Algebra Theorem Proving in Lean

Li Xiang

September 3, 2021

1 Introduction: the Lean ecosystem

To establish the foundation of the project, we begin by describing what Lean is and how it works. Lean is a programming language and project launched in 2013 by Leonardo de Moura of Microsoft Research. It has a community that communicates on Github, a messaging site called Zulip, and on Discord in the case of university students working on projects. Lean is supported by experts dedicated to the serialization of mathematics while helping newcomers as they come along to explore.

Functionally, Lean is used as a theorem prover. The Lean community has built up a mathematics library of definitions and theorems, which are built themselves out of functions, classes, structures, and types, which can represent a variety of different concepts, and once one thing is defined or proven it can be manipulated extensively. For example, one can declare a variable G as a Type, which is essentially the most generic of declarations, define groups and subgroups, then declare that G is a group and H as a subgroup of G. Types commonly hold elements, so that one can say `(g:G)`, meaning that g is an element of the group G. We can continue by defining what it means to be an abelian group, call the definition `is_abelian`, and write a theorem with a hypothesis `(h:is_abelian G)`, and use this fact as required.

Naturally, the mathematics library, called mathlib[4], already contains groups, subgroups, and what it means to be abelian, but it is useful to tinker yourself. Starting from scratch and working towards a theorem you want to prove requires building a pyramid with a wide base so that everything you need is available. It is useful to have a series of lemmas that do the heavy lifting for you later on. Mathlib contains numerous lemmas you can sift through to find what you need, or perhaps you may discover that mathlib does not contain the piece you need. In fact, there is a list of maths not yet contained in mathlib that can be understood by undergraduates, to serve as a guiding light to undergraduates wishing to contribute.

What is the best way to gain experience in programming? It is a possibility to work from scratch and use the library and mild guesswork on applying what is found. It is also beneficial to seek guidance, which can be difficult in relatively new and underutilized languages. Despite this, there exist several interactive tutorials that work to educate the user. One of the more accessible of these is called the Natural Number Game[7] which one can access through any browser. One learns about the primary tools, called tactics, which are analogous to what one may do when approaching a written proof. For example, one may choose to introduce a variable, apply a known equality, establish cases, prove by contradiction, and so on. However, it may be the responsibility of the programmer to learn more advanced techniques that are entirely necessary to continue, either through queries online or by finding examples within mathlib itself.

A marriage between classical theorem proving and a computer's understanding is required to progress. We can wave our hands all we like, but Lean will not see maths quite the same way as mathematicians do. Arguably, it would struggle to see maths at all. To reconcile the two, we choose to work with *type theory*.

2 Background: Type Theory

What is a proof? All mathematicians will have an internalised answer to this question, but not all will give you it to you straight, without some pause for thought. It is easy for a mathematician to look at some text and decide whether or not it constitutes a proof, but not so easy for them to tell a computer how to do the same. This is the challenge of computational pure mathematics. In the case of Lean, type theory provides the solution.

2.1 Types and Terms

Type theory is made of *terms* and *types*. Every term has a type that it belongs to. For example, the term `2` 'belongs to' or 'has' the type of natural numbers, `nat`. In type theory, and in Lean, we write this as '`2 : nat`'. Mathematicians should whisper '$2 \in \mathbb{N}$' very quietly to themselves when reading this, but types are far more general than sets. The following are some important examples of types.

- `Type` is the type to which all other types belong. For example, `nat : Type`.
- `Prop` is the type of propositions. For example, `1 + 1 = 2 : Prop`, `1 + 1 = 3 : Prop`, and `the_Riemann_hypothesis_is_true : Prop`.
- `A → B` is the type of maps from type `A` to type `B`. A term of the type `A → B` can be considered as a particular function from `A` to `B`, which is constructed by *lambda abstraction*[5, p.22] with the syntax `λa, f(a)` in type theory. For example, we could define `f` as a function that takes a natural number and adds one to it, as the following Lean codes show:

```
def f : nat → nat := λn, n+1
```

2.2 Truth and Proofs

We have seen the type *Prop*, but what does it mean for a term `P : Prop` to be true or false? Well, propositions are not only terms of type `Prop`, but they are also types themselves. If we can demonstrate that a proposition has at least one term belonging to it, we interpret it as true. In this sense, if `P : Prop` and `p : P`, it is natural to say that `p` is a proof of `P`.

Let's see an example. Suppose we have `Q P : Prop`, `q : Q`, and `f : Q → P`. This means we know `Q` to be true (proven by `q`), and we have a map from `Q` to `P`. How can we prove `P`? We simply define a term `p : P` as `f(q)`! In Lean this might look something like this:

```
lemma p : P := f(q)
```

We have told Lean how to make `p`, Lean has type-checked our definition, and it agrees that `f(q)` has type `P`. We are now free to use `p : P` in any future code.

At the most fundamental level, all we ever do in Lean is define a term of some type using what has come before. It is a deep and beautiful thing that this can serve as a foundation of mathematics.

2.3 Currying

This section serves to illustrate one of the many type-theoretic idiosyncrasies that a mathematician must get used to when learning to use Lean.

Let `A`, `B` and `C` be types, and consider the type `B → C` of maps from `B` to `C`. Since this is a type like any other, no one can deny that we have another type `A → (B → C)` of maps from `A` to `(B → C)`. Now a term `f : A → (B → C)` is a map sending a term of type `A` to another map, which in turn sends a term of type `B` to a term of type `C`. So we have a single output which depends on two inputs, and with a little thought it should be clear that we have create the direct analogue of $f : A \times B \to C$ in set theory [5, p.23]. Here are two examples:

```
def add : ℕ → ℕ → ℕ := λm, λn, n+m

lemma two_only_even_prime (n : ℕ) :
is_prime n → is_even n → n = 2 := sorry
```

("sorry" here means that the detail of proof is omitted.)

2.4 Tactics

When it comes to writing Lean code for lemmas and theorems with nontrivial mathematical content, we do not do everything by defining terms as we have seen so far unless it is simple and concise to do so. Instead we use tactics, which are higher-level commands that tell Lean how to prove a lemma or theorem in a manner much more familiar to anyone who is used to following 'normal' proofs.

