesting in their own right.

Themes:

- Current successes are based on 20th century insights.
  - logical languages and axiomatic frameworks
  - syntactic notions, deductive systems, normal forms
  - semantic notions, completeness
  - decidability and decision procedures
  - computability and semantics of computation
- Current limitations need better theoretical understanding.

Outline

- Introduction
  - formal verification in general
  - interactive theorem proving
  - the state of the art
- Practical foundations
  - logical frameworks
  - assertion languages
  - proof languages
- Automation
  - decision procedures
  - search procedures
  - combination procedures
- Mathematical knowledge management

Logical frameworks

Mizar is based on a Tarski-Grothendieck set theory (with universes).

HOL, Isabelle, and HOL light are based on simple type theory.

Coq is based on constructive dependent type theory.

Modulo some axioms and creativity, these are interpretable in one another.

Logical frameworks

```
theorem PrimeNumberTheorem:
  "(%n. pi n * ln (real n) / (real n)) ----> 1"

!C. simple_closed_curve top2 C ==>
  (?A B. top2 A /\ top2 B /\
    connected top2 A /\ connected top2 B /\
  ~(A = EMPTY) /\ ~(B = EMPTY) /\
   (A INTER B = EMPTY) /\ (A INTER C = EMPTY) /\
       (B INTER C = EMPTY) /\
      (A UNION B UNION C = euclid 2)

!d k. 1 <= d /\ coprime(k,d)
  ==> INFINITE { p | prime p /\ (p == k) (mod d) }
```

Logical frameworks

```
Theorem Sylow's_theorem :
  [/\ forall P,
      [max P | p.-subgroup(G) P] = p.-Sylow(G) P,
      [transitive G, on 'Syl_p(G) | 'JG],
      forall P, p.-Sylow(G) P ->
        #|'Syl_p(G)| = #|G : 'N_G(P)|
   & prime p -> #|'Syl_p(G)| %% p = 1%N].

Theorem Feit_Thompson (gT : finGroupType)
   (G : {group gT}) :
  odd #|G| →solvable G.

Theorem simple_odd_group_prime (gT : finGroupType)
    (G : {group gT}) :
  odd #|G| →simple G →prime #|G|.
```

Logical frameworks

```
Local Inductive S1 : Type :=
| base : S1.

Axiom loop : base ~~> base.

Definition S1_rect (P : S1 -> Type) (b : P base) (l
    : loop # b ~~> b)
  : forall (x:S1), P x
  := fun x => match x with base => b end.

Axiom S1_rect_beta_loop
  : forall (P : S1 -> Type) (b : P base) (l : loop #
      b ~~> b),
  apD (S1_rect P b l) loop ~~> l.

Theorem int_equiv_loopcirc :
  equiv int (base ~~> base).
```

Logical frameworks

We do not read, write, and understand mathematics at the level of a formal axiomatic system.

Challenge: Develop ways of verifying mathematics at an appropriate level of abstraction.

Assertion language

Consider the following mathematical statements:

- For every $x \in \mathbb{R}$, $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.
- If G and H are groups and f is a homomorphism from G to H, then for every $a, b \in G$, $f(ab) = f(a)f(b)$.
- If F is a field of characteristic p and $a, b \in F$, then $(a+b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p$.

How do we parse these?

Assertion language

Observations:

1. The index of the summation is over the natural numbers.
2. $\mathbb{N}$ is embedded in $\mathbb{R}$.
3. In "$a \in G$," G really means the underlying set.
4. ab means multiplication in the relevant group.
5. p is a natural number (in fact, a prime).
6. The summation operator make sense for any monoid (written additively).
7. The summation enjoys extra properties if the monoid is commutative.
8. The additive part of any field is so.
9. $\mathbb{N}$ is also embedded in any field.
10. Alternatively, any abelian group is a $\mathbb{Z}$-module, etc.

Assertion language

The hierarchy of algebraic structures involves:

- Subclasses: every abelian group is a group
- Reducts: the additive part of a ring is an abelian group
- Instances: the integers are an abelian group
- Embedding: the integers are embedded in the reals
- Uniform constructions: the automorphisms of a field form a group

Advantages:

- Reusing notation: 0, $+$, $a \cdot b$
- Reusing definitions: $\sum_{i \in I} a_i$
- Reusing facts: identities involving sums

Assertion language

A lot of implicit knowledge goes into reading and understanding ordinary mathematical statements.

Challenge: Develop formal models of *everyday* mathematical language.

