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.

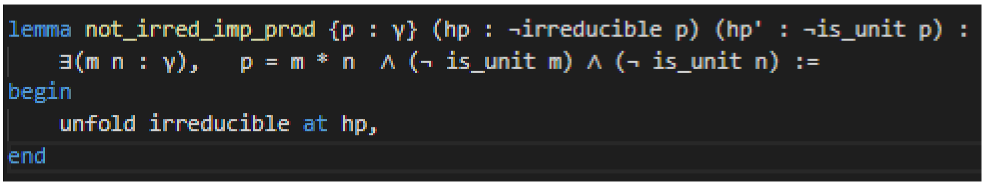
1.1 Lean

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

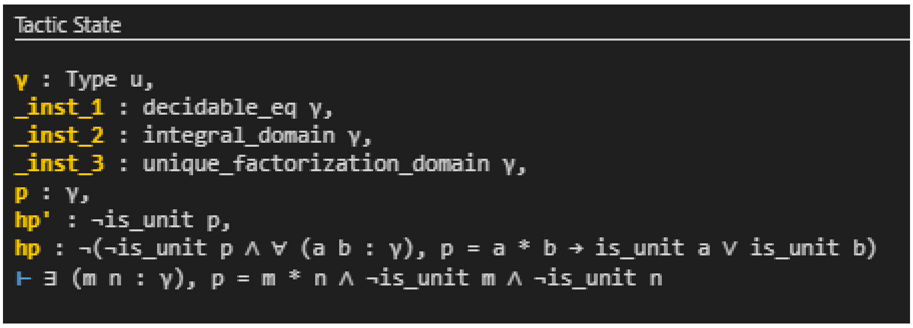
For example, to prove that any reducible non-unit can be written as a product of two non-units, we may want to work with the definition of irreducibility.



```
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
```

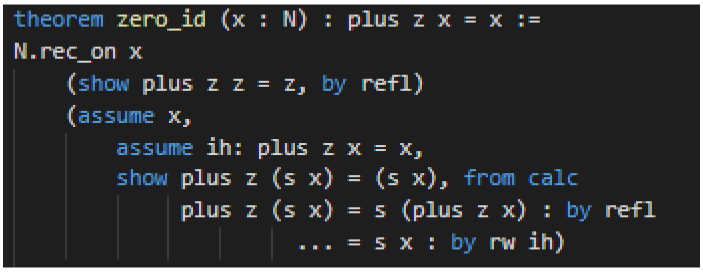
[1] Forgive the code screenshots. Getting Unicode to work in LaTeXturned out to be rather daunting.

As we can see here, it has updated `hp` be $\neg$(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.

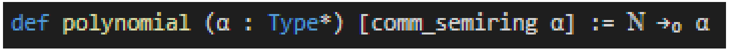
It contains the beginnings of Algebra, Analysis, Topology, some elementary Number Theory. On the Algebra side, it contains basic definitions about Groups, Rings, Modules etc. And properties about Integral Domains, Euclidean Domains, Unique Factorisation Domains etc.

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.

It is also worth noting that by default Lean works with constructive logic that lacks the law of the excluded middle, $p \lor \neg p$. However, this can be easily worked around if we specify that we are using classical logic.

Similarly, since constructability is default, Lean also does not allow for the Axiom of Choice, unless we explicitly state that our work is non-computable.

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.*

[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.

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.

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 $Frac(\alpha[X])$

□

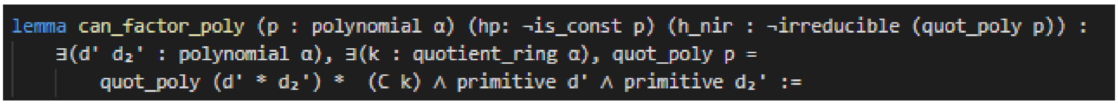
2 Implementation Overview

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.

