Formal Methods in Mathematics and the Lean Theorem Prover

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

July 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.
- Aesthetic Integration uses formal methods to verify properties of trading algorithms executed by exchanges.

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
- The Lean Theorem Prover
- 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)
- Agda (constructive dependent type theory)
- Metamath (set 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

Thomas 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 provides a synthetic approach to algebraic topology.

- 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 has verified 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.
- Developing better user interfaces and tools.

The Lean Theorem Prover

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

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

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

The Lean Theorem Prover

Why develop a new theorem prover?

- It provides a fresh start.
- We can incorporate the best ideas from existing provers, and try to avoid shortcomings.
- We can craft novel engineering solutions to design problems.

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,
  - extend the capabilities of Lean itself,
  - write proof-producing automation

The Lean Theorem Prover

Overarching goals:

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

History

- The project began in 2013.
- Lean 2 was "announced" in the summer of 2015.
- A major rewrite was undertaken in 2016.
- The new version, Lean 3 is in place.
- A standard library and automation are under development.
- HoTT development is ongoing in Lean 2.

People

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

Libraries: Jeremy Avigad, Floris van Doorn, Leonardo de Moura, Robert Lewis, Gabriel Ebner, Johannes Hölzl, Mario Carneiro

Past project members: Soonho Kong, Jakob von Raumer

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

The Lean Theorem Prover

Notable features:

- based on a powerful dependent type theory
- written in C++, with multi-core support
- small trusted kernel with independent type checkers
- 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 structural / nested / mutual / well-founded recursive definitions down to primitives
- flexible means of writing declarative proofs and tactic-style proofs
- server support for editors, with proof-checking and live information

The Lean Theorem Prover

- editor modes for Emacs and VSCode
- a javascript version runs in a browser
- a fast bytecode interpreter for evaluating computable definitions
- a powerful framework for metaprogramming via a monadic interface to Lean internals
- profiler and roll-your-own debugger
- simplifier with conditional rewriting, arithmetic simplification
- SMT-state extends tactics state with congruence closure, e-matching
- online documentation and courseware
- enthusiastic, talented people involved

Logical Foundations

Lean is based on a version of the Calculus of Inductive Constructions, with:

- a hierarchy of (non-cumulative) universes, with a type *Prop* of propositions at the bottom
- dependent function types (Pi types)
- inductive types (à la Dybjer)

Semi-constructive axioms and constructions:

- quotient types (the existence of which imply function extensionality)
- propositional extensionality

A single classical axiom:

- choice

Defining Functions

Lean's primitive recursors are a very basic form of computation.

To provide more flexible means of defining functions, Lean uses an *equation compiler*.

It does pattern matching:

```
def list_add {α : Type u} [has_add α] :
  list α → list α → list α
| [] _                 := []
| _  []                := []
| (a :: l) (b :: m) := (a + b) :: list_add l m

#eval list_add [1, 2, 3] [4, 5, 6, 6, 9, 10]
```

Defining Functions

It handles arbitrary structural recursion:

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

#eval fib 10000
```

It detects impossible cases:

```
def vector_add [has_add α] :
  Π {n}, vector α n → vector α n → vector α n
| ._  nil              nil         := nil
| ._  (@cons ._ _ a v) (cons b w) := cons (a + b)
                                          (vector_add v w)

#eval vector_add (cons 1 (cons 2 (cons 3 nil)))
                 (cons 4 (cons 5 (cons 6 nil)))
```

Defining Inductive Types

Nested and mutual inductive types are also compiled down to the primitive versions:

```
mutual inductive even, odd
with even : ℕ → Prop
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)

inductive tree (α : Type)
| mk : α → list tree → tree
```

Defining Functions

The equation compiler handles nested inductive definitions and mutual recursion:

```
inductive term
| const : string → term
| app   : string → list term → term

open term

mutual def num_consts, num_consts_lst
with num_consts : term → nat
| (term.const n)  := 1
| (term.app n ts) := num_consts_lst ts
with num_consts_lst : list term → nat
| []      := 0
| (t::ts) := num_consts t + num_consts_lst ts

def sample_term := app "f" [app "g" [const "x"], const "y"]

#eval num_consts sample_term
```

Defining Functions

We can do well-founded recursion:

```
def div : nat → nat → nat
| x y :=
  if h : 0 < y ∧ y ≤ x then
    have x - y < x, from ...,
    div (x - y) y + 1
  else
    0
```

Here is Ackermann's function:

```
def ack : nat → nat → nat
| 0     y     := y+1
| (x+1) 0     := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)
```

Type Class Inference

Type class resolution is well integrated.