Under the hood, of course, Lean is still purely defining terms and type checking those definitions.

Tactic proofs look something like this:

```
lemma p : P :=
begin
    //tactics go here
end
```

This will be how most of our code will be written in what follows.

3 Goals

Based on the functionality of Lean, our goals were mainly divided into three categories so that we can first familiarize ourselves with the language, then use Lean as a tool to prove theorems at undergraduates ‘level, and finally try to apply Lean in our courses.

Category A: we aimed to make basic definitions and prove lemmas from the first principles. This category includes division, primes, greatest common divisors, equivalence relations, and integers. This exploration process let us familiarise ourselves with the basic tactics and see how Lean works from the perspective of constructors. Since we all took Natural Number Game as our beginning tutorial of Lean, we naturally agreed that extending the definition of natural numbers to integers would be one of our goals for category A. This process of defining integers will be clearly explained later as an example of programming.

Category B: we aimed to prove theorems using basic definitions from mathlib. We proved three theorems in this category: Chinese Remainder Theorem, Euler's Theorem and Lagrange's Theorem. Although most of these theorems already exist in mathlib in a condensed way, proving these again is still a wise choice, and it is highly recommended for future beginning users of Lean. Proving theorems using basic definitions is a good way to practice how to use tactics and look through what's in and what's still not yet contained in mathlib.

Category C: we aimed to prove theorems that could be applied in our courses for tutoring. Lean as a tool has the advantage to teach students proofs in an interactive way, so we hoped to apply Lean in undergraduate teaching. After proving theorems in Category B, we had a more solid base of knowledge of group theory, we decided to prove theorems related to the principal ideal domain which could let us integrate all of our knowledge in group theory. Due to the limit of time, we didn't manage to prove the whole theorem, but the structure of the proof is built and the possible challenges that might appear when realizing the proof in Lean have been discussed. More details about our progress on this proof will be discussed in section 7.

4 Category A: Defining Integers

In this section, we will give an outline of defining integers on Lean as an example of programming. Since it's only for displaying how a Lean project works, we won't go through details of everything. Instead, we will show what main steps and challenges it might include, especially focusing on the definition stuffs. See the complete codes in Appendix A.

4.1 Steps

We should outline the main steps at the very beginning. Recall that $\mathbb{Z}$ is defined as the quotient set $(\mathbb{N}\times\mathbb{N})/\sim$ under the equivalence relation $\sim$ which is defined as $(a,b)\sim(c,d)\leftrightarrow a+d=b+c$. Given the definition, the programming can be split into the following steps:

1. Give a general implementation of an equivalence relation;
2. Apply the equivalence relation $\sim$ to give a definition of the integers;
3. Add the group structure to integers.

For step 1, the relation on a set X is defined as the subset of the Cartesian product $X\times X$ with the type `set (X × X)`, which is defined as `(X × X) → Prop` on Lean and equivalent to `X → X → Prop` by currying. The codes for the first step are shown below.

```
def is_equiv_type {X:Type}(S:X→ X→ Prop):=
  (∀ a:X, S a a) ∧ (∀ a b:X, S a b → S b a) ∧
  (∀ a b c:X, S a b → S b c → S a c)
```

For step 2, we first define the relation $\sim$ and prove that it is an equivalence relation on Lean.

```
def diff : ℕ × ℕ → ℕ × ℕ → Prop :=
  λ a, λ b, a.fst + b.snd = a.snd + b.fst
lemma diff_is_equiv : is_equiv_type diff := sorry
```

Then we need the notion of the quotient and use it to define the integers. The quotient set is defined as a set that satisfies a certain property on Lean.

```
def equiv_class_type {X:Type}(S:X→ X→ Prop)(a:X):= {x:X | S x a
    }
def quotient_type {X:Type}(S:X→ X→ Prop):=
  {e:set X | ∃ a:X, e = equiv_class_type S a}
def ourInt := quotient_type diff
```

For step 3, we add the group structure into the integers. To begin with, we define the zero element, the addition and the inverse under the addition. The outline of the code is shown below.

```
def zero: ourInt := sorry
def add_ourInt : ourInt → ourInt → ourInt := sorry
def inv_ourInt: ourInt → ourInt := sorry
```

Next, we verify that these definitions satisfy the axioms that make integers be a group. Here shows the statement of those axioms. To make it simpler, we rephrase `add_ourInt a b` to be `a add b` (This can be done by `infix` on Lean).

```
lemma add_zero_ourInt(a:ourInt) : a add zero = a := sorry
lemma add_assoc_ourInt(a b c:ourInt) :
  (a add b) add c = a add (b add c) := sorry
lemma inv_add_ourInt(a:ourInt):(inv_ourInt a) add a=zero := sorry
```

Finally, the group structure is derived directly from the definitions and lemmas we had before. The syntax is shown in the following codes.

```
def ourInt_group: group(ourInt) := {
  one := zero,
  mul := add_ourInt,
  (...ommited...)
  mul_left_inv := inv_add_ourInt
}
```

4.2 Challenges

One of the main challenges introduces us to coercions, which served to convert types. Suppose `G` is a group and `H` is a subgroup of `G`. While `G` is a Type in Lean, H is a term of type `subgroup G`. While you can say that g is in G through `(g : G)`, you can't do this for $h \in H$. Instead, `H` is coerced to a Type using an upward arrow with a bar on its end, called `coe_sort`. This then holds two pieces of information: h is in the group G, written ↥`h:G`, and a proof that h is also in the subgroup H. This extra notation and brouhaha is necessary because Lean treats universes, types, and terms as different levels of a system, and you use an arrow to essentially take an elevator to the more general level. I am far from well-versed on how to deal with a coercion when it pops up, but the options I've learned are to divide it into cases, which splits the two pieces of information I mentioned before, or to use theorems in mathlib to manipulate the arrows as desired.

