Formalized mathematics in the Lean proof assistant

Robert Y. Lewis

Vrije Universiteit Amsterdam

CARMA Workshop on Computer-Aided Proof
Newcastle, NSW
June 6, 2019

Credits

Thanks to the following people for some of the contents of this talk:

- Leonardo de Moura
- Jeremy Avigad
- Mario Carneiro
- Johannes Hölzl

Table of Contents

Proof assistants in mathematics

Computers in mathematics

Mathematicians use computers in various ways.

- typesetting
- numerical calculations
- symbolic calculations
- visualization
- exploration

Let's add to this list: checking proofs.

Interactive theorem proving

We want a language (and an implementation of this language) for:

- Defining mathematical objects
- Stating properties of these objects
- Showing that these properties hold
- Checking that these proofs are correct
- Automatically generating these proofs

This language should be:

- Expressive
- User-friendly
- Computationally efficient

Interactive theorem proving

Working with a proof assistant, users construct definitions, theorems, and proofs.

The proof assistant makes sure that definitions are well-formed and unambiguous, theorem statements make sense, and proofs actually establish what they claim.

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

Much of the system is “untrusted.” Only a small core has to be relied on.

Interactive theorem proving

Some systems with large mathematical libraries:

- Mizar (first order set theory)
- HOL/HOL Light (simple type theory + higher order logic)
- Isabelle (simple type theory + higher order logic)
- Coq (constructive dependent type theory)
- ACL2 (primitive recursive arithmetic)
- PVS (classical dependent type theory)
- Lean (constructive dependent type theory)

Interactive theorem proving

- Set theory: familiar, well-studied, very inefficient computation
- Simple type theory: computationally powerful, flexible, less expressive
- Dependent type theory: very expressive, beautiful mathematical theory, subtle and finicky

Dependent type theory

Simple type theory

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

```
#check ℕ
#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 F f
#check λ x : ℕ, m
#check (λ x : ℕ, m) n
```

Terms are either constants, variables, applications, or lambda abstractions.

Dependent type theory

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

```
variables (A : Type) (m n : ℕ)

#check tuple A n      -- Type
#check matrix ℝ m n   -- Type
#check Zmod n         -- Type

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.

So we can write down complex expressions for types, just as we can write down complex expressions for values.

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            -- Type
#check list A              -- Type
#check list (prod A ℕ)     -- Type

#check tuple.empty         -- Π A : Type, tuple A 0
```

Terms are either constants, variables, applications, lambda abstractions, type universes, or Π abstractions.

Inductive types

```
inductive empty : Type

inductive unit : Type
| star : unit

inductive bool : Type
| tt : bool
| ff : bool

inductive prod (A B : Type)
| mk : A → B → prod A B

inductive sum (A B : Type)
| inl : A → sum A B
| inr : B → sum A B
```

Inductive types

The more interesting ones are recursive:

```
inductive nat : Type
| zero : nat
| succ : nat → nat

inductive list (A : Type) : Type
| nil {} : list A
| cons : A → list A → list A
```

Every inductive type comes with a recursor, with computation rules.

```
#check false.rec_on -- ∀ C : Prop, false → C

def nat.double : ℕ → ℕ
| 0       := 0
| (n + 1) := n + n + 2

#check (rfl : nat.double 4 = 8)
```

Inductive propositions

```
inductive false : Prop

inductive true : Prop
| trivial : true

inductive and (A B : Prop)
| intro : A → B → and A B

inductive or (A B : Prop)
| inl : A → or A B
| inr : B → or A B

#check trivial    -- true
```