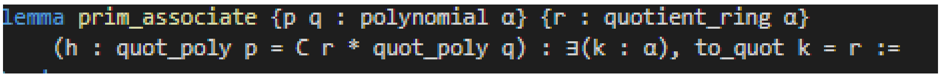
- `can_factor_poly`: Which states that any reducible polynomial p in $Frac(\alpha)[X]$ can be written as $kd'd_2'$, where k is in $Frac(\alpha)$, and d', d_2' are primitive polynomials in $\alpha[X]$.

  ```
  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 :=
  ```

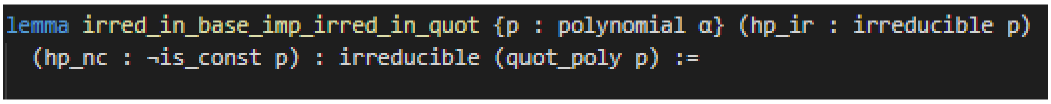
  This was implemented as a proof by contradiction in my previous iteration, where I argued that r would have to be in the form a^{-1}, for some $a \in \alpha$.

  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.

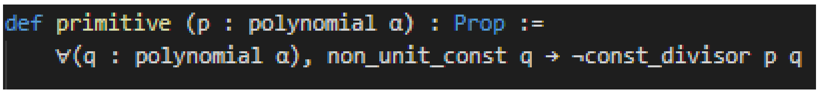
  

  ```
  lemma irred_in_base_imp_irred_in_quot {p : polynomial α} (hp_ir : irreducible p)
    (hp_nc : ¬is_const p) : irreducible (quot_poly p) :=
  ```

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

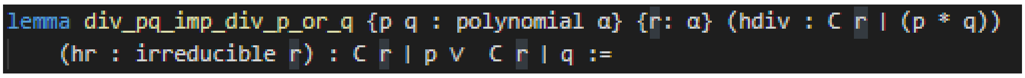
This depended on The Primitive Polynomial Lemma, which was implemented in two parts.

- 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.

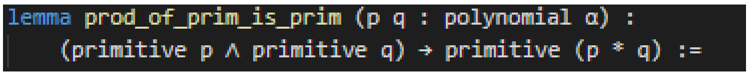
  

  ```
  lemma div_pq_imp_div_p_or_q {p q : polynomial α} {r: α} (hdiv : C r ∣ (p * q))
      (hr : irreducible r) : C r ∣ p ∨  C r ∣ q :=
  ```

  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 `not_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

3.1 Choosing which Proof to Implement

As mentioned in Section 1.1.3, mathlib typically contains definitions/implementations of theorems in the most general setting possible. The most general version of Gauss's Lemma holds in GCD domains, which may or may not have unique factorisation.

To prove this, instead of defining primitive polynomials as polynomials with no non-unit constant divisors, we define them as polynomials having trivial content.

The content of a polynomial p is defined as the ideal generated by the coefficients of p.

This proof is broadly similar to the one I detailed in Section 2, except that it relies on Ideal Arithmetic. [1].

I started working on this proof, but sidelined it in favour of the less general proof because Lean has no easy mechanism for dealing with R-linear combinations, so I would get bogged down by calculations to show extremely finicky details.

The little work that I did do can be found in the previously linked repo, in `src/gauss_lemma_content_edition.lean`.

3.2 Strict Type Checking

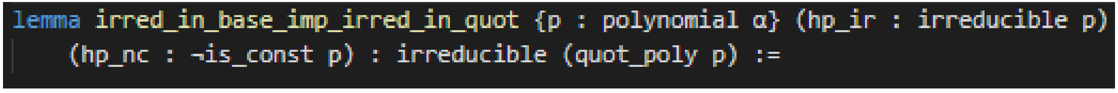
You may have noticed in the previous section that my textual description of the lemma and the actual Lean definition are not strictly identical.

For example, I described `can_factor_poly` as stating 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 $\alpha[X]$.

But what I have written is for any p in $\alpha[X]$, if it's canonical embedding into $Frac(\alpha)[X]$ is irreducible, then it's embedding into $Frac(\alpha)[X]$ can be written as the product of some rational k and the embedding of the product of two primitive α-polynomials d', d_2'.