```
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

infix `^`:= pow
```

Type Class Inference

```
@[simp] theorem pow_zero (a : α) : a^0 = 1 := by unfold pow

theorem pow_succ (a : α) (n : ℕ)  : a^(n+1) = a * a^n :=
by unfold pow

theorem pow_mul_comm' (a : α) (n : ℕ) : a^n * a = a * a^n :=
by induction n with n ih; simp [*, pow_succ]

theorem pow_succ' (a : α) (n : ℕ) : a^(n+1) = a^n * a :=
by simp [pow_succ, pow_mul_comm']

theorem pow_add (a : α) (m n : ℕ) : a^(m + n) = a^m * a^n :=
by induction n; simp [*, pow_succ', nat.add_succ]

theorem pow_mul_comm (a : α) (m n : ℕ) :
  a^m * a^n = a^n * a^m :=
by simp [(pow_add a m n).symm, (pow_add a n m).symm]

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

Proof Language

Proofs can be written as terms, or using *tactics*.

```
def gcd : nat → nat → nat
| 0        y := y
| (succ x) y := have y % succ x < succ x,
                  from mod_lt _ $ succ_pos _,
                gcd (y % succ x) (succ x)

theorem gcd_dvd_left (m n : ℕ) : gcd m n ∣ m := ...

theorem gcd_dvd_right (m n : ℕ) : gcd m n ∣ n := ...

theorem dvd_gcd {m n k : ℕ} : k ∣ m → k ∣ n → k ∣ gcd m n := ...
```

Proof Language

```
theorem gcd_comm (m n : ℕ) : gcd m n = gcd n m :=
dvd_antisymm
  (have h₁ : gcd m n ∣ n, from gcd_dvd_right m n,
    have h₂ : gcd m n ∣ m, from gcd_dvd_left m n,
    show gcd m n ∣ gcd n m, from dvd_gcd h₁ h₂)
  (have h₁ : gcd n m ∣ m, from gcd_dvd_right n m,
    have h₂ : gcd n m ∣ n, from gcd_dvd_left n m,
    show gcd n m ∣ gcd m n, from dvd_gcd h₁ h₂)

theorem gcd_comm₁ (m n : ℕ) : gcd m n = gcd n m :=
dvd_antisymm
  (dvd_gcd (gcd_dvd_right m n) (gcd_dvd_left m n))
  (dvd_gcd (gcd_dvd_right n m) (gcd_dvd_left n m))
```

Proof Language

```
theorem gcd_comm₂ (m n : ℕ) : gcd m n = gcd n m :=
suffices ∀ {m n}, gcd m n ∣ gcd n m,
  from (dvd_antisymm this this),
assume m n : ℕ,
show gcd m n ∣ gcd n m,
  from dvd_gcd (gcd_dvd_right m n) (gcd_dvd_left m n)

theorem gcd_comm₃ (m n : ℕ) : gcd m n = gcd n m :=
begin
  apply dvd_antisymm,
  { apply dvd_gcd, apply gcd_dvd_right, apply gcd_dvd_left },
  apply dvd_gcd, apply gcd_dvd_right, apply gcd_dvd_left
end
```

Proof Language

```
theorem gcd_comm₄ (m n : ℕ) : gcd m n = gcd n m :=
begin
  apply dvd_antisymm,
  { have : gcd m n ∣ n, apply gcd_dvd_right,
    have : gcd m n ∣ m, apply gcd_dvd_left,
    show gcd m n ∣ gcd n m, apply dvd_gcd; assumption },
  { have : gcd n m ∣ m, apply gcd_dvd_right,
    have : gcd n m ∣ n, apply gcd_dvd_left,
    show gcd n m ∣ gcd m n, apply dvd_gcd; assumption },
end

theorem gcd_comm₅ (m n : ℕ) : gcd m n = gcd n m :=
by apply dvd_antisymm;
   {apply dvd_gcd, apply gcd_dvd_right, apply gcd_dvd_left}
```

Proof Language

```
attribute [ematch] gcd_dvd_left gcd_dvd_right dvd_gcd
    dvd_antisymm

theorem gcd_comm₆ (m n : ℕ) : gcd m n = gcd n m :=
by finish

theorem gcd_comm₇ (m n : ℕ) : gcd m n = gcd n m :=
begin [smt] eblast end
```

Lean as a Programming Language

Lean implements a fast bytecode evaluator:

- It uses a stack-based virtual machine.
- It erases type information and propositional information.
- It uses eager evaluation (and supports delayed evaluation with thunks).
- You can use anything in the Lean library, as long as it is not `noncomputable`.
- The machine substitutes native nats and ints (and uses GMP for large ones).
- It substitutes a native representation of arrays.
- It has a profiler and a debugger.
- It is really fast.

Compilation to native code is under development.

