Formalising Mathematics

Project 1

MSc in Pure Mathematics
Imperial College London

Overview of the Project

Motivation: Group Theory in Lean

The project is based on some parts of the first question of the past year paper of Year 1 (1174449). The aim of the project is as follows:
For a group G, define the following statements:

1. G is abelian
2. G is cyclic
3. All cyclic groups are abelian.

A group G is called abelian (or commutative) if $a * b = b * a \ \forall \ a, b \in G$.
A group is called cyclic when it can be generated by a single element.
Let G be a group under $*$.
If there is some element $x \in G$ such that $G = \{x^n | n \in Z\}$ then, G is cyclic.
Here, G is generated by x (and x is a generator of G). To prove this, I started with the formation of 'mygroup' and defined the group axioms for it, namely, associativity, existence of identity and existence of inverse.
I created a class 'abel_grp' which is the set of all abelian groups satisfying all properties of 'mygroup' along with the commutativity axiom. I then defined integer powers needed to define every element of cyclic groups and created a class 'cyclic_grp'. To prove that all cyclic groups are abelian, I defined a function called 'npow' which defined non-negative powers over $g \in G$ in the form of $g^0 = 1$ and $g^{(n+1)} = g^n * g$, for n non-negative. After this, I attempted to extend it to integer powers to prove the theorem. For that, I stated and proved basic lemmas for non-negative powers (defined by 'npow') over g and used them to prove results about integer powers (defined by 'zpow') over an element $g \in G$. Then, I used these lemmas to get the following, which helps to show that cyclic groups commute in order to prove them abelian:

```
lemma pow'_comm: zpow g a * zpow g b = zpow g b * zpow g a
```

The code translates to:

$$a, b \in Z : g^a * g^b = g^b * g^a$$

Explanation of the code

Note: Throughout this project, we assume that natural numbers begin with zero as is standard in Lean and are denoted by N. However, I have referred to them as non-negative numbers in text and denoted them by N.

0.1 Defining a Group

A group is a set G with a binary operation satisfying associativity, existence of inverse and existence of an identity.

I defined a class 'mygroup' (to represent the groups of type G) that extends *has_one* G, *has_mul* G, *has_inv* G implying that G has a multiplication $* : G \to G \to G$, an identity $1 : G$ and an inverse $^{-1}$ such that $g^{-1} : G \to G$ for $g \in G$.

0.2 Defining an Abelian Group

If a group has the property that $a * b = b * a$ for every pair of elements a and b, we say the group is abelian. All commutative groups are abelian.

```
class abel_grp (G : Type)
extends mygroup G : Type:=
(mul_com : ∀ a b : G, a * b = b * a)
```

As in the image of the code, I defined a class 'abel_grp' which has all properties of 'mygroup' with an extra axiom of commutativity defined by 'mul_com'. To prove that commutativity holds in an abelian group H, the lemma 'mul_comu' is defined. I proved this lemma using 'mul_com' of the abelian groups - which is defined in the class 'abel_grp'.

0.3 Defining Cyclic Groups

To define the cyclic groups, it is important to define powers that an element of 'mygroup' will be raised to. In other words, let g be the generator of a cyclic group G. We need to define integer powers so that every element can be represented in form of some power of g. Since, it is easier to work with non-negative numbers, I first defined 'npow':

```
def npow (g : G) : ℕ → G
| 0 := 1
| (n+1) := npow n * g
```

The 'npow' is a function from $N \to G$ such that for any $g \in G$, $g^0 = 1$ and $g^{n+1} = g^n * g$. However, cyclic groups are defined as $G = \{g^n | n \in Z\}$. So, using the power function defined for non-negative numbers I defined a function for integer powers that an element (a generator) in 'mygroup' G will be raised to, so as to form a class of cyclic groups. This helps to define a function for integer powers called 'zpow':

```
/- We now extend to defining powers for integers-/
def zpow (g : G) : ℤ → G
| (int.of_nat n) := npow g n
| (int.neg_succ_of_nat n) := (npow g (n + 1))⁻¹
```

