Formal Methods in Mathematics and the Lean Theorem Prover

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

January 2017

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)
- checking things (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.
- CompCert has verified the correctness of a C compiler.
- The seL4 microkernel has been verified.
- 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.

I will focus on mathematics:

- Problems are conceptually deeper, less homogeneous.
- 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.

Computation in mathematics

Computation plays an increasing role in pure mathematics:

- Lyons-Sims sporadic simple group of order $2^8 \cdot 3^7 \cdot 5^6 \cdot 7 \cdot 11 \cdot 31 \cdot 37 \cdot 67$ (1973)
- The four color theorem (Appel and Hakens, 1976)
- Feigenbaum's universality conjecture (Lanford, 1982)
- Nonexistence of a finite projective plane of order 10
- The Kepler conjecture (Hales, 1998)
- Catalan's conjecture (Mihăilescu, 2002)
- The existence of the Lorenz attractor (Tucker, 2002)
- Bounds on higher-dimensional sphere packing (Cohn and Elkies, 2003)
- Universal quadratic forms and the 290 theorem (Bhargava and Hanke, 2011)

Other examples

- Weak Goldbach conjecture (Helfgott, 2013)
- Maximal number of exceptional Dehn surgeries (Lackenby and Meyerhoff, 2013)
- Better bounds on gaps in the primes (Polymath 8a, 2014)

See Hales, "Mathematics in the Age of the Turing Machine," and Wikipedia, "Computer assisted proof."

See also computer assisted proofs in special functions (Wilf, Zeilberger, Borwein, Paule, Moll, Stan, Salvy, . . .).

See also anything published by Doron Zeilberger and Shalosh B. Ekhad.

Search in mathematics

There are fewer notable successes in the use of formal search methods.

McCune proved the Robbins conjecture in 1996, using his theorem prover *EQP*.

In 2014, Konev and Lisitsa used a SAT solver to find a sequence of length 1160 with discrepancy 2, and show that no such sequence exists with length 1161.

Recently Heule, Kullmann, and Marek used a SAT solver to show that very two-coloring of the positive natural numbers has a monochromatic Pythagorean triple ($a^2 + b^2 = c^2$).

You can color the numbers up to 7824 without one. The proof that this is impossible for 7825 is almost 200 Terabytes long.

Formal methods in mathematics

I will focus on verification, rather than discovery.

But once again, there isn't a clear separation.

Even if you are primarily interested in automation and computation, it is best to do it within a precise formal framework:

- so you can trust the results, and
- so there is no ambiguity as to what they mean.

Outline

- Formal methods in mathematics
- Interactive theorem proving
- Challenges
- Lean
- The Lean response
- Conclusions

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 substantial 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)
- Lean (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

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 based on dependent type theory.

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 recently 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* (λ*i*. *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* (λ*x*. (*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* ≡ λ*x*. ∑*i*<*n*. *X i x*"
  shows
    "*weak_conv_m* (λ*n*. *distr M borel* (λ*x*. *S n x* / *sqrt* (*n* * σ^2)))
      (*density lborel std_normal_density*)"

Challenges

The main challenges for verified mathematics:

- Developing expressive assertion languages.
- Developing powerful proof languages.
- Verifying computation.
- Developing automation.

Assertion languages

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?

Assertion languages

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.

Assertion languages

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.

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. $\sum_{i \in I} c \cdot a_i = c \cdot \sum_{i \in I} a_i$

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.

Proof languages

Read the definition of *proof* in a logic textbook.

Then take an ordinary mathematics textbook off the shelf, and compare.

There is a mismatch.

Interactive theorem provers have to fill the gap.

Proof languages

```
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
```

Proof languages

```
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 languages

```
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 languages

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.

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

Mathematical referees are not trained to check the correctness of code.

In any case, that can be very hard to do: it is easy make mistakes when implementing a complex algorithm.

Challenge: Develop ways of ensuring that the results of a computation mean what we think they mean.

Automated reasoning

Ideal: given an assertion, φ, either

- provide a proof that φ 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

Automated reasoning

To be effective, automated reasoning relies on heuristics and performance optimizations.

