Formalising Mathematics Coursework 1

[redacted]

1 What is going on in the code

In this project, I attempt(spoiler: inefficiently and unsuccessfully) to prove that disjoint cycles of a symmetric group commute. This turned out to be much harder than I had expected. In my first 50 or so lines, I defined the group of permutations, and showed it has a group structure using instance:

```
/-
Giving my_perm X group structure.
Inverse and identity "equiv" elements already exist, and we use them to get the group structure.
-/
instance my_symm_grp : group (my_perm X):= {
  mul := λ f g, equiv.trans g f, -- Multiplication in the group is given by composing the two bijections
  one := equiv.refl X,
  mul_assoc := λ f g h, equiv.trans_assoc h g f,
  -- Associativity of the group follows from associativity of bijections
  one_mul := by {intro a, simp},
  mul_one := by {intro a, simp},
  inv := equiv.symm,
  mul_left_inv := by {intro a, ext x, simp}, -- "simp" failed to simplify before I introduced x
}
-- Do not fully understand why I had to use "by" here and not "begin-end" stuff

@[simp] lemma my_perm_comp : (f * g).to_fun = f.to_fun ∘ g.to_fun  := begin ext x, simp, end
```

Afterwards, I proceed by defining what the orbit of an element is:

```
def orbit {X : Type} [fintype X] (f : my_perm X) (x : X) : set X := {y : X | ∃ n : ℤ, (f ^ n).to_fun x = y}
```

I define an equivalence relation on an arbitrary permutation f (that x and y are related if and only if they share an orbit):

```
/-
We define an equivalence relation using our definition of orbit
-/
inductive rel {X : Type} [fintype X] (f : my_perm X) : X → X → Prop
| a : ∀ (x y : X) (h : orbit f x = orbit f y), rel x y
```

With help from Yaël of the Xena Project discord, I define a function that gives me the size of the equivalence class of an arbitrary element x. What happens here is that there is a quotient being taken on the equivalence

relation defined, and we define the size of the class to be the fiber of the quotient map on the equivalence class. This definition turns out to be problematic, but I am not quite sure why that is.

```
def size_of_class {X : Type} [fintype X] (f : my_perm X) : X → ℕ :=
λ x, (finset.univ.filter $ λ y, orbit.rel f x = orbit.rel f y).card
```

Then, I define what it means to be a cycle. This definition turns out to be ultimately unhelpful, as disjoint permutations commute regardless of if they are cycles or not.

```
/-
A cycle is a permutation f with only one non-trivial orbit.
In this definition, I define `is_cycle` as a function taking the permutation f to the statement that there is only one non-trivial orbit.
-/
def is_cyc : (my_perm X) → Prop :=
λ f, ((∃ (x : X), (size_of_class f x) > 1) →  ∀ y : X, (size_of_class f y) > 1 → orbit.rel f x = orbit.rel f y )= true
```

Finally, I was able to make and prove the statement that disjoint cycles commute:

```
-- Defining what it means for cycles to be disjoint
def are_dj_cyc (f g : my_perm X) := ∀ (x : X), (size_of_class f x = 1) ∨ (size_of_class g x = 1)

#check are_dj_cyc X f g

-- Disjoint cycles commute
theorem dj_cyc_comm (h : are_dj_cyc X f g) : f * g = g * f :=
```

The proof of the final theorem is complete, however I have "sorry"ed some lemmas along the way.

2 What I struggled with

First we look at lemmas I was not able to prove. For the first one, everything I'd like to say here is already in the comments.

```
--Unused lemma, although I feel like I might have needed this to prove the next lemma if it wasn't "sorryed"
lemma class_non_empty {X : Type} [fintype X] (f : my_perm X) :
∀ x : X, 1 ≤ size_of_class f x :=
begin
  intro x,
  change 1 ≤ (finset.univ.filter $ λ y, orbit.rel f x = orbit.rel f y).card,
  -- I've got a statement here that says "The equivalence class of any x has cardinality at least 1",
  -- and I've got the obvious element x in that equivalence class, but I do not know how to proceed.
  sorry,
end
```

I simply do not have the Lean knowledge at the moment to work with finite sets. Really, this is 'trivial' in the mathematical sense; because the equivalence class of x is precisely the image of x under the quotient map, of course the fiber of said equivalence class has cardinality at least 1. However, I simply do not know what to do with a statement like this in Lean.

I run into a similar problem in this second lemma:

```
lemma has_triv_orbit (f : my_perm X) (x : X) : size_of_class f x = 1 ↔ (f.to_fun x = x) :=
begin
  split,
  {
    intro h,
    by_contra h1,
    sorry},
  {
    intro h,
    sorry}
end
```

What I'd like to argue is that since the fiber of the quotient map includes x, the equivalence class having size 1 is necessary and sufficient to show that the orbit of x under the action of f must be a singleton set. However, I again do not have the necessary knowledge yet to deal with statements like this in Lean.

Lastly, and perhaps the biggest problem in my coursework, is this definition we saw before of "size_of_class". Lean spits out an error telling me that the size of the set is "noncomputable", and says that it depends on "Prop.linear_order". I do not understand what it means for something to be noncomputable in Lean, so I believe I have defined this in completely the wrong way.

3 Things I have learnt

Throughout this coursework, I have learnt many (basic) things about Lean. I learnt to define function types (with 'def') and inductive types (with 'inductive'). I was also able to write my own lemmas.

When it comes to proving said lemmas, I gained a lot of familiarity with various tactics. However, I am still somewhat uncomfortable with the Lean way of "working backwards" in proofs, that is, starting from the goal and manipulating the goal until we get to the point that the hypothesis can be directly applied.

I learnt how to give a set (or a 'Type') a group structure using 'instance'. This helped me a lot, as I was able to immediately use familiar group notation on the set after using 'instance'.

I also learnt to define an equivalence relation, and partially learnt about quotient maps.

4 Things to improve

I can improve greatly on keeping statements simple. For instance, the way this theorem (disjoint permutations commute) was proven in matlib involved much simpler definitions of what it means to be disjoint. I did not necessarily need my permutations to be cycles, and could have avoided the whole mess that I got into with cardinality of finite sets and quotients if I had realised that what I wanted to prove could be done without needing cycles.

Another thing I need to improve on is familiarity with the lemmas in matlib, or rather, familiarity with looking them up by 'library_search'. I spend too long trying to find the right lemmas as I am not able to come up with the precise statement so that 'library_search' will immediately give me the lemma I want.

I also need to understand instances more. In this coursework, I included

```
{X:Type} [fintype X]
```

in a lot of definitions, but I do not know if they are absolutely needed.