Implementing Gauss's Lemma in Lean

Aditya Agarwal

March 2, 2019

1 Background

I attempted to implement a proof of Gauss's Lemma in Lean over the Australian summer of 2018-2019. This report provides the background on Lean and the Gauss Lemma, and details my (yet incomplete) proof.

1.1 Lean

Lean is an Interactive theorem prover that implements mathematics using a form of dependent type theory known as the Calculus of Inductive Constructions.

1.1.1 Interactive Theorem Proving

Interactive Theorem Provers are tools for writing proofs in formal mathematics that seek to abstract away the low level busy work. They work by having humans guide them by suggesting which "tactic" to use, and showing what they have inferred. For example

For example, to prove that any reducible non-unit can be written as a product of two non-units, we may want to work with the definition of irreducibility.

```
lemma not_irred_imp_prod {p : γ} (hp : ¬irreducible p) (hp' : ¬is_unit p) :
    ∃(m n : γ),   p = m * n  ∧ (¬ is_unit m) ∧ (¬ is_unit n) :=
begin
    unfold irreducible at hp,
end
```

[1]

So we can tell the theorem prover to unfold the definition of irreducibility, at which point it will inform us on what the new state of the proof is.

```
Tactic State

γ : Type u,
_inst_1 : decidable_eq γ,
_inst_2 : integral_domain γ,
_inst_3 : unique_factorization_domain γ,
p : γ,
hp' : ¬is_unit p,
hp : ¬(¬is_unit p ∧ ∀ (a b : γ), p = a * b → is_unit a ∨ is_unit b)
⊢ ∃ (m n : γ), p = m * n ∧ ¬is_unit m ∧ ¬is_unit n
```

[1] Forgive the code screenshots. Getting Unicode to work in LaTeXturned out to be rather daunting.

As we can see here, it has updated `hp` be ¬(definition of irreducibility). So by specifying which tactic to use, we may guide the theorem prover towards the proof.

These tactics help the prover generate a proof, that is fed into a proof verifier, which then checks the correctness of the proof.

In Lean, we may also interface with the proof verifier directly, in what is known as "term mode".

For example, we show $0 + x = x$ by induction. Here s is the successor operation, and z is zero.

```
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)
```

1.1.2 Dependent Type Theory

The Calculus of Constructions that Lean implements is a form of Dependent Type Theory, an alternative axiomatisation of mathematics. Rather than being constructed out of sets, each object has a fixed type, that type belongs to some universe u.

For example,

- $n : \mathbb{N}$ n belongs to the type of natural numbers.
- $p \iff p : Prop$ - $p \iff p$ is a proposition, which is also a type.

This type may depend on a parameter such as

- list α
- polynomial α

We also have the notion of Π-types and Σ-types, which build new types out of other types.

Π types denote the notion of a dependent function type. A function whose return type may depend on the input parameter. E.g. the function $f(a) := x + a$ that takes an a in any ring α, returns $x + a$ in $\alpha[x]$

$$C : \Pi\ \alpha : \texttt{Type u}, \alpha \to \texttt{polynomial}\ \alpha$$

Similarly, Σ-types denote the cartesian product, where the type for the second element is allowed to depend on the first.

i.e. $\Sigma\ \alpha\ \beta$ denotes the type $\alpha \times \beta$, where β is allowed to depend on the type of α.

1.1.3 Mathlib

Mathlib is the mathematical components library for Lean.

It contains the beginnings of Algebra, Analysis, Topology, some elementary Number 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.

I initially wanted to do some Algebraic Number Theory, but found Galois Theory was missing entirely. So I chose to implement the Gauss Lemma, the first step towards implementing Galois Theory.[2]