If we want to use the results in mathematics, we want strong assurances that they are correct.

Challenge: Develop powerful automation that is nonetheless trustworthy.

Challenges

Recap of the main challenges:

- Developing expressive assertion languages.
- Developing powerful proof languages.
- Verifying computation.
- Developing automation.

The Lean Theorem Prover

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

- The project began in 2013.
- It was "announced" in the summer of 2015.
- The system underwent a major rewrite in 2016.
- The new version, Lean 3 is in place.
- The main focus now: rebuilding the library, developing automation.

Lean 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
- a programming environment in which one can
  - compute with objects with a precise formal semantics,
  - reason about the results of computation, and
  - write proof-producing automation

The Lean Theorem Prover

Overarching 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
- supports constructive reasoning, quotients and extensionality, and classical reasoning
- elegant syntax and a powerful elaborator
- well-integrated type class inference
- a function definition system compiles recursive definitions down to primitives
- flexible means for writing declarative proofs and tactic-style proofs

The Lean Theorem Prover

Notable features:

- server support for editors, with proof-checking and live information
- Emacs mode in place, VSCode under development
- browser version runs in Javascript (for tutorials)
- fast bytecode evaluator
- monadic interface to Lean internals, for writing tactics and automation
- profiler and roll-your-own debugger
- simplifier with conditional rewriting, arithmetic simplification
- resolution theorem prover written in Lean itself
- *(new!)* SMT-state extends tactics state with congruence closure, e-matching
- enthusiastic, talented people involved

The Lean Theorem Prover

Code base: Leonardo de Moura, Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Daniel Selsam

Standard library: Jeremy Avigad, Floris van Doorn, Leonardo de Moura, Robert Lewis, Gabriel Ebner

Past project members: Soonho Kong, Jakob von Raumer

Contributors: Assia Mahboubi, Cody Roux, Parikshit Khanna, Ulrik Buchholtz, Favonia (Kuen-Bang Hou), Haitao Zhang, Jacob Alexander Gross, Andrew Zipperer

The Lean response

Recall the main challenges:

- Developing expressive assertion languages.
- Developing powerful proof languages.
- Verifying computation.
- Developing automation.

Simple type theory

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

```
check ℕ    -- Type
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 p.1
check m + 2 * n + 7
check F f
check F (g m)
check f (p, (g m, n)).1.2
```

Simple type theory

Modern variants include variables ranging over types and type constructors.

```
variables α β : Type

check list α
check set α
check α × β
check α × ℕ

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

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

Dependent type theory

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

```
variables (α : Type) (m n : ℕ)

check tuple α n
check matrix ℝ m n
check Zmod n

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

check s ++ t     -- tuple α (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 α β : Type

check @prod    -- Type u_1 → Type u_2 → Type (max
  1 u_1 u_2)
check @list    -- Type u_1 → Type (max 1 u_1)

check prod α β
check list α
check prod α ℕ
check list (prod α ℕ)
```

Dependent type theory

Dependent type theory is robust enough to define algebraic structures and relationships between them in a nice way.

```
structure Semigroup : Type :=
(carrier : Type)
(mul : carrier → carrier → carrier)
(mul_assoc :
    ∀ a b c, mul (mul a b) c = mul a (mul b c))
```

We can use class inference to handle instances and inheritance automatically.

Dependent type theory

```
class semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c, a * b * c = a * (b * c))

class monoid (α : Type u) extends semigroup α, has_one α :=
(one_mul : ∀ a, 1 * a = a) (mul_one : ∀ a, a * 1 = a)

def pow {α : Type u} [monoid α] (a : α) : ℕ → α
| 0     := 1
| (n+1) := a * pow n

theorem pow_add {α : Type u} [monoid α] (a : α) (m n : ℕ) :
  a^(m + n) = a^m * a^n :=
begin
  induction n with n ih,
  { simp [add_zero, pow_zero, mul_one] },
  rw [add_succ, pow_succ', ih, pow_succ', mul_assoc]
end

instance : linear_ordered_comm_ring int := ...
```

The Lean response

Challenge #1: Develop expressive assertion languages.

The Lean response:

- Use dependent type theory (with inductive types) to define mathematical objects.
- Use type class inference to manage the algebraic hierarchy and coercions.

Propositions and proofs in dependent type theory

In simple type theory, we distinguish bteween

- 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  α : Type
variable  S : α → Prop
variable  R : α → α → Prop

local infix ` ≺ `:50 := R

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

Encoding proofs

Given `(p : Prop)`, view `(t : p)` as saying that t is a proof of P.

