Interactive Theorem Proving

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

May 2015

Sequence of lectures

1. Formal methods in mathematics
2. Automated theorem proving
3. Interactive theorem proving
4. Formal methods in analysis

Recap

Formal methods in mathematics:

- "Formal methods" = use of logic-based methods in CS
- Two roles for formal methods:
    - Discovery: searching for proofs, searching for objects
    - Verification: checking correctness
- They are widely used in CS.
- There are a few successes for formal methods in mathematical discovery, but there should be more.
- There are some more impressive successes in verification, though it is not ready for prime time.

Recap

Automated reasoning:

- Ideal: given φ, find a proof, or a counterexample
- One can distinguish between “domain general” and “domain specific”
- Domain general:
    - propositional logic (fast SAT, CDCL)
    - first-order theorem proving (resolution theorem provers, etc.)
    - equational reasoning
    - higher-order reasoning
    - “combination” methods
- Undecidability, combinatorial explosion set in.

Interactive theorem proving

Definitions and assertions are made in a formal axiomatic system.

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

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

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
```

Proof terms

```
theorem quotient_remainder {x y : ℕ} :
  x = x div y * y + x mod y :=
by_cases_zero_pos y
  (show x = x div 0 * 0 + x mod 0, from
    (calc
      x div 0 * 0 + x mod 0 = 0 + x mod 0 : mul_zero
        ... = x mod 0                     : zero_add
        ... = x                           : mod_zero)⁻¹)
  (take y,
    assume H : y > 0,
    show x = x div y * y + x mod y, from
      nat.case_strong_induction_on x
        (show 0 = (0 div y) * y + 0 mod y, by simp)
        (take x,
          assume IH : ∀x', x' ≤ x →
            x' = x' div y * y + x' mod y,
          show succ x = succ x div y * y + succ x mod y,
                ...
```

Interactive theorem proving

Compare to automated theorem proving or computer algebra:

- You can express anything.
- Every object has a precise specification.
- Everything is proved, down to axioms.

But it is a lot more work.

The ideal: combine

- axiomatic proof
- automation
- numeric and symbolic computation
- exploration

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)

I am contributing to the development of a new theorem prover, Lean.

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
- The Kepler conjecture
- The Feit-Thompson theorem

Contents

- formal frameworks
- assertion languages
- proof languages
- automated support
- verifying computation

Formal frameworks

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

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

Coq, Agda, and Lean are based on (constructive) dependent type theory.

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

Set theory

Consider the statement “there are no infinite descending sequences of natural numbers.”

In set theory, one might render this as follows:

$$\forall f\ (f \in (\mathbb{N} \to \mathbb{N}) \to \exists i\ (i \in \mathbb{N} \wedge (f(i+1) \geq f(i)))$$

You need a different $+$ and $\geq$ for each domain.

The system doesn’t know which objects are functions.

Simple type theory

In simple type theory, every object has a *type*.

There are:

- base types like `nat`, `int`, and `bool`.
- Type constructors, like $A \to B$, $A \times B$, $list A$.

This gives type inference, overloading, pattern matching.

$$\forall f : \mathbb{N} \to \mathbb{N} \;\; \exists i \;\; f(i+1) \geq f(i).$$

Dependent type theory

Some types depend on types: $A \times B$, $\mathit{list}\ (A \times B)$, ...

Some types depend on parameters: $\mathit{vec}\ A\ n$, $\mathit{Zmod}\ n$.

Dependent type theory allows types to depend on parameters.

The same language can be used to encode

- types
- mathematical objects
- propositions
- proofs.

Type inference

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 $a, b \in G$, $f(a \cdot b) = f(a) \cdot f(b)$."

"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?

Type inference

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 is a $\mathbb{Z}$-module, etc.

Type inference

Spelling out these details formally can be painful.

Typically, the relevant information can be *inferred* by keeping track of the *type* of objects we are dealing with:

- In “$a \in G$,” the “$\in$” symbol expects a set on the right.
- In “ab,” multiplication takes place in “the” group that a is assumed to be an element of.
- In “$x^i / i!$,” one expects the arguments to be elements of the same structure.

Type inference: not only inferring types, but also inferring information from type considerations.

Type inference

Structure hierarchies:

- 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

Type inference

```
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]

structure group [class] (A : Type) extends monoid A, has_inv A :=
(mul_left_inv : ∀a, mul (inv a) a = one)
```

Assertion languages

```
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) }
```

Assertion languages

```
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|.
```

Assertion languages

```
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).
```

Assertion languages

```
variables {C D : Category} {F G H : C ⇒ D}
definition nt_compose
    (η : G ⟹ H) (θ : F ⟹ G) : F ⟹ H :=
natural_transformation.mk
  (take a, η a ∘ θ a)
  (take a b f, calc
    H f ∘ (η a ∘ θ a) = (H f ∘ η a) ∘ θ a : assoc
                  ... = (η b ∘ G f) ∘ θ a : naturality
                  ... = η b ∘ (G f ∘ θ a) : assoc
                  ... = η b ∘ (θ b ∘ F f) : naturality
                  ... = (η b ∘ θ b) ∘ F f : assoc)
```

Assertion languages

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.

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
```

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 : S^1} → (c : Cover x)
               → Path (encode (decode{x} c)) c
encode-decode {x} = S^1-induction
   (\ (x : S^1) →  (c : Cover x)
                  → Path (encode{x} (decode{x} c)) c)
   encode-loop^ hedberg x where
     postulate hedberg : _

decode-encode  : {x : S^1} (alpha : Path base x)
               → Path (decode (encode alpha)) alpha
decode-encode {x} alpha =
 path-induction
  (\ (x' : S^1) (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)

Omega_1[S^1]-is-Int : HEquiv (Path base base) Int
Omega_1[S^1]-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.

Verifying 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)

Verifying 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)

Verifying computation

Challenges:

- Understand how to reason about mathematical algorithms.
- Understand how to ensure their correctness.
- Understand how to incorporate computation into proof.

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)

Question: how to incorporate them into interactive theorem provers?

Automated reasoning

Isabelle incorporates a good deal of automation:

- a good simplifier / term rewriter
- a tableau theorem prover ("auto")
- a resolution theorem prover ("metis")
- decision procedures for linear arithmetic and Presburger arithmetic
- an interface to the Z3 SMT solver, with proof reconstruction.

HOL-light also has very good automation, including:

- A decision procedure for the universal fragment of real-closed fields, based on semidefinite programming and sum-of-squares.
- Decision procedures for vector spaces.
- Algebraic procedures based on Gröbner bases.

Automated reasoning

Challenges:

- Understand how to develop verified automated procedures.
- Understand how to import / reconstruct proof sketches.
- Understand how to customize / control automation.
- Understand how to combine automated procedures.

Conclusions

Verifying correctness is not the most exciting part of mathematics, but correctness is important.

Moreover, having precise specifications is valuable:

- It makes it possible to use and trust computation.
- It makes it possible to use and trust automation.

We are at the dawn of a new era, but the technology still has a long way to go.