Sometimes we also need an explicit change of type. Recall that the canonical map $X \to X/\sim$ sends a to $[a]$. The equivalence class $[a]$ with the type `set X` is in the quotient set $X/\sim$, but it is expected to have the type `↑(quotient_type diff)`. Thus, it is necessary to convert the former type to the latter one. The solution for that is the angular bracket. Generally speaking, for `p:P`, if we have a proof `h:p∈Q`, then `⟨p,h⟩` has the type `↑Q`. In this case, the lemma `equiv_class_type_in_quotient_type` below shows that $[a] \in X/\sim$, whose proof can be provided in the second argument inside the angular bracket to convert the type of $[a]$ for defining the canonical map.

```
lemma equiv_class_type_in_quotient_type{X:Type}(S:X→ X→ Prop)(a
    :X):
  equiv_class_type S a ∈ quotient_type S:= begin
    use a,
  end
def can{X:Type}(S:X→ X→ Prop):X→ quotient_type S:=
  λa,⟨equiv_class_type S a, equiv_class_type_in_quotient_type S a⟩
```

Once the canonical map is defined, we can define the integer zero in step 3.

```
def zero : ourInt := can diff (0,0)
```

Another challenge lies in taking a representative of an equivalence class. It relies on the notion of the section, which is a map $s : X/\sim \to X$ that satisfies $c \circ s = \mathrm{id}_{X/\sim}$ where c refers to the canonical map. Thus, a section is actually a choice of the representative.

We want to pick out a specific section to get a representative. But how do we know that there exists a section? Actually, the existence is guaranteed by how we defined `quotient_type` before. To choose a particular section on Lean, we first show that the element in the quotient, which is an equivalence class, is a nonempty set by the definition of `quotient_type`. Next, a built-in function `set.nonempty.some`[3] is introduced to choose an element of that nonempty set, which is precisely an representative of that equivalence class. This choice forms a construction of a particular section.

```
lemma equiv_class_nonempty{X:Type}{S:X→ X→ Prop}
  (h: is_equiv_type S)(e:quotient_type S):e.val.nonempty := sorry
def particular_sec{X:Type}{S:X→ X→ Prop}(h: is_equiv_type S):
  quotient_type S → X :=
  λe, set.nonempty.some (equiv_class_nonempty h e)
```

If we have a map $f : X \to Y$, then the corresponding map $\tilde{f} : X/\sim \to Y$ is induced by a particular section $s : X/\sim \to X$ such that $\tilde{f} = f \circ s$.

```
def induced_op{X:Type}{Y:Type}{S:X→ X→ Prop}(h: is_equiv_type S
    )
  (op:X → Y) : quotient_type S → Y := λ e, op (particular_sec e
    )
```

This provides a method of defining the addition and the inverse under the addition in step 3. Take defining the inverse as an example. Given a map $f : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}, (a, b) \mapsto [(b, a)]$, the inverse is induced by f using `induced_op`.

```
def pre_inv: ℕ×ℕ → ourInt := λ a, can diff (a.snd,a.fst),
def inv_ourInt: ourInt→ ourInt:=induced_op diff_is_equiv pre_inv
```

4.3 Remarks

From this little project of defining integers, we get familiar with the main characteristic of Lean. That is, Lean is a proof assistant based on type theory, which makes the type the first-class citizen in Lean programming. In this category, the currying as a technique of type theory gives a nice way to state the definition of the equivalence relation. But the omnipresence of the type also leads to the difficulties of converting all kinds of different types, which becomes the biggest challenge most of the time.

5 Category B: Chinese Remainder Theorem (Ring Version)

In this section, we start a real project of proving the ring version of Chinese Remainder Theorem (CRT). See the complete code in Appendix B.

5.1 Mathematical Part

The ring version of Chinese Remainder Theorem is stated below.

Theorem 5.1. *[2, p.291] If $I_1, \cdots, I_n$ are ideals of the commutative ring R such that $I_i + I_j = R$ for any i, j $(i \neq j, 1 \leq i, j \leq n)$, then*

$$R/\bigcap_{i=1}^{n} I_i = \bigoplus_{i=1}^{n} R/I_i$$

The proof can be seen in any textbook of algebra. Here shows the steps.

1. Define a map $f : R \to \bigoplus_{i=1}^{n} R/I_i, r \mapsto (r + I_1, \cdots, r + I_n)$;
2. Verify that f is a ring homomorphism;
3. Show $\ker f = \bigcap_{i=1}^{n} I_i$;
4. Show that f is surjective.

Then the result is derived by the first isomorphism theorem. The only difficulty lies in the fourth step. To show the surjection, we need a lemma.

Lemma 5.2. *[2, p.292] Given the condition of Theorem 5.1, for any i $(1 \leq i \leq n)$, there exists $r \in R$ such that $r - 1 \in I_i$ and $r \in \bigcap_{j \neq i} I_j$.*

This is equivalent to say that $I_i + \bigcap_{j \neq i} I_j = R$.

Proof. Fix i. For any $j \neq i$, $I_i + I_j = R$, there exists $a_j \in I_i$ and $b_j \in I_j$ such that $a_j + b_j = 1$ since $I_i + I_j = R$. Let $r = \prod_{j \neq i} b_j$. Then $r \in \bigcap_{j \neq i} I_j$ since each I_j is an ideal. Furthermore, $r - 1 = (\prod_{j \neq i}(1 - a_j)) - 1 \in I_i$ since the constant term vanishes $(1 + (-1) = 0)$ and each non-vanishing term has a factor $a_j \in I_i$ for some $j \neq i$. $\square$

Now we can prove that f is surjective. For any $b = (b_1 + I_1, \cdots, b_n + I_n) \in \bigoplus_{i=1}^n R/I_i$ and for any index i, there exists $r_i \in R$ such that $r_i - 1 \in I_i$ and $r_i \in \bigcap_{j \neq i} I_j$ by Lemma 5.2. Let $r = \sum_{i=1}^n b_i r_i$. To show that $f(r) = b$, we only need to show that $r + I_j = b_j + I_j$, or $r - b_j \in I_j$ for any index j. Indeed, $r - b_j = b_j r_j + (\sum_{i \neq j} b_i r_i) - b_j = b_j(r_j - 1) + (\sum_{i \neq j} b_i r_i) \in I_j$ since each term belongs to I_j.

