Formalising Mathematics

Project 2

Additi Pandey

02119403

MSc in Pure Mathematics

Imperial College London

Overview of the Project

Motivation: Ideals form a Complete Lattice
The project is based on defining ideals from a commutative ring R, their infimum and supremum, the top and the bottom elements, proving their inclusion, and constructing some lemmas to give them a complete lattice structure. A lattice L is said to be *complete* if :

1. Every subset S of L has a least upper bound, and
2. Every subset S of L has a greatest lower bound.

A complete lattice has a top and a bottom element. So, after defining ideals from scratch under the structure called `my_ideal`, using Lean's commutative rings and the ideal test, I constructed lemmas to show their inclusion, the existence of a top element (the whole ring), and a bottom element (zero ideal), their join (sum), and meet (intersection).
I then took a slightly different approach to define the complete lattice in the project. Instead of directly defining the supremum, I defined infimum of two ideals I, J and a (possibly an infinite) set of ideals. This construction takes all the ideals that contain I, J, and takes their infimum, which gives the supremum of two ideals I, J as well as that of a possibly finite set of ideals. However, this approach could not be used when performing concrete calculations. I declared a new instance called `has_add`, which defines the ideals of the form $I+J$. I defined several smaller lemmas like $I \leq I \sqcup J$ and $I \leq I+J$ to show that the Lean's supremum called `has_sup` is the same as the sum of ideals $I + J$. This gave a formula to Lean's definition of supremum `has_sup`, which can be used for calculations when dealing with ideals.

Explanation of the code

I first imported all the Lean tactics and `set_like.basic`, which is a typeclass for types with an extensional property, that is, the property that "things are equal if and only if they have the same elements". I then defined the ideals by the Ideal Test using Lean's commutative rings.
Note: Throughout the code, I have denoted the ideal inclusion by $\leq$ rather than by $\subseteq$.

0.1 Defining Ideals and their Inclusion

Definition of an ideal:
A subring A of a ring R is called a (two-sided) ideal of R if for every $r \in R$ and every $a \in A$ both ra and ar are in A. So by Ideal Test, we say: A nonempty subset A of a ring R is an ideal of R if

1. ra, ar are in A whenever $a \in A$ and $r \in A$.
2. $a - b \in A$ whenever $a, b \in A$.

I used Lean's commutative ring and denoted by R, and defined the ideals as left-sided ideals by the Ideal Test. I took `zero_mem'` to represent the non-empty subset of R, as every ring

has the trivial ideal. I took the first axiom to be represented by `mul_mem'` and since the ring is commutative, I just defined them in the form of left ideals. I then represented the second axiom by `sub_mem'`. So my definition of ideals of commutative rings in Lean, represented by a structure called `my_ideal` became as follows:

```
structure my_ideal (R : Type*) [comm_ring R] :=
(carrier : set R)
(zero_mem' : (0 : R) ∈ carrier)
(mul_mem' {a b} : b ∈ carrier → a * b ∈ carrier)
(sub_mem' {a b} : a ∈ carrier → b ∈ carrier → a - b ∈ carrier)
```

I then went onto defining the instance called `set_like` which defines an injective map from `my_ideal R` to the set R. It implies that every ideal is a subset of R, and different ideals correspond to different subsets. I then defined `mem_carrier` and `ext`. While the former shows that an element is in the underlying carrier set of the ideal $\iff$ it is in the ideal as well, the latter shows that two ideals are equal $\iff$ they have the same elements.
The `set_like` already defines inclusion of ideals by considering them a partial order with induced ordering as subsets of R are a partial order. However, I made an attempt to show inclusion more exclusively. Given two ideals I, J, we say that $I \leq J \iff$ all elements of $I \leq$ all elements of J, both satisfying the same sets of axioms defined by `my_ideal`.
I introduced the definition of `copy`, followed by lemmas `coe_copy` and `copy_eq`, which replace the copy of `my_ideal` with a new 'carrier' equal to the old one and help to fix definitional equalities.
I proved the axioms of `my_ideal`, starting with `zero_mem`, and moving on to the two properties constructing them as theorems called `mul_mem` and `sub_mem`. Using these three axioms, along with the help of another axiom, which said that all ideals contain the additive inverses of their elements, I proved `add_mem`, showing that ideals are closed under addition. Since, my ring R is commutative and I defined `my_ideal` as left ideals, I introduced another theorem called `mul_mem_right` defining the right ideals.

