Proving Gauss's Lemma in Lean

Aditya Agarwal

Australian National University

March 1, 2019

Outline

What is Lean?

- Lean is an Automated/Interactive theorem prover.
- Other examples include HOL and Coq.
- You encode proofs in a formal language and Lean checks it's correctness
- Lean can also infer some of the details of the proof for you.

Proofs in Lean

There are two ways of writing proofs in Lean, Term mode and Tactic mode.

Term Mode

In term mode you explicitly write down every step of the proof.

Example:

```
theorem zero_id (x : N) : plus z x = x :=
N.rec_on x
    (show plus z z = z, by refl)
    (assume x,
        assume ih: plus z x = x,
        show plus z (s x) = (s x), from calc
            plus z (s x) = s (plus z x) : by refl
                    ... = s x : by rw ih)
```

Tactics Mode

Tactics mode allows you to use "tactics", which direct Lean on how it should work towards the next step.

Example:

```
lemma not_const_imp_non_zero  : ¬is_const p → p ≠ (0 : polynomial α)
:= begin
  intro hp,
  show is_const p, by {rw [hp, is_const], simp}
end
```

Dependent Type Theory

Lean is built on Dependent Type Theory, an alternative foundation of mathematics. Specifically, it implements the Calculus of Inductive Constructions.

Dependent Type Theory

- Each object has a type.
- E.g.
  - $n : \mathbb{N}$
  - $p \iff p : Prop$
- This type may depend on a parameter such as
  - list α
  - polynomial α
- The type of this object is fixed

Mathlib

Mathlib is the mathematical components library for Lean.

- It contains the very beginnings of Algebra, Analysis, Topology, and Category Theory.
- On the Algebra side, it contains basic definitions about Groups, Rings, Modules etc.
- And properties about Integral Domains, Euclidean Domains, Unique Factorisation Domains etc.
- Galois Theory is missing entirely.

Mathlib

Generally, mathlib implements theories in the most abstract setting possible.
For example, polynomial is defined as

```
def polynomial (α : Type*) [comm_semiring α] := ℕ →₀ α
```

- This is motivated by the desire for code reuse.
- E.g. We know that any ordered finite set has a maximum. This property can be used wherever ordered finsets come up. like to argue that there is a coefficient of maximum degree.
- Alternative implementations like lists of coefficients would require a separate proof about the fact.
- Similarly, it provides a uniform API between components of the library.
- However, this does have the disadvantage that definitions can become non-intuitive.

Gauss's Lemma

Theorem

Let α be a Unique Factorisation Domain.
Let p be a polynomial with coefficients in α. Then p factors in $\alpha[X]$ if and only if it factors in Frac$(\alpha)[X]$.

Gauss's Lemma

Definition

Let p be a polynomial. We say p is primitive if the only constants in α which divide p are units.

Lemma (Gauss's Primitive Polynomial Lemma)

Let p, q be primitive polynomials. Then pq is a primitive polynomial.

Gauss's Primitive polynomial Lemma

Proof.

Assume for a contradiction that pq is not primitive. Then we have some $c \in \alpha$ such that c is not a unit and $c \mid p$.
α is a UFD, so c has some irreducible factor that divides pq. So WLOG, we may assume c is irreducible, and hence prime.
$\therefore \alpha/(c)$ is a domain.
$\therefore \alpha/(c)[x]$ is a domain.
$c \mid pq$ so pq vanishes in $\alpha/(c)[x]$.
As $\alpha/(c)[x]$ Integral Domain, p vanishes or q vanishes.
$\therefore c \mid p$ or $c \mid q$.
$\therefore$ Either p or q is not primitive.
Contradiction. $\square$

Gauss's Lemma ($\implies$)

Proof.

Let p be an irreducible in α. Any irreducible term must be primitive, so we know p is primitive.

Suppose for a contradiction that p is not irreducible in $Frac(\alpha)$.

Then $p = ab$, for some $a, b \in Frac(\alpha)[x]$.

We may multiply a and b by some $c_1, c_2\alpha$ so that $c_1 a, c_2 b$ have α coefficients. $c_1 c_2 p = c_1 a c_2 b$. We can factor $c_1 a$ to some $c_1' a'$ and $c_2 b$ to $c_2' b'$, so that a' and b' are primitive.

$\square$

Proof.

(contd.)

Hence $p = \frac{c_1' c_2'}{c_1 c_2} a' b'$.

If $\frac{c_1' c_2'}{c_1 c_2}$ in α, then we have produced a factorisation for p. A contradiction.

If $\frac{c_1' c_2'}{c_1 c_2}$ is not a unit in α, then the denominator in reduced form of $\frac{c_1' c_2'}{c_1 c_2}$ must divide each coefficient of $a' b'$, because $p = \frac{c_1' c_2'}{c_1 c_2} a' b'$ has α coefficients.

Hence $a' b'$ is not primitive. Contradiction.

$\square$

Proof in Lean

I took two broad approaches towards implementing the proof in Lean.

- Proof Above
- A proof using the notion of content.

The content is defined as the ideal generated by all the coefficients of p.
The content proof is broadly similar to the proof above, but by using properties of Ideals, it avoids the need for Unique Factorisation (Just the presence of a GCD).
However, Lean doesn't have a decent API for dealing with R-linear combinations, so I abandoned it.

Defining the Lemma

Theorem

Let p be a polynomial with coefficients in α. Then p factors in $\alpha[X]$ if and only if it factors in $Frac(\alpha)[X]$.

- p is a polynomial with coefficients in α, i.e. of type `polynomial` α.
- Factorisation in $Frac(\alpha)[X]$ only makes sense for polynomials in `polynomial (quotient_ring` α`)`.
- So we explicitly need to specify that the canonical embedding of p is irreducible.

```
lemma irred_in_base_imp_irred_in_quot {p : polynomial α} (hp_ir : irreducible p)
    (hp_nc : ¬is_const p) : irreducible (quot_poly p) :=
```

Rationals that are Integers

- Similarly, we cannot claim that an element of *Frac* α is an element of α.
- Lean only has total functions, so we cannot define any `to_base`: *Frac* $\alpha \to \alpha$ directly.
- We may use an option type, defining `to_base`: *Frac* $\alpha \to$ `option` α.
  An option type is a wrapper around alpha, so f(a) may be Just a or Nothing.

```
inductive option (α : Type u)
| none {} : option
| some    : α → option
```

- We may choose to define a coercion that depends on a hypothesis that somehow guarantees that the function is well defined.
- But dealing with options or carrying additional hypotheses around around would get annoying very quickly.
- So I chose to account for a rational r being an integer by stating $\exists (a : \alpha), \texttt{to\_quot}\ \ a = r$.
  This does have the disadvantage that I have additional variables to worry about.
  So, for example, so state that a scalar multiple of m is an @*alpha* polynomial, I wrote the following:
  ```
  ∃ (c : α) (d : polynomial α), quot_poly (C c) * m = quot_poly d
  ```

The Actual Proof

The source code is up on GitHub, at
`https://github.com/chocolatier/theoremproving/`.
Keeping the above points in mind, it is quite straightforward.