5.2 Outline of Implementation

5.2.1 Statement of the Theorem

First we declare a commutative ring R as a global variable which can be applied to the whole file.

```
universes ur
variables {R : Type ur}
variables [comm_ring R]
```

Next, we have to think about how to list n ideals $I_1, \cdots, I_n$. It turns out that we can use a function $\{1, \cdots, n\} \to \{\text{ideals of } R\}$. There are many possible choices for expressing $\{1, \cdots, n\}$, such as `finset.range n`, `fin(n)`, or more generally, `fintype` α. In this section, we decide to take the first one, since it has much support of set operations in mathlib.

Here shows the code of the statement of the theorem, which is modelled on `ideal.quotient_inf_ring_equiv_pi_quotient` in mathlib[6].

```
theorem CRT_ring_version_general{n:ℕ}(I:finset.range n → ideal R
    )
  (h:∀ (i j:finset.range n),i≠ j→ (I i)⊔(I j) = ⊤):
  ⊓( i, I i).quotient ≃+* (Π i, (I i).quotient)
```

For the completeness, the statement of Lemma 5.2 is shown below, which is modelled on `ideal.exists_sub_one_mem_and_mem` in mathlib[**sref:ideal˙inf**].

```
lemma sum_intersect_lemma{n:ℕ}(I:finset.range n → ideal R)
  (h:∀ (i j:finset.range n),i≠ j→ (I i)⊔(I j) = ⊤)
  (i:finset.range n) : ∃ r:R, r-1 ∈ I i ∧ (∀ j≠ i,r∈ I j)
```

5.2.2 Proof of the Theorem

We follow the four steps mentioned in the mathematical part.

```
let f:R→ (Π(i:finset.range n), (I i).quotient):= sorry,
let hom: R →+* (Π i, (I i).quotient):= sorry,
have ker_of_hom: hom.ker = ⊓( i, I i):= sorry,
have hom_surjective: function.surjective hom:= sorry,
```

Then the goal is closed by the first isomorphism theorem.

```
rw ← ker_of_hom,
exact ring_hom.quotient_ker_equiv_of_surjective hom_surjective,
```

How to define f and hom (step 1,2) will be discussed later. Here shows the sketch (pseudocode) of step 3,4.

Algorithm 1 step 3: hom.ker = (⊓ i, I i)

```
1: have: x∈hom.ker → x∈(⊓ i, I i) :=
2:     x∈hom.ker → f x = 0 → ∀ i, x∈I i → x∈(⊓ i, I i)
3: have: x∈(⊓ i, I i) → x∈hom.ker :=
4:     x∈(⊓ i, I i) → ∀ i, x∈I i → f x = 0 → x∈hom.ker
```

Algorithm 2 step 4: function.surjective hom

```
1: for any b:(Π i, (I i).quotient)
2: let b_pick: finset.range n → R such that
3:     ∀ i, (b_pick i)+$I_i$ is the $i$-th entry of b
4: let r_pick: finset.range n → R such that
5:     ∀ i, r i is given by Lemma 5.2
6: let r:= ∑ i , (b_pick i) * (r_pick i)
7: have: f(r)=b
8:     for any index j
9:     have: r - b_pick j =
10:      (b_pick j)*(r_pick j - 1) + (∑i≠j,(b_pick i)*(r_pick i))
11:    have: (b_pick j) * (r_pick j - 1) ∈ I j
12:    have: ∑ i ≠ j, (b_pick i) * (r_pick i) ∈ I j
13:    have: r - b_pick j ∈ I j
14:    have: r+$I_j$ = b_pick j+$I_j$
15:    have: f(r)'s $j$-th entry = b's $j$-th entry
```

5.3 Challenges

5.3.1 Operate a List with Arbitrarily Many Entries

To define the map f in step 1, we have to manipulate the list $(r+I_1,\cdots,r+I_n)$ with arbitrarily many components. The key is the correspondence between the list and a function. Specifically speaking, the list is in $\bigoplus_{i=1}^{n} R/I_i$, which can be considered as a function which maps the index $i \in \{1,\cdots,n\}$ to an element in R/I_i. From this point of view, we can define f on Lean as the following.

```
let f := λr,(λ i, ideal.quotient.mk (I i) r),
```

5.3.2 Define And Verify a Ring Homomorphism

The ring homomorphism is a structure on Lean, which consists of five members: the map itself and the proofs of four propositions that make the map become a homomorphism (mapping 0 to 0, mapping 1 to 1, preserving the addition and preserving the multiplication). Thus, define and verify a homomorphism on Lean is essentially providing these five members into the structure. The code for defining and verifying the homomorphism in step 2 is outlined below.

```
let hom: R →+* (Π i, (I i).quotient):= begin
  fconstructor,
  exact f, -- for the first member: f
  sorry, -- other four members, i.e., proofs of four propositions
end,
```

5.3.3 Convert Types

In the proof of Lemma 5.2, we consider the complement set $\{1, \cdots, n\} \setminus \{i\}$ to deal with $i \neq j$. A related built-in function is `finset.erase`, which is useful when being expressed as an indexed set in a sum or a product. However, `(finset.range n).erase i` is not valid on Lean due to the type error. On Lean, `A.erase` α requires that `A` has the type `finset` α. In this case, `i` has the type `finset.range n`, but `finset.range n` itself is not of the type `finset (finset.range n)`. A solution is `finset.univ`, which is shown below.

```
let comple:=(finset.univ:finset (finset.range n)).erase i,
```

5.3.4 Pick an Element in an Existential Proposition

In step 4, to show that f is surjective, we have to consider $b = (b_1 + I_1, \cdots, b_n + I_n) \in \bigoplus_{i=1}^n R/I_i$. How do we extract b_i from the i-th entry of b, i.e., $b_i + I_i$? First, we have a proposition below by the definition of the quotient.

```
have has_preimage:
  ∀i,∃r, ideal.quotient.mk (I i) r = b i := sorry,
```

Next, the problem becomes how to pick an element in the existential proposition `has_preimage`. It can be reduced to a problem that we have solved in Category A before. Generally speaking, the existential proposition defines a nonempty set that satisfies the property. Then `set.nonempty.some`[3] comes in to choose an element in the nonempty set.

```
lemma some_nonempty{P:α→ Prop}(hw: ∃ r:α, P r):
  {j:α | P j}.nonempty:= sorry
def construct_some{P:α→ Prop}(hw: ∃ r:α, P r):α:=
  set.nonempty.some (some_nonempty hw)
```

5.3.5 Prove a Property in Terms of the Sum

In step 4, we have to show that $\sum_{i \neq j} b_i r_i \in I_j$. This can be proved by the fact that $r_i \in I_j$ for each $i \neq j$. How can this argument be valid on Lean?