0.2 Defining Bottom and Top Element

After defining the ideals, their inclusion and equality, I introduced their bottom and top element into the code. Joseph. A. Gallian's book titled, "Contemporary Abstract Algebra" ($7th$ Edition) states in the chapter 'Ideals and Factor Rings' that
"For any ring R, 0 and R are ideals of R."
Therefore, the bottom element is definitely the zero ideal. Since, no ideal can be bigger than the ring itself, the top element of a ring R is always the ideal R. So, in order to construct these ideals, I defined them in the underlying `carrier` set and proved that all axioms of `my_ideal` hold for them as well.

0.3 Defining Intersection and Sum of Two Ideals

After I had defined inclusion, bottom and top element of ideals, I went onto defining the intersection of two ideals.

Let $S = \{J_i \mid i \in I\}$ of (left) ideals of a ring R, where I is an index set.
First, $\sqcap S$ is a left ideal of R: if $a, b \in \sqcap S$, then $a, b \in J_i$ for all $i \in I$. Consequently, $a - b \in J_i$ and so $a - b \in \sqcap S$. Furthermore, if $r \in R$, then $r * a \in J_i$ for any $i \in I$, so $r * a \in \sqcap S$ also. Hence $\sqcap S$ is a left ideal. By construction, $\sqcap S$ is clearly contained in all of J_i, and is clearly the largest such ideal.
Hence, working exactly along the definition, I constructed the infimum of two ideals given by the instance `has_inf` as:

```
instance : has_inf (my_ideal R) :=
⟨ I J,
  { carrier := I  J,
    zero_mem' :=
    begin
    fconstructor, apply zero_mem,apply zero_mem,
    end,
    mul_mem' :=
    begin
    intros a b c, fconstructor, simp at c, cases c, simp, apply mul_mem,
    exact c_left, simp at c, cases c, simp, apply mul_mem, exact c_right,
    end,
    sub_mem' :=
    begin
    intros a b c d, simp at c, simp at d, cases c, cases d, simp,
    split, apply sub_mem, exact c_left, exact d_left, apply sub_mem,
    exact c_right, exact d_right,
    end
  }
⟩
```

This is the infimum of two ideals and of (possibly an infinite) sets of ideals under the instance `has_Inf`. Now, Lean can define the supremum of ideals using the definition of `has_Inf`, but I am going to define another instance called `has_add`, which gives the formula to define the supremum of ideals. This defines an ideal which is of the form $I + J$ for any two ideals I, J and represents the Lean's supremum for the ideals I, J.

0.4 Defining Complete Lattice

Given the infimum of two ideals as well as of (possibly an infinite) sets of ideals, Lean creates a supremum of two ideals and of (possibly an infinite) sets of ideals. Since both $\sqcup S$ and $\sqcap S$ are well-defined, and exist for finite S, so `my_ideal` is a lattice. Additionally, both operations work for arbitrary S, which is a set of left ideals, so `my_ideal` is complete. Hence, I defined a complete lattice under the instance `complete_lattice`:

```
instance : complete_lattice (my_ideal R) :=
{ le           := (≤),
```

```
  lt           := (<),
  bot          := (⊥),
  bot_le       :=  S x hx, (mem_bot.1 hx).symm  S.zero_mem,
  top          := (⊤),
  le_top       :=  S x hx, mem_top x,
  inf          := (⊓),
  Inf          := has_Inf.Inf,
  le_inf       :=  a b c ha hb x hx, ⟨ha hx, hb hx⟩,
  inf_le_left  :=  a b x, and.left,
  inf_le_right :=  a b x, and.right,
  .. complete_lattice_of_Inf (my_ideal R) $  s,
    is_glb.of_image ( S T,
      show (S : set R) ≤ T  S ≤ T, from set_like.coe_subset_coe) is_glb_binfi }
```