```
theorem and_comm : 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_comm' : p ∧ q → q ∧ p :=
λ h, ⟨h^.right, h^.left⟩

check and_comm -- ∀ {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 rw [-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 rw [gcd_eq_one_of_coprime co at this]; exact this,
show false,
  from absurd <2 ∣ 1> dec_trivial
```

The Lean response

Challenge #2: Develop powerful proof languages.

The Lean response:

- Use dependent type theory, providing a uniform approach to writing expressions.
- Provide helpful syntax for common proof constructs.
- Use type classes and automation to manage details.

Verifying code

We can use Lean as a programming language:

```
def fib : ℕ → ℕ
| 0     := 1
| 1     := 1
| (n+2) := fib (n+1) + fib n

vm_eval fib 1000
```

Verifying code

```
universe variable u
parameters {α : Type u} (r : α → α → Prop) [decidable_rel r]
local infix ≼ := r

def ordered_insert (a : α) : list α → list α
| []       := [a]
| (b :: l) := if a ≼ b then a :: (b :: l) else b :: ordered_insert l

def insertion_sort : list α → list α
| []       := []
| (b :: l) := ordered_insert b (insertion_sort l)

vm_eval insertion_sort (λ m n : ℕ, m ≤ n)
          [5, 27, 221, 95, 17, 43, 7, 2, 98, 567, 23, 12]
```

Verifying code

There is good support for Haskell-style monads.

Programming in Lean feels like programming in any (pure) functional programming language.

The bytecode evaluator is quite efficient.

Jared Roesch is working on extraction to C++.

The Lean response

Challenge #3: Verifying code.

The Lean response:

- We can write and reason about code in Lean.
- The bytecode evaluator is efficient, though it requires a measure of trust.
- Extraction to C++ requires more trust.
- For the strongest guarantees, we can write proof-producing code in Lean.

Developing automation

In Lean, core functions are written in C++, as well as performance-critical automation.

- There is a library of core tactics.
- There is already a good term rewriter (simplifier).
- A special SMT-state, with support for congruence closure and e-matching, is showing signs of life.

But we can also write automation in Lean.

A monadic interface provides access to Lean internals.

Gabriel Ebner has written a superposition theorem prover in this way.

Developing automation

The `meta` keyword indicates we are outside the formal system.

```
meta def contra_aux : list expr → list expr → tactic unit
| []          hs := failed
| (h₁ :: rs) hs :=
  do t₀ ← infer_type h₁,
     t  ← whnf t₀,
     (do a ← match_not t,
         h₂ ← find_same_type a hs,
         tgt ← target,
         pr ← mk_app `absurd [tgt, h₂, h₁],
         exact pr)
     <|> contra_aux rs hs

meta def contra : tactic unit :=
do ctx ← local_context,
   contra_aux ctx ctx
```

The kernel still checks all results.

The Lean response

Challenge #4: Developing automation.

The Lean response:

- Develop a powerful API in C++.
- Develop performant, general purpose automation in C++.
- Use Lean as a metaprogramming languages, for smaller tactics and specialized automation.
- Extract C++ code for better performance.

The goal is to have both performance and flexibility.

A good library consists not just of definitions and theorems, but also expertise: procedures, methods, heuristics.

Conclusions

Shankar: "We are in the golden age of metamathematics."

Formal methods will have a transformative effect on mathematics.

Computers change the kinds of proofs that we can discover and verify.

In other words, they enlarge the scope of what we can come to know.

It will take clever ideas, and hard work, to understand how to use them effectively.

But it's really exciting to see it happen.