IMPERIAL

Department of Mathematics
Imperial College London

Coursework 1: Group Actions

Student name: *Frankie Feng-Cheng WANG*

Course: *MATH70040-Formalising Mathematics* – Lecturer: *Dr Bhavik Mehta*
Due date: *January 26, 2026*

Contents

1. Introduction

This report formalises the theory of group actions in Lean 4. I define a custom action class, construct the permutation representation, and prove that stabilizers form subgroups.

1.1. Main result.

> **Theorem.** Let X be a G-set. For each $g \in G$, the map $\varphi_g : X \to X$ defined by $\varphi_g(x) = g \cdot x$ is a permutation of X. The map $\Phi : G \to \mathrm{Sym}(X)$ given by $\Phi(g) = \varphi_g$ is a group homomorphism, and for all $g \in G$ and $x \in X$, $\Phi(g)(x) = g \cdot x$.

1.2. Why formalise group actions?. Formalisation in Lean forces every step to be explicit: hidden assumptions become visible, informal "clearly" arguments must be justified, and type checking prevents mistakes. An external examiner familiar with group theory can read this report without prior Lean knowledge, as each section presents the mathematics first, then shows the corresponding Lean code. Lean's type system ensures that the permutation representation really is a group homomorphism and that stabilizers really are subgroups—claims that informal proofs sometimes take for granted.

2. Background and Design Choices

2.1. Why a custom action class. Mathlib already provides `MulAction`, but I define a small custom class `GroupAction` with just two axioms: compatibility with multiplication and the identity action. This keeps the development minimal and makes it easy to map Lean lines to textbook lines. The cost is that I cannot use existing `MulAction` lemmas, but the benefit is transparency.

2.2. What Lean is checking. Lean treats each statement as a typed object. When I define `sigmaPerm`, Lean checks that the inverse function is a two-sided inverse. When I define `phi`, Lean checks that its image lies in the group `Equiv.Perm X` and that multiplication is permutation composition. This explicit checking prevents hidden gaps that can slip into informal proofs.

3. Definitions

This section introduces the group action class and the standing assumptions used throughout the file.

3.1. Lean code. I first declare a minimal class for group actions:

```
^^Iclass GroupAction (G : Type*) [Monoid G] (X : Type*) where
^^I-- The action map: apply g to x.
^^Iact : G → X → X
^^I-- Compatibility with multiplication in G.
^^Iga_mul : ∀ g₁ g₂ x, act (g₁ * g₂) x = act g₁ (act g₂ x)
^^I-- The identity element acts as the identity function.
```

```
^^Iga_one : ∀ x, act 1 x = x
```

Note that I use `Monoid` here because the definition of a group action does not require inverses. Later results about permutation representations and stabilizers require a `Group`, so the subsequent sections assume `[Group G]`.
Throughout the file I work with a group G acting on a type X:

```
^^I-- Fix a group and an action instance.
^^Ivariable {G : Type*} [Group G] {X : Type*} [GroupAction G X]
```

4. Concrete Examples of Group Actions

This section presents six concrete instances of the `GroupAction` class, illustrating how the abstract definition applies to familiar mathematical structures.

4.1. Symmetric group acting on a set. Let X be any type. The symmetric group $\mathrm{Sym}(X)$ acts on X by evaluation: for $\sigma \in \mathrm{Sym}(X)$ and $x \in X$, define $\sigma \cdot x := \sigma(x)$.

```
^^Iinstance permGroupAction {X : Type*} : GroupAction (Equiv.Perm X) X :=
^^I{ act := fun g x => g x
^^Iga_mul := by
^^Iintro g1 g2 x
^^Irfl
^^Iga_one := by
^^Iintro x
^^Irfl }
```

The axiom proofs are immediate by reflexivity: composition of permutations and function evaluation commute definitionally in Lean.

4.2. Group acting on itself by left multiplication. Every group G acts on itself by left multiplication: $g_1 \cdot g_2 := g_1 * g_2$.

```
^^Iinstance groupAsGSet {G : Type*} [Group G] : GroupAction G G :=
^^I{ act := fun g1 g2 => g1 * g2
^^Iga_mul := by
^^Iintro g1 g2 g3
^^Irw [mul_assoc]
^^Iga_one := by
^^Iintro g
^^Irw [one_mul] }
```

The `ga_mul` axiom follows from associativity of group multiplication, and `ga_one` from the identity axiom.

4.3. Subgroup acting on the group. A subgroup $H \leq G$ acts on G by left multiplication. Elements of H are coerced to elements of G using `(h : G)`.

```
^^Iinstance subgroupAsGSet {G : Type*} [Group G] (H : Subgroup G) :
↪  GroupAction H G :=
^^I{ act := fun h g => (h : G) * g
^^Iga_mul := by
^^Iintros
^^Isimp [mul_assoc]
^^Iga_one := by
^^Iintros
^^Isimp }
```

The `simp` tactic handles the coercion and applies associativity and identity axioms.