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 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
```

One language fits all

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.

It can also encode *programs*, since terms have computational meaning.

The Lean theorem prover

Background: Lean

Lean is a new(ish) proof assistant, developed principally by Leonardo de Moura at Microsoft Research, Redmond.

Calculus of inductive constructions with:

- Non-cumulative hierarchy of universes
- Impredicative `Prop`
- Quotient types and propositional extensionality
- Axiom of choice available
- Powerful and accessible tactic language

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

Lean system contributors

- Leonardo de Moura
- Gabriel Ebner
- Soonho Kong
- Jared Roesch
- Daniel Selsam
- Sebastian Ullrich

(Any omissions not intentional!)

Lean 4

The current version is Lean 3.

Lean 4: coming 20?? (pre-release version available)

Lean’s mathematical libraries

mathlib origins

In mid 2017, the "standard library" was split off from the Lean system code base.

Motivation: separate the duties of system and library development.

The standard library became `mathlib` (but isn't just math).

mathlib development

`mathlib` is a community-driven effort with contributors from many different backgrounds.

Maintainers:

- Mario Carneiro
- Jeremy Avigad
- Reid Barton
- Johan Commelin
- Sébastien Gouëzel
- Simon Hudon
- Chris Hughes
- Robert Y. Lewis
- Patrick Massot

Major contributors:

- Seul Baek
- Kevin Buzzard
- Floris van Doorn
- Keeley Hoek
- Johannes Hölzl
- Kenny Lau
- Scott Morrison
- Neil Strickland

Coordination happens on GitHub and in the leanprover Zulip chat room.

mathlib contents

Contents:

- algebraic structures and theories
- analysis (incl. Frechét derivative on normed vector spaces)
- measure and probability theory
- linear algebra (incl. matrices)
- number theory (Pell equations, p-adic numbers)
- set theory (model of ZFC)
- topology (filters, uniform spaces)
- tactics (algebraic normalization, linear arithmetic)

The library is growing fast.

- early 2018: 50k loc
- early 2019: 120k loc
- last week: 150k loc

Type classes

mathlib is designed with a heavy reliance on *type classes* to manage abstract structures.

- algebraic hierarchy
- topological structure
- linear structure
- metric structure
- coercions, embeddings
- decidability

Formalizations based on mathlib

- Jesse Han and Floris van Doorn formalized the unprovability of the continuum hypothesis in ZFC.
- Kevin Buzzard, Johan Commelin, and Patrick Massot formalized the definition of a perfectoid space.
- Tom Hales is using Lean as the basis for his Formal Abstracts project.
- Kevin Buzzard has been teaching undergrad mathematicians using Lean (to great success).
- Sander Dahmen, Johannes Hölzl, and I formalized Ellenberg and Gijswijt's solution to the cap set problem. (part of the Lean Forward project in Amsterdam)
- `https://github.com/leanprover-community/mathlib/blob/100-thms/docs/100-theorems.md`

Intermission: Lean in action

Automation in Lean

Lean as a programming language

Think of `+` as a program. An expression like `12 + 45` will *reduce* or *evaluate* to `57`.

But `+` is defined as unary addition – inefficient!

Lean implements a virtual machine which performs fast, untrusted evaluation of Lean expressions.

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

Lean’s built-in package manager is implemented entirely in Lean.

Lean as a programming language

Definitions tagged with `meta` are "VM only," and allow unchecked recursive calls.

```
meta def f : ℕ → ℕ
| n := if n=1 then 1
       else if n%2=0 then f (n/2)
       else f (3*n + 1)

#eval (list.iota 1000).map f
```

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

Lean's answer: go meta, and use Lean itself.

(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

```
meta def p_not_p : list expr → list expr → tactic unit
| []          Hs := failed
| (H1 :: Rs) Hs :=
  do t ← infer_type H1,
     (do a   ← match_not t,
         H2  ← find_same_type a Hs,
         tgt ← target,
         pr  ← mk_app ‘absurd [tgt, H2, H1],
         exact pr)
     <|> p_not_p Rs Hs

meta def contradiction : tactic unit :=
do ctx ← local_context,
   p_not_p ctx ctx

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

The good and the bad

The good

- Formal specification removes all ambiguity in definitions.
- It can bring subtle questions to the surface. (e.g. canonical isomorphisms)
- As proofs become more computational, formalization increases the reliability.
- As automation gets better, proof assistants can really assist.
- There are interesting educational applications.
- There are many uses of a maintained formal library.
- It’s fun. (Addictive? Masochistic?)

The bad

- Formalization is hard. (Expensive. Frustrating.)
- The automation isn't ready yet.
- There's no room for mathematical creativity.
- The standard of rigor is too strong.
- There are too many proof assistants, too many libraries.

The future?

How to efficiently, effectively formalize modern mathematics?

Still an open research subject.

References

- Lean: `http://leanprover.github.io/`
- *Theorem Proving in Lean* tutorial: `https://leanprover.github.io/theorem_proving_in_lean/`
- mathlib: `https://github.com/leanprover-community/mathlib`
- Formal Abstracts: `https://formalabstracts.github.io/`
- Lean Forward: `https://lean-forward.github.io/`
- Lean Together workshop: `https://lean-forward.github.io/lean-together/2019/`
- Zulip chat room: `https://leanprover.zulipchat.com/`
- Kevin Buzzard's Xena project: `https://xenaproject.wordpress.com/`
- Patrick Massot's lean_format: `https://www.math.u-psud.fr/~pmassot/lean/`