Formalization and Interactive Theorem Proving

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

June 2016

Sequence of lectures

1. Mathematical Understanding
2. The History of Dirichlet's Theorem
3. Formalization and Interactive Theorem Proving
4. The Role of the Diagram in Euclid's *Elements*
5. Modularity in Mathematics

Outline

- Overview
- Assertion languages
- Proof languages
- Structures and the algebraic hierarchy
- Finding the right concepts
- Verifying computation
- Automation

Formal verification

Formal methods can be used to verify correctness:

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

Formal verification in industry

Formal methods are commonly used:

- Intel and AMD use ITP to verify processors.
- Microsoft uses formal tools such as Boogie and SLAM to verify programs and drivers.
- CompCert 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.

Mathematics has a different character, however:

- 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)

I am contributing to the development of a new theorem prover, called *Lean*:

`http://leanprover.github.io`

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 central limit theorem

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

Homotopy type theory is a new field of research.

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

This makes it possible to reason "homotopically" in interactive theorem provers that support dependent type theory.

Outline

- Overview
- Assertion languages
- Proof languages
- Structures and the algebraic hierarchy
- Finding the right concepts
- Verifying computation
- Automation

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

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) = σ^2" **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 * σ^2)))
        (density lborel standard_normal_density)"

Tracking mathematical objects

In ordinary mathematics, an expression may denote:

- a natural number: 3, $n^2 + 1$
- an integer: -5, $2j$
- an ordered triple of natural numbers: $(1, 2, 3)$
- a function from natural numbers to reals: $(s_n)_{n \in \mathbb{N}}$
- a set of reals: $[0, 1]$
- a function which takes a measurable function from the reals to the reals and a set of reals and returns a real: $\int_A f \; d\lambda$
- an additive group: $\mathbb{Z}/m\mathbb{Z}$
- a ring: $\mathbb{Z}/m\mathbb{Z}$
- a module over some ring: $\mathbb{Z}/m\mathbb{Z}$ as a $\mathbb{Z}$-module
- an element of a group: $g \in G$
- a function which takes an element of a group and a natural number and returns another element of the group: g^n
- a homomorphism between groups: $f : G \to G$
- a function which takes a sequence of groups and returns a group: $\prod_i G_i$
- a function which takes a sequence of groups indexed by some diagram and homomorphisms between them and returns a group: $\lim_{i \in D} G_i$

Tracking mathematical objects

In set theory, these are all sets.

We have to rely on hypotheses and theorems to establish that these objects fall into the indicated classes.

For many purposes, it is useful to have a discipline which assigns to each syntactic object a *type*.

This is what type theory is designed to do.

Simple type theory

In simple type theory, we start with some basic types, and build compound types.

```
check ℕ    -- Type₁
check bool
check ℕ → bool
check ℕ × bool
check ℕ → ℕ
check ℕ × ℕ → ℕ
check ℕ → ℕ → ℕ
check ℕ → (ℕ → ℕ)
check ℕ → ℕ → bool
check (ℕ → ℕ) → ℕ
```

Simple type theory

We then have terms of the various types:

```
variables (m n: ℕ) (f : ℕ → ℕ) (p : ℕ × ℕ)
variable  g : ℕ → ℕ → ℕ
variable  F : (ℕ → ℕ) → ℕ

check f
check f n
check g m n
check g m
check (m, n)
check pr₁ p
check m + n^2 + 7
check F f
check F (g m)
check f (pr₂ (pr₁ (p, (g m, n))))
```

Simple type theory

Modern variants include variables ranging over types and type constructors.

```
variables A B : Type

check list A
check set A
check A × B
check A × ℕ

variables (l : list A) (a b c : A) (s : set A)

check a :: l
check [a, b] ++ c :: l
check length l
check '{a} ∪ s
```

Dependent type theory

In dependent type theory, type constructors can take terms as arguments:

```
variables (A : Type) (m n : ℕ)

check tuple A n
check matrix ℝ m n
check Zmod n

variables (s : tuple A m) (t : tuple A n)

check s ++ t     -- tuple A (m + n)
```

The trick: types themselves are now terms in the language.

Dependent type theory

For example, type constructors are now type-valued functions:

```
variables A B : Type

constant prod : Type → Type → Type
constant list : Type → Type

check prod A B
check list A
check prod A ℕ
check list (prod A ℕ)
```

Note: foundationally, we want to *define* `prod` and `list`, not declare then.

Assertion languages

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

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

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 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.
```

Propositions and proofs in dependent type theory

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.

Encoding propositions

```
variables p q r : Prop

check p ∧ q
check p ∧ (p → q ∨ ¬ r)

variable  A : Type
variable  S : A → Prop
variable  R : A → A → Prop

local infix ` ≺ `:50 := R

check ∀ x, S x
check ∀ f : ℕ → A, ∃ n : ℕ, ¬ f (n + 1) ≺ f n
check ∀ f, ∃ n : ℕ, ¬ f (n + 1) ≺ f n
```

Encoding proofs

Given `P : Prop`, view `t : P` as saying "t is a proof of P."