Mathlib provides a principle `finset.sum_induction`[1] which can prove the property in terms of the sum. In general, it can be shown that $\sum_{i \in S} g_i$ for some finitely indexed set S satisfies a property P if we manage to show that the following conditions hold:

1. 0 satisfies P;
2. g_i satisfies P for each $i \in S$;
3. a, b satisfying P implies $a + b$ satisfying P.

Thus, we can outline the code as the following way.

```
have p0 : p 0 := sorry,
have p_any: ∀ i∈S,  p (g i) := sorry,
have p_add: ∀ a b , p a → p b → p (a+b) := sorry,
exact finset.sum_induction g p p_add p0 p_any,
```

6 Category C: Application

To what extent will we be able to use Lean in an undergraduate environment? The research to find new paths forward in some of the subprojects we have been working on would be too much work to teach and provide benefit in a normal curriculum, and it would be most beneficial to avoid complications. It is perhaps viable to demonstrate how to define a particular class, such as a group, and to use this example to define another thing. Any type of exercise like this would require an already-written foundation filled with `sorries` where students should provide their input. It is a possibility that the best approach would be to model the natural number game. Teach students the tactics and the methods, apply them, and perhaps have students provide a written explanation of what the proof does. A written follow-up is highly beneficial because of the considerable discrepancy between proving in Lean and in a classic proofs course setting, and this may help bridge the gap and give Lean a valuable teaching asset.

It is of course the role of the educators to learn how to program in Lean and to design the necessary program to give to students. Some of the individual projects featured in our collection of reports have technical work to satisfy Lean and would be largely useless as tools to teach proofs. It would be wise to consult Lean aficionados, such as the creator of the Lean-specific course already taught at Imperial College London. Definitions and the use of tactics may be easily enough picked up to be beneficial on a short-term basis assuming Lean is not integrated into the maths degree as a whole, since this may be a little risky considering that it is still a little too niche to put all eggs in the Lean basket.

7 Future Direction: PIDs

In the latter stages of the project, we were introduced to principal ideal domains (PIDs) and the structure theorem for finitely generated modules over a principal ideal domain. This has not been covered in the library so far. So the implementation of the PID stuffs on Lean is our future direction.

The structure theorem is stated below.

Theorem 7.1. *[2, p.354] Let M be a finitely generated R-module where R is a PID. Then there exists a chain of ideals $I_1 \supseteq I_2 \supseteq \cdots \supseteq I_m$ for some $m, n (m \leq n)$ such that*

$$M \simeq R/I_1 \oplus R/I_2 \oplus \cdots \oplus R/I_m \oplus R^{n-m}.$$

Moreover, these ideals $I_1, \cdots, I_m$ are unique.

The proof of the theorem is split into the four following pieces, from Lemma 7.2 to 7.5.

Lemma 7.2. *Let R be a PID and F be a free R-module of rank n. If S is an R-submodule of F, then S is free of rank at most n.*

Lemma 7.3. *Let R be a commutative ring with unit 1. Let M be an R-module. Then the following statements are equivalent.*

1. *Every non-empty family of R-submodules of M contains a maximal element;*
2. *Every R-submodule of M is finitely generated;*
3. *Every increasing sequence $M_1 \subseteq M_2 \subseteq \cdots$ of R-submodules of M stablize.*

Lemma 7.4. *For a free R-module F of rank n and an R-submodule S of F, there is a basis $\{e_1, \ldots, e_n\}$ of F and a chain of ideals $I_1 \supseteq I_2 \supseteq \cdots \supseteq I_m$ of R for some $m \leq n$ such that*

$$S \simeq I_1 e_1 \oplus I_2 e_2 \oplus \cdots \oplus I_m e_m.$$

These ideals are uniquely determined by F and S.

Lemma 7.5. *The uniqueness part of Theorem 7.1.*

The main steps of implementing PID concepts on Lean is outlined below:

- derive necessary results in terms of PID (such as PID is UFD);
- define the basic notions of free module, torsion module, etc.;
- state the four lemmas above properly;
- prove the lemmas in order.

The potential challenges might be dealing with :

- the quotient of a module;
- module homomorphism;
- direct sum of arbitrarily many modules;
- multilinear maps and the determinant that occurs in the proof of Lemma 7.4.

References

[1] *algebra.big_operators.basic - mathlib docs*. mathlib for Lean - API documentation. URL: https://leanprover-community.github.io/mathlib_docs/algebra/big_operators/basic.html#finset.sum_induction (visited on 08/30/2021).

[2] Paolo Aluffi. *Algebra : chapter 0*. American Mathematical Society, 2009, p. 354.

[3] *data.set.basic - mathlib docs*. mathlib for Lean - API documentation. URL: https://leanprover-community.github.io/mathlib_docs/data/set/basic.html#set.nonempty.some (visited on 08/29/2021).

[4] *index - mathlib docs*. mathlib for Lean - API documentation. URL: https://leanprover-community.github.io/mathlib_docs/.

[5] N.J Institute For Advanced Study (Princeton and Univalent Foundations Program. *Homotopy type theory : univalent foundations of mathematics*. Univalent Foundations Program, 2013, p. 23.

[6] *ring_theory.ideal.operations - mathlib docs*. mathlib for Lean - API documentation. URL: https://leanprover-community.github.io/mathlib_docs/ring_theory/ideal/operations.html (visited on 08/30/2021).