0.5 Linking $I + J$ to Lean's Supremum

I had claimed earlier in the project, when defining `has_add` that it is the supremum of two ideals. However, given the instances `has_inf` which is the intersection of two ideals I, J and `has_Inf`, which is the arbitrary intersection of (possibly infinite) sets of ideals, Lean defines `has_sup` for the ideals I, J by taking the infimum using `has_Inf` of all the sets that contain I, J. So, it already has a supremum for a pair of ideals which help in giving it a lattice structure, without having to declare an instance for it. However, I claimed earlier that this supremum defined by Lean is $I + J$ as defined by the instance `has_add`.
In this section, I will walk through the thought-process and code of linking $I + J$ to Lean's `has_sup`. I had to prove the lemma that $I + J = I \sqcup J$. But I cannot prove that they are equal until I show it for some arbitrary element x, and use it to prove that $I + J \leq I \sqcup J$ and $I \sqcup J \leq I + J$. So I started off by defining an element of R, belonging to $I + J$ as:

```
@[simp] lemma has_add_mem (x : R) : x ∈ I + J   (i ∈ I) (j ∈ J), x = i + j :=
begin
  split,
  tauto,
  itauto,
end
```

The lemma `has_add_mem` shows that any element x in $I + J$ can be represented by $x = i + j$, where $i \in I$ and $j \in J$. I then moved onto proving the lemma:

```
lemma has_add_eq_has_sup : I + J = I ⊔ J :=
```

To prove this equality, I had to show that their elements are also equal.
That is, $x \in I + J \iff x \in I \sqcup J$. So, I used the lemma `ext` that says that two ideals are equal if they have the same elements. Since, this lemmas is true $\forall(x : R)$, so I fixed an arbitrary element x. Then, I "split" the lemma into $x \in I + J \implies x \in I \sqcup J$ and

$x \in I \sqcup J \implies x \in I + J$.
To prove the first part, I took $x \in I + J$. I then proved that for $x = p + r$, $p \in I$, $r \in J$:

```
hs: x = p + r
hv1: J ≤ I ⊔ J
hv: r ∈ I ⊔ J
hu1: I ≤ I ⊔ J
hu: p ∈ I ⊔ J
```

For the second part, which was to show that: $x \in I \sqcup J \implies x \in I + J$, I used $I \sqcup J \leq I + J$, to show that any $x \in I \sqcup J$, is contained in $I + J$. So, to prove the fact used, I introduced and proved the hypotheses:

```
h1: I ≤ I + J
h2: J ≤ I + J
```

Since, Lean has `has_sup` defined already, there is an axiom called `sup_le` defined in it, which says that $\forall\ a, b, c$ being elements of $\alpha, a \leq c \implies b \leq c \implies a \sqcup b \leq c$, here $\alpha : Type\ u$, for some universe u. So, given $h1,\ h2$, it could prove that $I \sqcup J \leq I + J$.

At the end of the code, I closed the namespace by `end my_ideal` and the `comm_ring` section. I also made sure to rectify any errors that the linter generates. I passed all the linting checks, and this concluded the code!