A human mathematician will identify those two statements as being identical, but a computer does not. If p is a polynomial in $\alpha[X]$, it is not a polynomial in $Frac(\alpha)[X]$. So it does not make sense to talk about p factoring in $Frac(\alpha)[X]$.

Instead, we need to specify that we are talking about the image of p under the canonical embedding from $\alpha[X]$ to $Frac(\alpha)[X]$.



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

[1] The full proof is available on Wikipedia. `https://en.wikipedia.org/wiki/Gauss%27s_lemma_(polynomial)`

Explicitly specifying that the obvious embedding of p in $Frac(\alpha)[X]$ is irreducible isn't too bad. But the reverse direction isn't so nice.

We cannot claim that an element of $Frac\,\alpha$ is an element of α. 2 in $\mathbb{Q}$ is the equivalence class of pairs $[(2,1)]$, while 2 in $\mathbb{Z}$ is just 2.

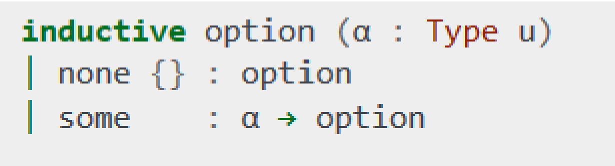
We can define a map $\psi : Frac\ \alpha \to \alpha$ that forgets the '1' and it being an equivalence class, but this is not possible in Lean.

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 Some a or none.



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

However, this would involve accounting for the possibility of `to_base a` being none at various points in the proof.

We may choose to define a coercion that depends on a hypothesis that somehow guarantees that the function would be 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

$\exists (a : \alpha),$ `to_quot` $a = r$.

i.e. there is an element a in the the base field α, such such that r is equivalent to $a's$ embedding in $Frac(\alpha)$

This does have the disadvantage that I have additional variables to worry about, and can become even more convoluted than in `can_factor_poly`. But on the balance I thought that it would be the least annoying option to deal with.

Unlike the other options, the statement itself does not provide a construction, and just asserts the existence of such an a. So such a construction would need to be supplied separately. However, this was not an issue for me, because I do not use the specific value of a at any point.

3.3 Proving "Trivialities"

Going into the project, I expected Lean to be able to infer a lot more that it actually can. For example, if $c \mid p$, then p is not primitive. Or that for any polynomial p, we can factor it as ap', where a is a constant, and p' is primitive. Facts I used without justification in my human proof.

Similarly, Lean isn't particularly good at rearranging equations or dealing with homomorphisms. Tactics like "simp" don't always do what you want it to do.

For example, I may have something the form $\phi(a)+\phi(b+c)$, a hypothesis $a+b+c=0$, and desire to show $\phi(a)+\phi(b+c)=0$, where ϕ is an homomorphism. Using the "simp" tactic here, even with the hypothesis, causes Lean to expand the equation out into $\phi(a) + \phi(b) + \phi(c)$[2]

"ring" can do a lot of equation rearrangement, but it times out on my equations, because I end up having a lot of variables due to reasons mentioned above. It also cannot deal with hypotheses or homomorphisms.

Similarly, "rewriting" doesn't account for ring axioms by default, so if we want to show $abc = b$, and have a hypothesis $h : ac = 1$, rewriting the target with h directly leads to a pattern match failure, and we need to rewrite the target using commutativity first.

This adds a lot of busy work into the proof.