The function raises $g \in G$ to an integer power such that it maps a non-negative number to either g^0 or $g^{(n+1)}$, and $(g^{(n+1)})^{-1}$ for a negative number such that here, $n = 0$ maps to g^{-1}. After having defined integer powers, the definition of cyclic groups can be formed as:

```
class cyclic_grp (G : Type)
extends mygroup G : Type :=
(grp_generator [] : ∃ g : G, ∀ a : G, ∃ n : ℤ , a = zpow g
```

Thus, cyclic groups represented by the class 'cyclic_grp' (as in the image of the code above) satisfy all axioms of 'mygroup', as well as have a generator defined by 'grp_generator [ ]', such that all elements in the cyclic group G can be defined in the form of some integer power of $g \in G$.

0.4 Every Cyclic Group is Abelian

I first defined attributes that are the axioms of 'mygroup' which will help in communicating information to Lean. I started with the proof of Socks-Shoes Property in Group theory which says that for $a, b \in G : (a * b)^{-1} = b^{-1} * a^{-1}$.
Mathematically, we will prove it as:

$$(a * b)^{-1} = b^{-1} * a^{-1}$$

Step 1:

$$a * b * (a * b)^{-1} = a * b * b^{-1} * a^{-1}$$

Step 2:

$$(a * b) * (a * b)^{-1} = a * (b * b^{-1}) * a^{-1}$$

Step 3:

$$1 = a * a^{-1}$$

Step 4:

$$1 = 1$$

So, to do Step 1, I needed a property to multiply both sides of the equation by $a * b$ so I created a lemma called 'left_cancel' which is left cancellation $a * b = a * c \implies b = c$. I also needed to apply left cancellation on the right hand side of the property while proving it. So I introduced a lemma 'inv_left_cancel' which states that for $a, b \in G : a * (a^{-1} * b) = b$. Then, using the group axioms defined for the class 'mygroup', I proved the property in the form of the lemma 'socks_shoes'.
Then, I introduced variables like an element $g \in G$ and elements $m, n, o \in N$. Some lemma like $g^0 = 1$ and $g^1 = g$, were added as 'simp' lemma so that they can simplify the steps later in the proof of $g^m * g^n = g^{(m+n)}$, which will eventually help prove that all cyclic groups are commutative.
To understand this further we will take a closer look at the steps for proving commutativity in group elements raised to non-negative power $m, n \in N$:

```
lemma add_succ: npow g (m + n.succ) = npow g (m+n).succ := rfl

-- To show: `npow g (m+n) = npow g m * npow g n`
@[simp]lemma pow_add: npow g (m+n) = npow g m * npow g n :=
begin
induction n with d hd,
simp,
rw mul_one,
simp,
rw add_succ,
rw pow_succ,
rw [hd, mul_assoc],
end

-- To show: `npow g m * npow g n = npow g n * npow g m`
@[simp]lemma pow_comm: npow g m * npow g n = npow g n * npow g m :=
begin
induction n with d hd,
simp,
rw mul_one,
rw [← pow_add, add_comm, pow_add],
end
```

To prove $g^{(m+n)} = g^m * g^n$, we proceed by induction on n. The base case

```
npow g (m + 0) = npow g m * npow g 0
```

can be easily deduced from the property of non-negative numbers and 'pow_zero' (which states that every element of G raised to power 0 is 1). However, to prove for the case of $n = d + 1$ or $d.succ$ as lean calls it, we will need both 'pow_succ' defined as:

```
pow_succ: npow g (n.succ)= npow g n  * g:= rfl
```

and 'add_succ' (as stated in the image) to solve in the following way:

```
npow g (m + d.succ) = npow g m * npow g d.succ
npow g (m + d) * g = npow g m * (npow g d * g)
```