[7] *The Natural Number Game*. www.ma.imperial.ac.uk. URL: https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/ (visited on 08/30/2021).

Appendix A Category A Codes

```
import CatA_equiv_relation
noncomputable theory
/- The file CatA_equiv_relation.lean should be included.
See https://github.com/ourlean/Equivalence-Relation -/

class ourgroup (G : Type) :=
(mul: G → G → G)
(one: G)
```

```
(inv: G → G)
(mul_assoc : ∀ (a b c : G), mul (mul a b) c = mul a (mul b c))
(mul_one : ∀ (a : G), mul a one = a)
(mul_right_inv : ∀ (a : G), mul a (inv a) = one)


def diff : ℕ × ℕ → ℕ × ℕ → Prop :=
λ a, λ b, a.fst + b.snd = a.snd + b.fst

lemma diff_is_refl : is_refl_type diff :=
begin
  unfold is_refl_type, /-just for our sanity-/
  intro a,
  unfold diff,
  rw add_comm,
end

lemma diff_is_symm : is_symm_type diff :=
begin
  unfold is_symm_type, /-sanity-/
  intros a b,
  intro h,
  unfold diff at *,
  symmetry,
  rw add_comm,
  rw add_comm b.fst,
  exact h,
end

lemma add_two_eq (a b c d : ℕ) (h : a = b) (g : c = d) : a + c =
    b + d := by  {exact congr (congr_arg has_add.add h) g}

lemma diff_is_trans : is_trans_type diff :=
begin
  unfold is_trans_type,
  intros a b c,
  intro h1,
  intro h2,
  unfold diff at *,
  /-have h1h2 : (a.fst + b.snd) + (b.fst + c.snd) = (a.snd + b.
    fst) + (b.snd + c.fst) := by add_two_eq (a.1 + b.2) (a.2 + b
    .1) (b.1 + c.2) (b.2 + c.1) h1 h2,-/
  have h3 : a.1 + b.2 + c.2 = a.2 + b.1 + c.2,
  rw h1,
  rw add_assoc a.2 at h3,
  rw h2 at h3,
  rw add_comm at h3,
  rw add_comm b.2 c.1 at h3,
  rw ← add_assoc at h3,
  rw ← add_assoc at h3,

  have g := add_right_cancel h3,
  rw add_comm,
  exact g,
end

lemma diff_is_equiv : is_equiv_type diff :=
begin
  unfold is_equiv_type,
  split,
  exact diff_is_refl,
  split,
  exact diff_is_symm,
  exact diff_is_trans,
end

def ourInt := quotient_type diff

/- zero -/

def zero:ourInt:=can diff (0,0)

lemma zero_reps:(particular_sec diff_is_equiv zero).fst
=(particular_sec diff_is_equiv zero).snd:=
begin
  have t:=par_sec_can diff_is_equiv (0,0),
  unfold diff at t,
  unfold zero,
  simp at *,
  exact t,
end

/- inverse -/

def pre_inv: ℕ×ℕ→ ourInt:=λ a, can diff (a.snd,a.fst)

def inv_ourInt: ourInt→ ourInt:=induced_op diff_is_equiv pre_inv

lemma inv_reps(a:ℕ× ℕ)(b:ℕ× ℕ):
(a.fst+b.fst=a.snd+b.snd) ↔ can diff b=pre_inv(a):=
begin
```

```
  unfold pre_inv,
  rw ← equiv_iff_same_image_under_can diff_is_equiv,
  unfold diff,
  simp,
  omega,
end

lemma inv_is_well_defined:
op_is_well_defined diff_is_equiv pre_inv:=
begin
  intros a b f,
  unfold pre_inv,
  rw ← equiv_iff_same_image_under_can diff_is_equiv,
  unfold diff at *,
  simp,
  ←rw f,
end

/- addition -/

def pre_add: ℕ×ℕ → ℕ×ℕ→ ourInt:=
λ a, λ b, can diff (a.fst+b.fst,a.snd+b.snd)

def add_ourInt: ourInt→ ourInt→ ourInt:=
induced_op2 diff_is_equiv pre_add

infix ‘ add ‘:55 := add_ourInt

lemma add_is_well_defined:
op2_is_well_defined diff_is_equiv pre_add:=
begin
  intros a b c d f g,
  unfold pre_add,
  rw ← equiv_iff_same_image_under_can diff_is_equiv,
  unfold diff at *,
  simp,
  rw add_assoc,
  rw add_comm b.fst,
  rw ← add_assoc _ (c.snd+d.snd) _,
  rw ← add_assoc,
  rw add_assoc (a.fst+c.snd),
  rw add_comm d.snd,
  rw f,
  rw g,
  ring,
end
```

```
/- integers as an abelian group under addition -/