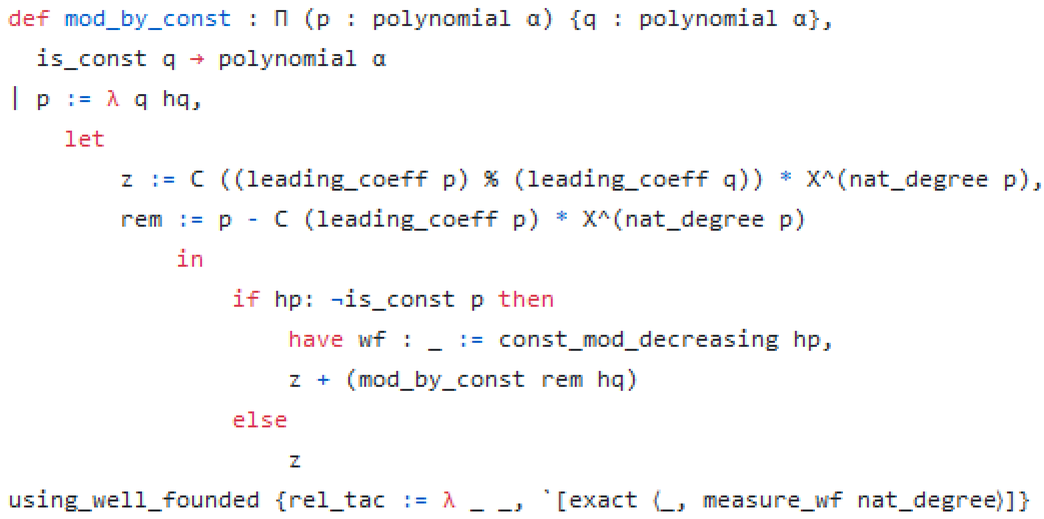
3.4 Choosing Definitions Carefully

Writing Lean Code involves a lot of refactoring. So it becomes important to choose your definitions carefully, because even a slight change in definitions can cause all your proofs using it to become invalid, causing you to repeat all the busywork mentioned above.

I learnt this the hard way, when I changed how I was defining a primitive polynomial.

My first definition of primitiveness was p is primitive, if for all non-unit constants c, $p \mod c \neq 0$.

The inbuilt polynomial mod function is only defined over fields[3], and for monic divisors, so I rolled my own mod function for constants.



```
def mod_by_const : Π (p : polynomial α) {q : polynomial α},
  is_const q → polynomial α
| p := λ q hq,
    let
        z := C ((leading_coeff p) % (leading_coeff q)) * X^(nat_degree p),
        rem := p - C (leading_coeff p) * X^(nat_degree p)
            in
                if hp: ¬is_const p then
                    have wf : _ := const_mod_decreasing hp,
                    z + (mod_by_const rem hq)
                else
                    z
using_well_founded {rel_tac := λ _ _, `[exact ⟨_, measure_wf nat_degree⟩]}
```

Where we just *mod* each coefficient of p by q individually.

Unfortunately, Lean could not figure out that $mod\_by\_constpq = 0$ equivalent to $q \mid p$, so all the divisibility lemmas in mathlib were hidden from me.

[2] This behaviour may not be replicated exactly. In my attempts to construct a minimal example, simp just fails to simplify. However, I have observed substantially similar behaviour in my actual proof.

[3] Lean calls them "discrete fields". Here "discrete" just means that the elements have decidable equality. The standard definition of field in Lean does not have decidable equality built in.

My options were

- Prove that $mod\_by\_const\ p\ q = 0 \iff q \mid p$, and manually invoke this equivalence every time I needed divisibility.
- Redefine primitiveness in terms of divisibility.

I chose the latter option, because the first one would make the code unnecessarily ugly. But this required that I rewrite any lemmas I had written with the old definition to work with the new one, which meant a lot of busy work.

So I should have been more careful about how I defined primitiveness, and looked at what was available in the library and take full advantage of it.

3.5 Sketching out the Proof First

Because Lean involves a lot of refactoring, you end up throwing a lot of little lemmas you have written out. So it is worthwhile just specifying and “sorry”-ing any lemma you might need, and worrying about them only after you've implemented the mathematically interesting parts.

Again, this is something I learnt the hard way. In my first attempt at the Gauss Lemma, where I defined primitiveness using `mod_by_const`, I started filling out any lemma that I could. So I proved properties like well foundedness of `mod_by_const` (i.e. the recursion in `mod_by_const` terminates) before I realised I couldn't access the divisibility properties with my current definition.

So when I rewrote my code, I ended up throwing away filled lemmas that I had spent time implementing. There is also a psychological factor - because I had put in so much effort with the old definition, I was reluctant to delete it. So I worked with it for longer than I should have.

This could have been avoided if I had just sorried the lemmas instead.

3.6 Code Bloat