The Lean Theorem Prover

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

June 29, 2017

Outline

Contents:

- An overview of Lean
- Metaprogramming in Lean

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

Some systems currently in use with substantial mathematical libraries:

- Mizar (set theory)
- HOL4 (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)
- Adga (constructive dependent type theory)

Why develop another?

The Lean Theorem Prover

Why develop another?

- 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, and
  - write proof-producing automation

The Lean Theorem Prover

Overarching goals:

- Verify mathematics.
- Verify hardware, software, and hybrid systems.
- 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

Logical Foundations

Lean has a hierarchy of non-cumulative type universes:

```
Sort 0, Sort 1, Sort 2, Sort 3, ...
```

These can also be written:

```
Prop, Type 0, Type 1, Type 2, ...
```

`Prop` is impredicative and definitionally proof irrelevant.

The latter means that if `p : Prop`, `s : p`, and `t : p`, then `s` and `t` are definitionally equal.

Logical Foundations

We have dependent function types `Π x : α, β x` with the usual reduction rule, `(λ x, t) s = t [s / x]`.

We have eta equivalence for functions: `t` and `λ x, t x` are definitionally equal.

We also have "let" definitions, `let x := s in t`, with the expected reduction rule.

Logical Foundations

Lean implements inductive families with primitive recursors, and the expected computation rules.

```
inductive vector (α : Type u) : ℕ → Type u
| nil : vector 0
| cons {n : ℕ} (a : α) (v : vector n) : vector (n+1)

#check (vector : Type u → ℕ → Type u)
#check (vector.nil : Π α : Type u, vector α 0)
#check (@vector.cons : Π {α : Type u} {n : ℕ},
  α → vector α n → vector α (n + 1))
#check (@vector.rec :
  Π {α : Type u} {C : Π (n : ℕ), vector α n → Sort u},
      C 0 (vector.nil α) →
      (Π {n : ℕ} (a : α) (v : vector α n), C n v →
                              C (n + 1) (vector.cons a v)) →
      Π {n : ℕ} (v : vector α n), C n v)
```

Logical Foundations

We can quotient by an arbitrary binary relation:

```
constant quot :
  Π {α : Sort u}, (α → α → Prop) → Sort u
constant quot.mk :
  Π {α : Sort u} (r : α → α → Prop), α → quot r
axiom quot.ind :
  ∀ {α : Sort u} {r : α → α → Prop} {β : quot r → Prop},
    (∀ a, β (quot.mk r a)) → ∀ (q : quot r), β q
constant quot.lift :
  Π {α : Sort u} {r : α → α → Prop}
    {β : Sort u} (f : α → β),
    (∀ a b, r a b → f a = f b) → quot r → β
axiom quot.sound :
  ∀ {α : Type u} {r : α → α → Prop} {a b : α},
    r a b → quot.mk r a = quot.mk r b
```

These (with eta) imply function extensionality.

Logical Foundations

Propositional extensionality:

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

Finally, we can go classical:

```
axiom choice {α : Sort u} : nonempty α → α
```

Here, `nonempty α` is equivalent to `∃ x : α, true`.

Diaconescu's trick gives us the law of the excluded middle.

Definitions that use choice to produce data are `noncomputable`.

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 sorry,
    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)
```

Defining Functions

Here is another example:

```
def nat_to_bin : ℕ → list ℕ
| 0       := [0]
| 1       := [1]
| (n + 2) :=
  have (n + 2) / 2 < n + 2, from sorry,
  (nat_to_bin ((n + 2) / 2)).concat (n % 2)

#eval nat_to_bin 1234567
```

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

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 := ...
```

Syntactic Gadgets

We have a number of nice syntactic gadgets:

- Anonymous constructors and projections.
- Uniform notation for records.
- Pattern matching with assumptions and `let`.
- Monadic *do* notation.
- Default arguments, optional arguments, thunks, tactic handlers.