and since these lemmas were already tagged with '@[simp]', the keyword simplified the following process for us so that we can use our induction hypothesis 'hp' to easily prove the lemma:

$$g^{m+(d+1)}$$

$$= g^{(m+d)+1}$$

$$= g^{m+d} * g,\ (m+d) \in N$$

Let $(m+d) = t \in N$ as I had already proved $g^{(t+1)} = g^t * g$ (see: lemma 'pow_succ' in the image above), using the induction hypothesis, followed by group axioms, gives the proof that $g^{(m+n)} = g^m * g^n$. This proof has simplified the attempt to show commutativity in cyclic groups. It has provided an approach to deal with integer powers from the perspective of non-negative powers.

However, to prove that cyclic groups commute, it is important to show that for $a, b \in Z$: $g^a * g^b = g^b * g^a$. For this, I needed several smaller lemma like 'pow_neg' and 'pow_pos' which make it easier to prove when encountered with g raised to a power in the form 'of_nat p' or 'neg_succ_of_nat p' as they turn them into g^p or $(g^{(p+1)})^{-1}$ for $p \in N$.

Having given a brief look at the lemma, I realised that this lemma can be proven in the way similar to the non-negative powers. However, to facilitate this ease, I needed the lemma 'zpow_add' that states that $g^a * g^b = g^{(a+b)}$ for the integers $a, b \in Z$. Proving 'zpow_add' by induction required several other lemmas like $a^{(n-1)} = a^n * a^{-1}$ and $g^a * g = g^{(a+1)}$. Lemmas dealing with commutativity of g with integer powers and non-negative powers, helped immensely when attempting to prove by converting integer powers into non-negative numbers by means of 'npow_zpow' (converts a positive integer power into a non-negative number power), as they are easier to deal with.

Once I had proven that $a, b \in Z : g^a * g^b = g^b * g^a$, I moved towards proving the theorem that every cyclic group is abelian. For that, I took a cyclic group K from class 'cyclic_grp', and declared elements $x, y \in K$. To show that $x * y = y * x$, I needed to write them down in the form of $g^n; n \in Z$

```
theorem cycl_abel {K : Type} [cyclic_grp K] (x y : K):
x * y = y * x:=
begin
obtain (g, hg) := grp_generator K,
obtain (m, hm) := hg x,
obtain (n, hn) := hg y,
rw hm,
rw hn,
rw pow'_comm,
end
```

As in the image of the code above, with the generator $g \in K$, the elements $x, y \in K$ can be written in the form of $x = g^m, y = g^n;\ m, n \in Z$. Then, using the commutativity of elements in G, raised to integer powers (defined by lemma 'pow'_comm'), it was proved that x, y commute in K. Since x, y are arbitrary elements in K so, all elements in K commute.

Therefore, K is an abelian group.

```
def cyclic_groups (P : Type) [cyclic_grp P] : abel_grp P :=
{ mul_assoc := mul_assoc,
one_mul := one_mul,
mul_one := mul_one,
inv_mul_self := inv_mul_self,
mul_inv_self := mul_inv_self,
mul_com := cycl_abel
```

Towards the end, I defined an arbitrary type P, which is a cyclic group, as an abelian group since it satisfies all group axioms defined in the class 'mygroup' as well as the property 'mul_com'- which is commutativity in abelian groups.
Therefore, I proved that every cyclic group is abelian.