lemma add_comm_ourInt(a:ourInt)(b:ourInt):a add b = b add a:=
begin
  unfold add_ourInt,
  unfold induced_op2,
  unfold induced_op2_by_sec,
  unfold pre_add,
  rw add_comm,
  rw add_comm (particular_sec diff_is_equiv a).snd,
end

lemma add_assoc_ourInt(a:ourInt)(b:ourInt)(c:ourInt):
(a add b) add c=a add (b add c):=
begin
  let x:= a add b,
  let y:= b add c,
  have x_def:a add b=x:=rfl,
  have y_def:b add c=y:=rfl,
  let p:=particular_sec diff_is_equiv a,
  let q:=particular_sec diff_is_equiv b,
  let r:=particular_sec diff_is_equiv c,
  have a_h:can diff p=a:=by apply can_par_sec_expand
    diff_is_equiv a,
  have b_h:can diff q=b:=by apply can_par_sec_expand
    diff_is_equiv b,
  have c_h:can diff r=c:=by apply can_par_sec_expand
    diff_is_equiv c,
  have x_h:can diff (p.fst+q.fst,p.snd+q.snd)=x:=begin
    have temp:can diff (p.fst+q.fst,p.snd+q.snd)=pre_add p q:=by
    unfold pre_add,
    rw temp,
    rw ← x_def,
    rw add_ourInt,
    rw induced_op2,
    rw induced_op2_by_sec,
  end,
  have y_h:can diff (q.fst+r.fst,q.snd+r.snd)=y:=begin
    have temp:can diff (q.fst+r.fst,q.snd+r.snd)=pre_add q r:=by
    unfold pre_add,
    rw temp,
    ←rw y_def,
    rw add_ourInt,
    rw induced_op2,
    rw induced_op2_by_sec,
  end,
  rw x_def,
  rw y_def,
  have pre_assoc: pre_add (p.fst+q.fst,p.snd+q.snd) r
  =pre_add p (q.fst+r.fst, q.snd+r.snd):=begin
    unfold pre_add,
    rw ← equiv_iff_same_image_under_can diff_is_equiv,
    unfold diff,
    simp,
    ring,
  end,
  let k:=pre_add (p.fst+q.fst,p.snd+q.snd) r,
  have k_def_1:pre_add (p.fst+q.fst,p.snd+q.snd) r=k:=rfl,
  have k_def_2:pre_add p (q.fst+r.fst, q.snd+r.snd)=k:=eq.symm
    pre_assoc,
  have pre_1:=op2_reduction add_is_well_defined x_h c_h k_def_1,
  have pre_2:=op2_reduction add_is_well_defined a_h y_h k_def_2,
  unfold add_ourInt,
  exact (rfl.congr (eq.symm pre_2)).mp pre_1,
end

@[simp] lemma add_zero_ourInt (a:ourInt): a add zero = a:=
begin
  unfold add_ourInt,
  unfold induced_op2,
  unfold induced_op2_by_sec,
  unfold pre_add,
  rw ← can_par_sec_expand diff_is_equiv a,
  rw ← equiv_iff_same_image_under_can diff_is_equiv,
  simp,
  unfold diff,
  simp,
  rw zero_reps,
  rw add_comm (particular_sec diff_is_equiv a).fst,
  rw add_comm (particular_sec diff_is_equiv a).snd,
  rw add_assoc,
  rw add_assoc (particular_sec diff_is_equiv zero).snd,
  simp,
  rw add_comm,
end

@[simp] lemma zero_add_ourInt (a:ourInt): zero add a = a:=
begin
  rw add_comm_ourInt,
  exact add_zero_ourInt a,
end
```

```
@[simp] lemma add_inv_ourInt(a:ourInt):a add (inv_ourInt a) =
    zero
:=begin
  unfold add_ourInt,
  unfold induced_op2,
  unfold induced_op2_by_sec,
  unfold pre_add,
  unfold zero,
  rw ← equiv_iff_same_image_under_can diff_is_equiv,
  unfold diff,
  simp,
  unfold inv_ourInt,
  unfold induced_op,
  unfold induced_op_by_sec,
  unfold pre_inv,
  rw inv_reps _ _,
  simp,
  unfold pre_inv,
end

@[simp] lemma inv_add_ourInt(a:ourInt): (inv_ourInt a) add a =
    zero
:=begin
rw add_comm_ourInt,
  exact add_inv_ourInt a,
end

instance: group (ourInt):=
{ mul := add_ourInt,
  mul_assoc := add_assoc_ourInt,
  one := zero,
  mul_one := add_zero_ourInt,
  one_mul:= zero_add_ourInt,
  inv := inv_ourInt,
  mul_left_inv := inv_add_ourInt
}
```