Lean as a Programming Language

```
#eval 3 + 6 * 27
#eval if 2 < 7 then 9 else 12
#eval [1, 2, 3] ++ 4 :: [5, 6, 7]
#eval "hello " ++ "world"
#eval tt && (ff || tt)

def binom : ℕ → ℕ → ℕ
| _     0     := 1
| 0     (_+1) := 0
| (n+1) (k+1) := if k > n then 0
                 else if n = k then 1
                 else binom n k + binom n (k+1)

#eval (range 7).map $ λ n, (range (n+1)).map $ λ k, binom n k
```

Lean as a Programming Language

```
section sort
universe variable u
parameters {α : Type u} (r : α → α → Prop) [decidable_rel r]
local infix ≼ : 50 := 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)

end sort

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

Lean as a Programming Language

There are algebraic structures that provides an interface to terminal and file I/O.

Users can implement their own, or have the virtual machine use the "real" one.

At some point, we decided we should have a package manager to manage libraries and dependencies.

Gabriel Ebner wrote one, in Lean.

Lean as a Metaprogramming Language

Question: How can one go about writing tactics and automation?

Various answers:

- Use the underlying implementation language (ML, OCaml, C++, ...).
- Use a domain-specific tactic language (LTac, MTac, Eisbach, ...).
- Use reflection (RTac).

Metaprogramming in Lean

Our answer: go meta, and use the object language.

(MTac, Idris, and now Agda do the same, with variations.)

Advantages:

- Users don't have to learn a new programming language.
- The entire library is available.
- Users can use the same infrastructure (debugger, profiler, etc.).
- Users develop metaprograms in the same interactive environment.
- Theories and supporting automation can be developed side-by-side.

Metaprogramming in Lean

The method:

- Add an extra (meta) constant: `tactic_state`.
- Reflect expressions with an `expr` type.
- Add (meta) constants for operations which act on the tactic state and expressions.
- Have the virtual machine bind these to the internal representations.
- Use a tactic monad to support an imperative style.

Definitions which use these constants are clearly marked `meta`, but they otherwise look just like ordinary definitions.

Metaprogramming in Lean

```
meta def find : expr → list expr → tactic expr
| e []        := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx ← local_context,
     t   ← target,
     h   ← find t ctx,
     exact h }
<|> fail "assumption tactic failed"

lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Metaprogramming in Lean

Summary:

- We extend the object language with a type that reflects an internal tactic state, and expose operations that act on the tactic state.
- We reflect the syntax of dependent type theory, with mechanisms to support quotation and pattern matching over expressions.
- We use general support for monads and monadic notation to define the tactic monad and extend it as needed.
- We have an extensible way of declaring attributes and assigning them to objects in the environment (with caching).
- We can easily install tactics written in the language for use in interactive tactic environments.
- We have a profiler and a debugging API.

Metaprogramming in Lean

The metaprogramming API includes a number of useful things, like an efficient implementation of red-black trees.

Tactics are fallible – they can fail, or produce expressions that are not type correct.

Every object is checked by the kernel before added to the environment, so soundness is not compromised.

Metaprogramming in Lean

Most of Lean's tactic framework is implemented in Lean.

Examples:

- The usual tactics: assumption, contradiction, ...
- Tactic combinators: `repeat`, `first`, `try`, ...
- Goal manipulation tactics: `focus`, ...
- A procedure which establishes decidable equality for inductive types.
- A transfer method (Hölzl).
- Translations to/from Mathematica (Lewis).
- A full-blown superposition theorem prover (Ebner).

The method opens up new opportunities for developing theories and automation hand in hand.

Metaprogramming in Lean

Having a programming language built into a theorem prover is incredibly flexible.

We can:

- Write custom automation.
- Develop custom tactic states (with monad transformers) and custom interactive frameworks.
- Install custom debugging routines.
- Write custom parser extensions.

Challenges

The main challenges for verified mathematics:

- Developing expressive assertion languages.
- Developing powerful proof languages.
- Verifying computation.
- Developing automation.
- Developing better user interfaces and tools.

Our reponse

The main challenges for verified mathematics:

- *Developing expressive assertion languages*: use dependent type theory, with type classes, carefully designed abstractions, an equation compiler, nice syntax.
- *Developing powerful proof languages*: use proof terms and tactics, custom automation
- *Verifying computation*: verify computations step by step for the highest degree of trust; extract verified code for performance.
- *Developing automation*: use native implementations for performance, but expose internals and use metaprogramming for flexibility ("whitebox automation").
- *Developing better user interfaces and tools* : use metaprogramming, and make everything extensible: automation, the parser, the debugger, etc.

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.

References

Documentation, papers, and talks on are `leanprover.github.io`.

There are various talks on my web page.

See especially the paper "A metaprogramming framework for formal verification," to appear in the proceedings of ICFP 2017.