What I found hard and/or surprising

After playing the 'Natural Number Game', I was very excited about learning Lean. It seemed very much like a puzzle. When the 'Formalising Mathematics' course started in this term, the sheets made me realise the importance of clarity in concepts to solve these puzzles. It took me some time to get used to the tactics and the coding environment, but I enjoyed working on the sheets. When I chose this topic for my first project, I thought it was going to be similar to what we did in the lectures. I had an idea of how to start and the topics seemed easier, primarily because we all know the definitions and proofs of the year 1 group theory theorems by heart. So, I decided to start from the scratch and create my own groups and call their class 'mygroup'. It was while doing this project that I realised how hard proving even the basic concepts could be. It made me understand that Lean is not all about mentally proving steps and applying tactics to see the outcome, in order to proceed further. It is, rather, about proving theorems and lemmas at their most fundamental level.
Since it was easier to deal with non-negative numbers as compared to the integers, I did not encounter much issue with my approach then. However, when I was working with integer powers, despite making several lemmas, I could not think of ways to proceed with induction on integers and did not get any closer to the lemma that I actually intended to prove. It was not until Professor Buzzard made me realise that I need to work backwards. Instead of looking at the final lemma to be proven, I went astray by proving numerous other lemmas which I did not require. It was while proving the final lemma, and creating intermediate lemmas to help me complete the proof that I realised that the lemmas I thought might help me later on, did not come to any use.
While there are a lot of things I learnt to prove and a lot of areas I understand I have to improve in, I believe, however that once I improve my approach of attempting a problem or proof, by practicing and getting used to the environment, I will be able to handle Lean in a much better way.
When proving that for any element $g \in G$, $g^a * g^b = g^b * g^a \quad a, b \in N$, I found it easy to mentally prove it because I could think of ways to proceed without writing it on paper. However, when I moved to integer powers with this approach, I did find it very hard to prove everything from scratch. I also find it fascinating yet surprising that coding in Lean helps understand theorems at their most basic level. For instance, something like $g^1 = g$, is considered too obvious, but I am fascinated to find that I could actually prove it on paper as well as implement it on Lean! Going through the mathlib documentation, re-doing the sheets, and clarifying doubts and getting suggestions on how to code in Lean, made me

understand the ways to deal with proofs and surprisingly, taught me how to actually do Mathematics!

What I learnt

I learnt the way to approach a proof, not just on computer theorem prover systems like Lean but in general, as well. Proving a theorem or a lemma is not always about building over smaller lemmas. Sometimes that approach may result in numerous lemmas, none of which may help in the proof of the final theorem or lemma. I realised that looking at what the lemma is asking and proving it first is a more efficient way to build a proof. It helps realise the lemmas that are needed and prevents from going astray. Although it did take me a lot of time to get used to the approach of coding in Lean, yet, I tried my best to use all that I have learnt in the process to put forward this project. The Xena Project really taught me the way to think and proceed. I learnt about the errors and some ways to correct them. I learnt to tackle integer powers by using lemmas on non-negative numbers. Mathematically, I believe that it has improved my understanding of groups.
For instance, I was stuck on the proof of

```
lemma pow'_comm: zpow g a * zpow g b = zpow g b * zpow g a
```

It took me two whole days and yet, I could not come up with a solution. When I started working backwards, that is, proving this lemma and seeing the lemma I need, I had a much better understanding of what is needed. With the new approach that I learnt, I could figure out solutions to most of the smaller lemmas that this final lemma needed.
I learnt about several tactics and their implementation. My favorite one is 'int.coe_nat_succ', which immensely helped me in dealing with the cases where I had g raised to an integer power of the form $(\uparrow hd + 1)$ and wanted to change the integer in the exponent to the non-negative number. It helped in making $(\uparrow hd + 1)$ into $\uparrow (hd.succ)$ which can be changed to natural number easily, by using the lemma 'npow_zpow' which states that for any $p \in N$

```
npow g p = zpow g p
```

There were some proofs that I found very challenging, yet like a puzzle, working on them is very intriguing.
A very important lesson that I learnt is to actually understand the proof of the lemmas and then work on proving the lemmas that build it. I learnt that Lean is not all about applying tactics that I think will fit. It is about actually implementing the proof and using the theorem prover system to logically check if the proof is correct.
I believe that with this new approach and curiosity to learn, I will be able to handle the theorems, definitions and proofs in a much better way mathematically and in Lean, and will have a lot more to offer in the next projects!