Syntactic Gadgets: Anonymous Constructors

```
example : p ∧ q → q ∧ p :=
λ h, and.intro (and.right h) (and.left h)

example : p ∧ q → q ∧ p :=
λ h, ⟨h.right, h.left⟩

#eval list.map (λ n, n * n) (range 10)

#eval (range 10).map $ λ n, n * n
```

Syntactic Gadgets: Record Notation

```
structure color :=
mk :: (red : nat := 0) (green : nat := 0) (blue : nat := 0)

#check color.mk 100 200 150
#check { color . red := 100, green := 200, blue := 203 }

-- you can omit the name of the structure when it can be inferred
def hot_pink : color := { red := 255, green := 192, blue := 203 }
def my_color : color := ⟨100, 200, 150⟩

-- defaults are used when omitted
example : { color . red := 100 }.blue = 0 := rfl

-- notation for record update
def lavender := { hot_pink with green := 20 }

#eval lavender.red
#eval lavender.green
```

Syntactic Gadgets: Pattern Matching

```
example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y
| ⟨x, px⟩ ⟨y, qy⟩ := ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) : ∃ x y, p x ∧ q y :=
match h₀, h₁ with
  ⟨x, px⟩, ⟨y, qy⟩ := ⟨x, y, px, qy⟩
end

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) : ∃ x y, p x ∧ q y :=
let ⟨x, px⟩ := h₀,
    ⟨y, qy⟩ := h₁ in
⟨x, y, px, qy⟩

example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y :=
λ ⟨x, px⟩ ⟨y, qy⟩, ⟨x, y, px, qy⟩
```

Syntactic Gadgets: Monads

```
variables (l : list α) (f : α → list β)
variables (g : α → β → list γ) (h : α → β → γ → list δ)

example : list δ :=
do a ← l,
   b ← f a,
   c ← g a b,
   h a b c
```

Lean instantiates:

- the option monad
- the list monad
- the state monad
- a tactic state monad for metaprogramming
- monad transformers (especially: adding state)

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 LLVM 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

To make this practical, we need efficient means of dealing with syntactic objects like:

- *names*: hierarchical identifiers, like `x`, `list.reverse`, `nat.zero_le`.
- *expressions*
- *pre-expressions*: "raw" expressions, waiting to be elaborated

Metaprogramming in Lean

Here is the `expr` type:

```
meta inductive expr (elaborated : bool := tt)
| var       {} : nat → expr
| sort      {} : level → expr
| const     {} : name → list level → expr
| mvar          : name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app           : expr → expr → expr
| lam           : name → binder_info → expr → expr → expr
| pi            : name → binder_info → expr → expr → expr
| elet          : name → expr → expr → expr → expr
| macro         : macro_def → list expr → expr
```

The virtual machine substitutes the internal representation.

The are various ways to "quote" expressions.

Metaprogramming in Lean

We can elaborate an expression when the tactic is defined:

```
example : true ∧ true :=
by do apply `(and.intro trivial trivial)
```

Conceptually, we are doing something like this:

```
meta def foo : tactic unit :=
do apply `(and.intro trivial trivial)

example : true ∧ true :=
by foo
```

The keyword `by` can be used to invoke a metaprogram anywhere a term is expected.

The tactic is invoked on the current tactic state, which includes the environment and current goal.

Metaprogramming in Lean

We can, alternatively, elaborate a pre-expression when the tactic is *executed*:

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, to_expr ``(or.inl %%e) >>= apply
```

Here, `%%e` is an *antiquotation*.

Alternatively, the `refine` tactic takes a pre-expression:

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, refine ``(or.inl %%e)
```

Metaprogramming in Lean

We can even defer name resolution to execution time:

```
example (p : Prop) : p → p ∨ false :=
by do intro `h, to_expr ```(or.inl h) >>= apply
```

Note that `h` doesn't "exist" until the tactic is executed.

In fact, tactics can be installed to run in an interactive mode:

```
example (p : Prop) : p → p ∨ false :=
begin
  intro h,
  apply or.inl h
end
```

The arguments are parsed as `` `h `` and ```` ```(or.inl h) ````, and the interactive versions of `intro` and `apply` do the right thing.

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 can be 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.

Conclusions

Summing up:

- We are off to a good start: we have an expressive logical framework, efficient infrastructure, nice syntax, a powerful elaborator, and a solid basis for user interaction.
- Automation (the simplifier, an smt state, etc.) is showing signs of life.
- The metaprogramming facilities offer exciting opportunities:
  - The development of theories and procedural expertise hand-in-hand.
  - The development of customized back-end automation for other tools and systems.
- There is a tremendous amount of work to do. If you are looking for a worthy cause, look no further.

References

These slides are on my web page.

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

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