Appendix B Category B Codes

```
import tactic
import ring_theory.ideal.basic
import ring_theory.ideal.operations
noncomputable theory
```

```
open_locale big_operators

universes ua ur
variables {α:Type ua} {R : Type ur}
variables [comm_ring R]

lemma some_nonempty{P:α→ Prop}(hw: ∃ r:α, P r):
  {j:α | P j}.nonempty:=
begin
  cases hw with r hw,
  use r,
  simp,
  exact hw,
end

def construct_some{P:α→ Prop}(hw: ∃ r:α, P r):α:=
  set.nonempty.some (some_nonempty hw)

lemma some_prop{P:α→ Prop}(hw: ∃ r:α, P r):
  P (construct_some hw):=
begin
  have some_in_set:construct_some hw∈ {j:α | P j}:=
    (some_nonempty hw).some_mem,
  simp at *,
  exact some_in_set,
end

lemma sum_intersect_lemma{n:ℕ}(I:finset.range n → ideal R)
(h:∀ (i j:finset.range n),i ≠ j→ (I i)⊔(I j) = ⊤)
(i:finset.range n) : ∃ r:R, r-1 ∈ I i ∧ (∀ j≠ i,r∈ I j) :=
begin
  let comple:=(finset.univ:finset (finset.range n)).erase i,

  have decomp: ∀ (j:finset.range n), j∈comple →
    ∃ (a∈I i)(b∈I j), a+b=(1:R) :=
  begin
    intros j f,
    have i_ne_j:=ne.symm (finset.ne_of_mem_erase f),
    have hij:=h i j i_ne_j,
    rw [ideal.eq_top_iff_one, submodule.mem_sup] at hij,
    exact hij,
  end,

  have decomp': ∀ (j:finset.range n), ∃ b:R, j∈comple →
    ((b∈ I j) ∧ (ideal.quotient.mk (I i) b)=1) :=
  begin
```

```
    intros j,
    by_cases f: j ∈ comple,
    have pre:=decomp j f,
    cases pre with a pre,cases pre with H_a pre,
    cases pre with b pre,cases pre with H_b pre,
    use b, intro _, split,
    exact H_b,
    have one_minus_a:=eq_sub_of_add_eq' pre,
    rw one_minus_a,
    simp,
    apply ideal.quotient.eq_zero_iff_mem.2,
    exact H_a,
    use 0,
  end,

  let b_pick:finset.range n→ R:=λj, construct_some (decomp' j),
  use ∏ j in comple,(b_pick j),
  split,

  rw ← ideal.quotient.eq,
  rw [ring_hom.map_one, ring_hom.map_prod],
  apply finset.prod_eq_one,
  intros x f,
  exact (some_prop (decomp' x) f).2,

  intros j f,
  have j_in_univ:=finset.mem_univ j,
  have j_in_comple:=finset.mem_erase_of_ne_of_mem f j_in_univ,
  rw ← finset.prod_erase_mul comple b_pick j_in_comple,
  apply ideal.mul_mem_left (I j) _,
  exact (some_prop (decomp' j) j_in_comple).1,
end

theorem CRT_ring_version_general{n:ℕ}(I:finset.range n → ideal R
   )
(h:∀ (i j:finset.range n),i ≠ j→ (I i)⊔(I j) = ⊤):⊓
( i, I i).quotient ≃+* (Π i, (I i).quotient) :=
begin
  let f: R → (Π(i:finset.range n), (I i).quotient) :=
    λr,(λ i, ideal.quotient.mk (I i) r),
  have f_def: ∀ r:R, f r =
    (λ i:finset.range n, ideal.quotient.mk (I i) r) :=
    congr_fun rfl,
  have f_def': ∀ r:R,∀ i, f r i=ideal.quotient.mk (I i) r :=
    by {intro r, have temp:=f_def r, exact congr_fun rfl,},
  have f_expand: ∀ a,∀ b, f a = b →
    (∀ x:finset.range n, f a x=b x) :=
    by {intros a b cond,rw cond,exact congr_fun rfl,},

  let hom: R →+* (Π i, (I i).quotient):=
  begin
    fconstructor,
    exact f,
    ext1,simp,rw f_def' _ _,
    simp,intros x y,
    ext1,simp, repeat {rw f_def' _ _},simp,
    ext1,simp,rw f_def' _ _,
    simp,intros x y,
    ext1,simp, repeat {rw f_def' _ _},simp,
  end,

  have ker_of_hom: hom.ker = ⊓( i, I i):=
  begin
    ext1,
    split,

    intro x_in_ker,
    have x_in_ker':=(ring_hom.mem_ker hom).mp x_in_ker,
    simp at x_in_ker' ⊢,
    intro i,
    have x_in_ker'':=f_expand x 0 x_in_ker' i,
    simp at x_in_ker'', rw f_def' at x_in_ker'',
    exact ideal.quotient.eq_zero_iff_mem.1 x_in_ker'',

    intro x_in_intersect,
    simp at x_in_intersect,
    have x_in_I: ∀ i, (ideal.quotient.mk (I i)) x=0 :=
    begin
      intro i,
      exact ideal.quotient.eq_zero_iff_mem.2 (x_in_intersect i),
    end,
    have x_in_ker': hom.to_fun x=0 :=
      by {simp,ext1,simp,rw f_def' _ _,exact x_in_I x_1,},
    exact (ring_hom.mem_ker hom).mp x_in_ker',
  end,

  have hom_surjective: function.surjective hom:=
  begin
    simp,
    unfold function.surjective,
    intro b,
  have has_preimage: ∀i,∃r, ideal.quotient.mk (I i) r = b i:=
  begin
    intro i,
    have sur:function.surjective (ideal.quotient.mk (I i))
      :=ideal.quotient.mk_surjective,
    exact sur (b i),
  end,
  let b_pick: finset.range n → R :=
    λi, construct_some (has_preimage i),
  have b_pick_def: ∀ i, b_pick i =
    construct_some (has_preimage i) := congr_fun rfl,

  have sum_intersect:= sum_intersect_lemma I h,
  simp at sum_intersect,
  let r_pick: finset.range n → R :=
    λi,construct_some (sum_intersect i),
  have r_pick_def: ∀ i, r_pick i =
    construct_some (sum_intersect i) := congr_fun rfl,

  let g:finset.range n → R:=λi,(b_pick i)*(r_pick i),
  have g_def:∀ i, g i=(b_pick i)*(r_pick i):=congr_fun rfl,
  let r:= Σ i , g i,
  use r,

  rw f_def r,
  ext1,
  have preimage_of_bx:=some_prop (has_preimage x),
  rw ← preimage_of_bx,
  rw ideal.quotient.eq,
  rw ← b_pick_def x,

  let comple:=(finset.univ:finset (finset.range n)).erase x,
  have x_in_univ:=finset.mem_univ x,
  have x_notin_comple:=finset.not_mem_erase x finset.univ,
  have r_another_express: r=(Σ i in comple, g i)+g x :=
    by {rw finset.sum_erase_add _ g x_in_univ},

  have rx_property:= some_prop(sum_intersect x),
  let rx_split:= r_pick x-1,
  have rx_split_def: rx_split=r_pick x-1 := rfl,
  have rx_in: rx_split ∈ I x := rx_property.1,
  have gx_express: g x=(b_pick x)+(b_pick x)*rx_split :=
    by {rw g_def,rw rx_split_def,ring,},
  have final_express:
    r-b_pick x=(b_pick x)*rx_split+(Σ i in comple, g i) :=
    by {rw r_another_express,rw gx_express,ring,},
      rw final_express,

      have first_term_in:=
        ideal.mul_mem_left (I x) (b_pick x) rx_in,
      apply ideal.add_mem (I x) first_term_in,

      let p: R → Prop := λu, u∈ I x,
      have p_def: ∀ u:R,(p u)=(u∈ I x) := congr_fun rfl,
      have p0: p 0 := ideal.zero_mem (I x),
      have p_any: ∀i∈comple,  p(g i):=
      begin
        intros i fi,
        have x_ne_i: x ≠ i :=
          by { contrapose fi, simp at fi,
               rw ← fi, exact x_notin_comple, },
        have ri_property:=some_prop(sum_intersect i),
        have in_x:=ri_property.2 x x_ne_i,
        rw [p_def (g i), g_def i],
        rw ← r_pick_def at in_x,
        exact ideal.mul_mem_left (I x) (b_pick i) in_x,
      end,
      have p_add: ∀ a b , p a → p b → p (a+b) :=
      begin
        intros a b,
        rw p_def,
        exact ideal.add_mem (I x),
      end,

      exact finset.sum_induction g p p_add p0 p_any,
    end,

    rw ← ker_of_hom,
    exact ring_hom.quotient_ker_equiv_of_surjective hom_surjective,
  end
```