Implementation Style in 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
```

Which is a function from $\mathbb{N}$ to the commutative semiring of coefficients α with a finite support (i.e. only finitely many non-zero values in the image).

This is largely motivated by the desire for code reuse. By defining properties in the most general setting possible, we can reuse them across many instances.

For example, We know that any ordered finite set has a maximum. This property can be used wherever ordered finite sets come up. Like to argue that a polynomial has a coefficient of maximum degree. The support of polynomial is finite and has a natural order. So ordered finsets having a maximum gives us the existence of a coefficient of maximum degree automatically.

While an 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 can make definitions non-intuitive, which can slow down the development process.

1.2 Gauss's Lemma

Before implementing it in Lean, we recall the statement and proof the Gauss Lemma.

Theorem 1 (Gauss's Lemma)**.** *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 $Frac(\alpha)[X]$.

Before we can prove Gauss's Lemma, we need to prove Gauss's Primitive Polynomial Lemma.

[2] My first plan was to work all the way to the Fundamental Theorem of Galois Theory, but that *grossly* overestimated how fast I could implement proofs in Lean.

Definition 1 (Primitive Polynomial)**.** *Let p be a polynomial in $\alpha[X]$. We say p is primitive if the only constants in α which divide p are units.*

Lemma 1 (Gauss's Primitive Polynomial Lemma)**.** *Let p and q be primitive polynomials. Then pq is a primitive polynomial.*

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 $\alpha/(c)[x]$ is an Integral Domain, p vanishes or q vanishes in $\alpha[X]$.

$\therefore c \mid p$ or $c \mid q$.

$\therefore$ Either p is not primitive or q is not primitive.

This is a contradiction. Hence pq must be primitive. □

Now we prove Gauss's Lemma.

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_1a, c_2b have α coefficients. $c_1c_2p = c_1ac_2b$. We can factor c_1a to some $c_1'a'$ and c_2b to $c_2'b'$, so that a' and b' are primitive.

Hence $p = \frac{c_1'c_2'}{c_1c_2}a'b'$.

Let $k = \frac{c_1'c_2'}{c_1c_2}$

We note that k must be in α, because otherwise if k is not in α, it's denominator (in reduced form) must divide each coefficient of $a'b'$. This is because $p = ka'b'$ has α coefficients.

Hence $a'b'$ is not primitive. But by the primitive polynomial lemma, we know $a'b'$ is primitive. This is a contradiction.

But now we have produced a non-trivial factorisation for p in $\alpha[X]$, namely $ka'b'$. This is a contradiction.

So it follows that if p is irreducible in $\alpha[X]$ is irreducible in $Frac(\alpha[X])$

□

2 Implementation Overview

My implementation is available at `https://github.com/chocolatier/theoremproving`.

Note: as of commit 745d1d79e473eb48788be8ba8b3cfb6774c424a7, not all the functions I describe are present/implemented. This is because I am in the middle of a code refactor, the reasons explained later.

The implementation of the forward direction of the Gauss Lemma itself is split into three separate lemmas.

- `can_factor_poly`: Which states that any reducible polynomial p in $Frac(\alpha)[X]$ can be written as $kd'd'_2$, where k is in $Frac(\alpha)$, and d', d'_2 are primitive polynomials in $\alpha[X]$.

```
lemma can_factor_poly (p : polynomial α) (hp: ¬is_const p) (h_nir : ¬irreducible (quot_poly p)) :
    ∃(d' d₂' : polynomial α), ∃(k : quotient_ring α), quot_poly p =
        quot_poly (d' * d₂') *  (C k) ∧ primitive d' ∧ primitive d₂' :=
```

  This largely a straightforward calculation.

- `prim_associate`: If p and q are primitive polynomials, r is in $Frac(\alpha)$, and $p = rq$, then r must be an integer.

```
lemma prim_associate {p q : polynomial α} {r : quotient_ring α}
    (h : quot_poly p = C r * quot_poly q) : ∃(k : α), to_quot k = r :=
```

  This was implemented as a proof by contradiction in my previous iteration, where I argued that r would have to be in the form a^{-1}, for some $a \in \alpha$.

  I had sorried this fact, but the human proof is that if $p = \frac{m}{n}q$, where n is not a unit and n, m are coprime, then $np = mq$. This is a contradiction, because the LHS and RHS have distinct coprime factors.

  And then I did a fairly longwinded deduction (full of sorries) showing that since $a \mid c_i$, for all coefficients c_i of q, because each $a^{-1}c_i$ corresponded to some coefficient c'_i of p, q was not primitive.

  I believe the argument I provided in the human proof above will be easier to implement in the refactor.

- irred_in_base_imp_irred_in_quot: The forward direction of the Gauss Lemma.

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

  It largely puts the above two lemmas together, and does some simple deduction to argue why a witness to reducibility constitutes an contradiction.

This depended on The Primitive Polynomial Lemma, which was implemented in two parts.

- A polynomial being primitive is defined as it having no constant divisors.

```
def primitive (p : polynomial α) : Prop :=
    ∀(q : polynomial α), non_unit_const q → ¬const_divisor p q
```

- `div_pq_imp_div_p_or_q`: If an irreducible r in α divides pq then it must divide p or it must divide q.

```
lemma div_pq_imp_div_p_or_q {p q : polynomial α} {r: α} (hdiv : C r ∣ (p * q))
    (hr : irreducible r) : C r ∣ p ∨  C r ∣ q :=
```

  This should be extremely straightforward, since $\alpha/(r)$ is an integral domain if r is prime is already present in mathlib. As is $R[X]$ being an integral domain if R is.

  But unfortunately, just stating that $f = 0$ in $\alpha/(r)[X]$ makes the type class resolutions time out. So the core of the proof remains unimplemented.

- `prod_of_prim_is_prim` : The product of two primitive polynomials is primitive.

```
lemma prod_of_prim_is_prim (p q : polynomial α) :
    (primitive p ∧ primitive q) → primitive (p * q) :=
```

  This is again the proof by contradiction mentioned in the previous section.

Each of the proofs above depend on numerous subsidiary lemmas that I wouldn't bother proving to another human, but Lean cannot infer on it's own. Such as texttt-tnot_irred_imp_prod: Any reducible element can be written as a product of other non-units. Each of these are also quite straightforward to implement.

While once everything is implemented, it is straightforward, getting there wasn't.

3 Challenges in Implementation