What I found hard and/or surprising

We were supposed to pick up a topic from Year 2 and above to work on this project. However, I found it hard to think of a topic, and once I decided on it, it would be even harder to think about what I would do in it. Initially, I thought I would work with Fields – define them from scratch, prove their elementary theorems and talk about the characteristic of a field. However, I could not think of an interesting way to bridge the the elementary theorems with the characteristic of a field. So, I discarded the idea and switched to ideals. I thought of defining their structure and proving theorems about the prime and the maximal ideals. This idea, however, did not appeal to me. Hence, I found it really hard to come up with a definite outline for the project and it consumed majority of my time. Following the advice of Professor Buzzard, I started working on defining the ideals and giving them the structure of a complete lattice.
I wanted to make sure that I understand each part of the code I write so that I am able to prove every lemma and theorem without using `sorry`. I had a fear engraved in my mind that perhaps, I cannot code in Lean, because it is too hard for me. During the time following the submission of my first project to finishing this one, I believe my thought process has undergone drastic changes. I realised that while I could understand what a `structure` and a `class` are, I could not link them mathematically to envision and understand their working in Lean. The syntax seemed mechanical to me because I could not logically comprehend

what it meant. For instance, I thought that the ideals defined by `my_ideal` can solely be treated as sets due to their underlying `carrier` set, but I did not realise that just one axiom defined in the structure `my_ideal` cannot alone be considered, as the `structure` comes with a set of defined conditions which need to be fulfilled.
Another problem that I faced was understanding comprehending the information supplied in the Lean Infoview. While proving `has_add`, I had to show that it satisfies all axioms of the structure `my_ideal`, but I could not proceed after proving for `zero_mem'` as I was unable to understand what Lean Infoview exactly meant. I misunderstood that representing two variables by the same letter at different places does not mean that they are the same variables!
I found it hard translating the proof that I had on paper to Lean. The reason is still the same- I am sometimes unable to make sense of the syntax. On paper, it is easier to write a proof as we understand what the person reading might be familiar with; however, on Lean, it is hard as I could not gauge what Lean could understand. But often, Lean amazes me. Seeing it smoothly prove certain theorems and lemmas, like constructing `has_sup` based on other lemmas used in the code has baffled me. I am really excited but equally nervous, to understand Lean better, interpret the errors and goals smoothly, and grow in mathematical knowledge as Lean always digs deeper into the most fundamental theorem, propelling me to think and link, in ways I never thought I could!

What I learnt

Thinking about representing mathematical structures in the form of a lattice seems like an easy task to do visually, as well as on paper. However, it is equally enjoyable in Lean. Since similar programs are there for mathematical structures such as submonoids and subgroups, it was not hard to understand the outline and envisage the direction of the project. However, creating the complete lattice structure for ideals was a tremendous learning experience.
Comparing my progress from the last project to this one, I have seen immense growth in my understanding of building lemmas and theorems in Lean. While I had a clear idea of proceeding throughout the project, I faced difficulty in writing the proofs of lemmas and theorems. Given my experience in the last project, where I was unable to prove g^{-1} is inverse of g, where $g \in$ group G, I constantly thought that there must be some missing intermediate steps in the mathematical proofs that I have on paper. Somewhere, the experience I had with my last project overshadowed the excitement and affected my approach to think logically in this project. However, I soon realised that the project has immensely improved my comprehension of theorem proving in Lean. Consequently, I was clear about the aim of each lemma and theorem throughout the project. I learnt the way to ask the right questions. I have developed the ability to find out what piece of information I require to proceed while proving.
I believe that talking to people who just started with Lean, and are taking the same course or are doing a Ph.D. in this domain, has helped me see through the way they think and code—watching Professor Buzzard thinking and coding in Lean has taught me how to think mathematically at the most fundamental level and build up on it while coding.
In addition to this, I have started reading and interpreting the errors that the code generates. I now understand that Lean points the error and tells what could be provided to fix it.
Although I still have a long way to go in order to be able to code in Lean swiftly, I am

happy that I have learnt a lot in this project and have effectively employed my learning of the previous project to be able to finish the code without any `sorry` or `lint` errors.
I hope to keep learning and use all I have learnt to understand and interpret Lean better, in order to grow mathematically and as a Lean coder!