Proof language

```
Lemma pullback_universal {A B C : Type} (f : A -> C)
  (g : B -> C)
: is_pullback_cone (pullback_cospan_cone f g).
Proof.
  intros X.
  apply (isequiv_adjointify
      (cospan_cone_to_map_to_pullback)).
  (* is_section *)
  intros [y1 [y2 y3]].
  unfold map_to_cospan_cone, cospan_cone_to_map_to_pullback.
  unfold cospan_cone_map2, cospan_cone_comm; simpl.
  unfold pullback_comm, compose; simpl. exact 1.
  (* is_retraction *)
  intros m. apply path_forall.
  intros x; simpl.
  apply pullback_path'.
  exists 1, 1. simpl.
  exact (concat_p1 _ @ concat_1p _).
Defined.
```

Proof language

```
lemma cdf_to_real_distribution:
  fixes F :: "real ⇒ real"
  assumes nondecF : "⋀x y. x ≤ y ⟹ F x ≤ F y" and
    right_cont_F : "⋀a. continuous (at_right a) F" and
    lim_F_at_bot : "(F ---> 0) at_bot" and
    lim_F_at_top : "(F ---> 1) at_top"
  shows "∃M. real_distribution M ∧ cdf M = F"
proof -
  have "∃μ :: real set ⇒ ereal.
      (∀(a::real) b. a < b ⟶ μ {a<..b} = F b - F a)
      ∧ measure_space UNIV (sets borel) μ"
    apply (rule cdf_to_measure)
    using assms by auto
  then obtain μ :: "real set ⇒ ereal" where
    1: "∀(a::real) b. a < b ⟶ μ {a<..b} = F b - F a" and
    2: "measure_space UNIV (sets borel) μ" by auto
  let ?M = "measure_of UNIV (sets borel) μ"
      ...
```

Proof language

```
Theorem Burnside_normal_complement :
  'N_G(S) \subset 'C(S) -> 'O_p^'(G) ><| S = G.
Proof.
move=> cSN; set K := 'O_p^'(G); have [sSG pS _] := and3P sylS.
have [p'K]: p^'.-group K /\ K <| G by rewrite pcore_pgroup pcore_normal.
case/andP=> sKG nKG; have{nKG} nKS := subset_trans sSG nKG.
have{pS p'K} tiKS: K :&: S = 1 by rewrite setIC coprime_TIg
    ?(pnat_coprime pS).
suffices{tiKS nKS} hallK: p^'.-Hall(G) K.
  rewrite sdprodE //= -/K; apply/eqP; rewrite eqEcard ?mul_subG //=.
  by rewrite TI_cardMg //= (card_Hall sylS) (card_Hall hallK) mulnC
      partnC.
pose G' := G^`(1); have nsG'G : G' <| G by rewrite der_normal.
suffices{K sKG} p'G': p^'.-group G'.
  have nsG'K: G' <| K by rewrite (normalS _ sKG) ?pcore_max.
  rewrite -(pquotient_pHall p'G') -?pquotient_pcore //= -/G'.
  by rewrite nilpotent_pcore_Hall ?abelian_nil ?der_abelian.
suffices{nsG'G} tiSG': S :&: G' = 1.
  have sylG'S : p.-Sylow(G') (G' :&: S) by rewrite (pSylow_normalI _
      sylS).
  rewrite /pgroup -[#|_|](partnC p) ?cardG_gt0 // -{sylG'S}(card_Hall
      sylG'S).
  by rewrite /= setIC tiSG' cards1 mul1n pnat_part.
apply/trivgP; rewrite /= focal_subgroup_gen ?(p_Sylow sylS) // gen_subG.
```

Proof language

```
encode-decode  : {x : S¹} → (c : Cover x)
               → Path (encode (decode{x} c)) c
encode-decode {x} = S¹-induction
   (\ (x : S¹) →  (c : Cover x)
                  → Path (encode{x} (decode{x} c)) c)
   encode-loop^ hedberg x where
     postulate hedberg : _

decode-encode  : {x : S¹} (alpha : Path base x)
               → Path (decode (encode alpha)) alpha
decode-encode {x} alpha =
 path-induction
  (\ (x' : S¹) (alpha' : Path base x')
      → Path (decode (encode alpha')) alpha')
  id alpha

all-loops : (alpha : Path base base) → Path alpha (loop^ (encode alpha))
all-loops alpha = ! (decode-encode alpha)

Omega₁[S¹]-is-Int : HEquiv (Path base base) Int
Omega₁[S¹]-is-Int =
   hequiv encode decode decode-encode encode-loop^
```

Proof language

Proofs work in subtle ways: setting out structure, introducing local hypotheses and subgoals, unpacking definitions, invoking background facts, carring out calculations, and so on.

Challenge: Develop formal models of *everyday* mathematical proof.

Automated reasoning

Most systems employ automated techniques to fill in small gaps in reasoning.

One can distinguish between:

- Domain-general methods and domain-specific methods
- Search procedures and decision procedures
- "Principled" methods and heuristics

Automated reasoning

Domain-general methods:

- Propositional theorem proving
- First-order theorem proving
- Higher-order theorem proving
- Equality reasoning
- 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 methods do especially well on large, homogeneous problems, but often fail otherwise.