```
theorem and_swap : p ∧ q → q ∧ p :=
assume H  : p ∧ q,
have H1 : p, from and.left H,
have H2 : q, from and.right H,
show q ∧ p, from and.intro H2 H1

theorem and_swap' : p ∧ q → q ∧ p :=
λ H, and.intro (and.right H) (and.left H)

check and_swap -- ∀ (p q : Prop), p ∧ q → q ∧ p
```

Encoding proofs

```
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,
                ...
```

Encoding proofs

```
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,
assert 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
```

Proof language

We can introduce automation, and syntactic sugar for various proof constructs.

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.

Outline

- Overview
- Assertion languages
- Proof languages
- Structures and the algebraic hierarchy
- Finding the right concepts
- Verifying computation
- Automation

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.

Algebraic structures

Relationships between structures:

- subclasses: every abelian group is a group
- reducts: the additive part of a ring is an abelian group
- instances: the integers are an ordered ring
- embedding: the integers are embedded in the reals
- uniform constructions: the automorphisms of a field form a group

Goals:

- reuse notation: 0, $a + b$, $a \cdot b$
- reuse definitions: $\sum_{i \in I} a_i$
- reuse facts: e.g. $\sum_{i \in I} c \cdot a_i = c \cdot \sum_{i \in I} a_i$

Algebraic structures

```
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 := ...
```

Finding the right concepts

There are too many variants of limits:

- $\lim_{n\to\infty} a_n = r$
- $\lim_{x\to a} f(x) = b$
- $\lim_{x\to a} f(x) = -\infty$
- $\lim_{x\to -\infty} f(x) = b$
- $\lim_{x\to a^-} f(x) = b$

Imagine proving $\lim_{x\to a}(fx + gx) = \lim_{x\to a} fx + \lim_{x\to a} gx$ for each variant.

Finding the right concepts

The solution: make the source and target arbitrary *filters*.

This was done by Hölzl, Immler, and Huffman in Isabelle.

(I will describe an implementation of their approach in Lean.)

Finding the right concepts

A *filter* is a collection of sets closed upwards, and under finite intersections.

```
structure filter (A : Type) :=
(sets            : set (set A))
(univ_mem_sets   : univ ∈ sets)
(inter_closed    : ∀ {a b}, a ∈ sets → b ∈ sets →
                     a ∩ b ∈ sets)
(is_mono         : ∀ {a b}, a ⊆ b → a ∈ sets → b ∈ sets)

definition eventually (P : A → Prop) (F : filter A) :
  Prop := P ∈ F

definition complete_lattice :
  complete_lattice (filter A) := ...
```

Finding the right concepts

Given $f : X \to Y$, a filter F_2 on Y, and a filter F_1 on X, define

"f tends to F_2, as the input tends to F_1."

```
definition mapfilter (f : X → Y) (F : filter X) : filter Y :=
{| filter,
  sets          := λ s, (f '- s) ∈ F,
  ... |}

definition tendsto (f : X → Y) (F₂ : filter Y) (F₁ : filter X)
    : Prop := mapfilter f F₁ ≤ F₂

theorem tendsto_iff (F₂ : filter Y) (F₁ : filter X) (f : X → Y):
  tendsto f F₂ F₁ ↔
    (∀ P, eventually P F₂ → eventually (λ x, P (f x)) F₁) :=
iff.refl _
```

Finding the right concepts

Now define some filters:

- `nhds x`: neighborhoods of `x`
- `[at x within s]`: neighborhoods of `x` intersected with `s \ ’{x}`
- `[at x] := [at x within univ]`
- `[at x-] := [at x within ’(-∞, x)]`
- `[at x+] := [at x within ’(x, ∞)]`
- `[at ∞]`: “neighborhoods” of ∞
- `[at -∞]`: “neighborhoods” of $-\infty$

Finding the right concepts

And some more notation:

```
f ⟶ y    := tendsto f (nhds y)
f ⟶ ∞    := tendsto f [at ∞]
f ⟶ -∞   := tendsto f [at -∞]
```

And now all these mean what they are supposed to mean!

```
f ⟶ y [at x]
f ⟶ y [at ∞]
f ⟶ y [at x-]
f ⟶ ∞ [at x]
f ⟶ -∞ [at ∞]
(λ x, x^2) ⟶ 4 [at 2]
(λ x, x^2) ⟶ ∞ [at -∞]
```

Finding the right concepts

And we can prove theorems at the right level of abstraction.

```
section
  variables {U V : Type} [normed_vector_space V]
  variables {f g : U → V} {a b : V}
  variable  {F : filter U}

  theorem add_approaches
      (Hf : (f ⟶ a) F) (Hg : (g ⟶ b) F) :
    (λ x, f x + g x) ⟶ (a + b) F := ...
end
```

Outline

- Overview
- Assertion languages
- Proof languages
- Structures and the algebraic hierarchy
- Finding the right concepts
- Verifying computation
- Automation

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)

Automated reasoning

Many interactive theorem provers incorporate automation.

Challenges:

- develop more powerful automated tools.
- verify the results (with proof terms, certificates, etc.)
- provide means to customize / control automation.
- combine automated procedures.

Conclusions

Formal verification is primarily concerned with verifying correctness.

Even so, it delivers insights into how mathematics works:

- how we communicate mathematical claims
- how we communicate mathematical proofs
- how we manage and understand mathematical notation
- how we organize mathematical structures
- how we organize mathematical data and theories
- what basic inferences we recognize as valid
- how we carry out nontrivial reasoning steps

Conclusions

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

Designing good theorem provers should go hand in hand with understanding how mathematics works.