Outline

- Introduction
  - formal verification in general
  - interactive theorem proving
  - the state of the art
- Practical foundations
  - logical frameworks
  - assertion languages
  - proof languages
- Automation
  - decision procedures
  - search procedures
  - combination procedures
- Mathematical knowledge management

Decision procedures

Full first-order theory:

- quantifier elimination
- "global methods" (Cooper, CAD)
- reductions to Rabin's *S2S*
- Feferman-Vaught

Sometimes it is better to focus on the universal fragment:

- Some theories are only decidable at this level.
- It can be more efficient.
- One can use certificates.
- A lot of mathematical reasoning is close to quantifier-free.
- Procedures can be *combined*.

Decision procedures

Some theories that are decidable:

- Linear real arithmetic, $Th(\langle \mathbb{R}, 0, 1, +, < \rangle)$
- Presburger arithmetic
- Mixed integer-linear arithmetic
- Real closed fields
- Algebraically closed fields
- Real vector spaces, inner product spaces (Solovay, Arthan, Harrison)
- Boolean Algebra with Presburger Arithmetic (and cardinality) (Kuncak)

All (except the last) have been implemented in theorem provers.

Decision procedures

Some theories with decidable universal fragment:

- integer / linear arithmetic reduces to integer / linear programming
- real closed fields: semidefinite optimization (Parillo, Harrison), lazy CAD (Jovanović, de Moura)
- algebraically closed fields: Groebner bases
- geometry: Wu's method, the area method
- uninterpreted functions (congruence closure)
- theories of arrays, lists

Decision procedures

The bad news:

- Decision procedures are often infeasible.
- Undecidability sets in quickly.
- Ordinary mathematical reasoning is heterogeneous.

Decidability results do not bear directly on practical problems:

- A decision procedure that runs too slowly on the examples you care about is worthless.
- An unprincipled search that gets all your inferences is fine.
- undecidability results often rely on contrived coding, irrelevant to practice.
- The set of inferences that can be verified in ZFC with less than 10^{100} symbols is decidable (in constant time!).

Decision procedures

What we really care about is having proof procedures that do well on the kinds of problems that come up "in practice."

Challenge: Understand what that means.

Search procedures

There is a large industry for general first-order theorem provers:

- resolution, paramodulation
- tableaux methods
- model-based methods

Some do well on wide and shallow, others on narrow and deep.

See:

- the TPTP library (Thousands of Problems for Theorem Provers)
- the annual CADE competition (Computers in Automated Deduction)

Larry Paulson's *Sledgehammer* tool for Isabelle is quite handy.

Search procedures

But current search procedures are limited:

- They do not do well with domain specific reasoning (like arithmetic).
- They do not capture other aspects of our expertise.
- They often fail on very straightforward inferences.

Challenge: Understand how ordinary mathematical knowledge and expertise can guide a search.

Combining decision procedures

Theorem (Nelson-Oppen)

Suppose T_1 and T_2 are "stably infinite," and their universal fragments are decidable. Suppose that the languages are disjoint, except for the equality symbol. Then the universal fragment of $T_1 \cup T_2$ is decidable.

In particular, if T_1 and T_2 have only infinite models, the conclusion holds.

This allows you to design decision procedures for individual theories and then put them together.

Combining decision procedures

SMT ("Satisfiability modulo theories") is based on this idea:

- A fast propositional SAT solver tries to build a satisfying assignment.
- Individual decision procedures examine proposals, and report conflicts.
- The SAT solver incorporates this information into the search.
- Some systems go beyond the universal fragment, for example, instantiating universal axioms in sensible ways.

See SMT-LIB and SMT-COMP.

Combining decision procedures

We are still trying to understand how:

- to handle quantifiers in such a framework
- to combine theories with non-disjoint signatures
- to combine general search procedures with domain specific methods and computation

Challenge: Understand how to effectively use and combine domain-specific expertise in general settings.

Automated reasoning

Contemporary successes build on core results in logic:

- formal axiomatic systems, semantics, and completness proofs
- cut elimination and cut-free provability
- normalization and the lambda calculus
- Skolemization and properties
- decision procedures
- model theory of algebraic structures
- Craig's interpolation lemma

Computer scientists have added many important insights.

But the theory guides the enterprise.

Mathematical Knowledge Management

Suppose one proves a theorem in one theorem prover. Can one import it into another?

Problems:

- different foundational frameworks
- different axiomatic bases
- different definitions (e.g. reals as Cauchy sequences vs. Dedekind cuts)
- different side conditions (e.g. $x/0$)
- proof terms get really big

Mathematical Knowledge Management

Other questions:

- How can we search for mathematical theorems in a large library?
- How can we assess relevance?
- How can questions and partial results be communicated between different reasoning tools?

Challenge: Understand how to represent ordinary mathematical knowledge and expertise in a robust way.

Summary

Formal methods are valuable to mathematics:

- They provide additional precision.
- They provide strong guarantees of correctness.
- They make it possible to verify mathematical computation.
- They make it possible to use automation in a reliable way.
- They assist in the storing, finding, and communicating mathematical knowledge.

Summary

Final message:

- The goal of mathematics is to extend mathematical knowledge.
- 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.
- Successes to date rely on a 20th century logical understanding of mathematical method.
- There are still fundamental conceptual problems that need to be addressed.
- Mathematical logic still has a role to play.