4.4. Conjugation action of a subgroup on itself. A subgroup H acts on itself by conjugation: $h_1 \cdot h_2 := h_1 h_2 h_1^{-1}$.

```
^^Iinstance subgroupAsGSet_conjugation {G : Type*} [Group G] (H : Subgroup
↪  G) : GroupAction H H :=
^^I{ act := fun h g => h * g * h⁻¹
^^Iga_mul := by
^^Iintro g₁ g₂ g₃
^^Isimp
^^Irw [← mul_assoc g₁]
^^Irw [mul_assoc g₁ g₂]
^^Irw [← mul_assoc]
^^Iga_one := by
^^Iintros
^^Isimp }
```

The `ga_mul` proof rearranges terms using associativity to show $(g_1 g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$ under conjugation.

4.5. Scalar action on complex vector spaces. The multiplicative group $\mathbb{C}^\times$ of nonzero complex numbers acts on $\mathbb{C}^n$ by scalar multiplication.

```
^^Iinstance vectorSpaceAsCStarSet (n : ℕ) : GroupAction (ℂˣ) (Fin n → ℂ) :=
^^I{ act := fun r v => fun i => (r : ℂ) * v i
^^Iga_mul := by
^^Iintros r1 r2 v
^^Iext i
^^Isimp [mul_assoc]
^^Iga_one := by
^^Iintro v
^^Iext i
^^Isimp }
```

Here `ℂˣ` denotes the units of $\mathbb{C}$ (invertible elements), and `Fin n → ℂ` represents functions from $\{0, \ldots, n-1\}$ to $\mathbb{C}$ (i.e., n-dimensional complex vectors). The `ext` tactic proves function equality pointwise.

4.6. Scalar action on real vector spaces. Similarly, $\mathbb{R}^\times$ acts on $\mathbb{R}^n$ by scalar multiplication.

```
^^Iinstance vectorSpaceAsRStarSet (n : ℕ) : GroupAction (ℝˣ) (Fin n → ℝ) :=
^^I{ act := fun r v => fun i => (r : ℝ) * v i
^^Iga_mul := by
^^Iintros r1 r2 v
^^Iext i
^^Isimp [mul_assoc]
^^Iga_one := by
^^Iintro v
^^Iext i
^^Isimp }
```

The structure is identical to the complex case, with $\mathbb{R}$ replacing $\mathbb{C}$.

5. Permutation Representation

The core construction is the map $\sigma_g : X \to X$ given by $\sigma_g(x) = g \cdot x$. The inverse of σ_g is $\sigma_{g^{-1}}$, so σ_g is a permutation. This yields the permutation representation $\phi : G \to \mathrm{Sym}(X)$.

5.1. Proof sketch (informal). To show that σ_g is a bijection, compute $\sigma_{g^{-1}}(\sigma_g(x)) = (g^{-1}g) \cdot x = 1 \cdot x = x$, and similarly $\sigma_g(\sigma_{g^{-1}}(x)) = x$. The representation is defined by $\phi(g) = \sigma_g$, and multiplicativity follows from the action axiom:

$$\phi(g_1 g_2)(x) = (g_1 g_2) \cdot x = g_1 \cdot (g_2 \cdot x) = (\phi(g_1)\phi(g_2))(x).$$

5.2. Lean code. First, define the underlying action map:

```
^^I/-- The action map `sigma g : X → X`
^^Igiven by `x ↦ g • x`. -/
^^Idef sigma (g : G) : X → X :=
^^I-- We use the action directly.
^^Ifun x => GroupAction.act g x
```

Then package it as a permutation using the inverse action:

```
^^I/-- The permutation of `X` induced by `g`,
^^Iwith inverse given by `g⁻¹`. -/
^^Idef sigmaPerm (g : G) : Equiv.Perm X := by
^^Irefine
^^I^^I{ toFun := sigma g
^^I^^I^^IinvFun := sigma g⁻¹
^^I^^I^^Ileft_inv := ?_
^^I^^I^^Iright_inv := ?_ }
^^I· intro x
^^Icalc
^^I-- Apply the action axiom, then cancel g⁻¹ * g.
^^IGroupAction.act g⁻¹ (GroupAction.act g x) =
^^IGroupAction.act (g⁻¹ * g) x := by
^^Isimpa using (GroupAction.ga_mul g⁻¹ g x).symm
^^I_ = GroupAction.act (1 : G) x := by simp
```

```
^^I_ = x := GroupAction.ga_one x
^^I· intro x
^^Icalc
^^I-- The symmetric calculation for g * g⁻¹.
^^IGroupAction.act g (GroupAction.act g⁻¹ x) =
^^IGroupAction.act (g * g⁻¹) x := by
^^Isimpa using (GroupAction.ga_mul g g⁻¹ x).symm
^^I_ = GroupAction.act (1 : G) x := by simp
^^I_ = x := GroupAction.ga_one x
```

Define the representation and its basic properties:

```
^^I/-- The permutation representation `phi : G → Equiv.Perm X`
^^Iinduced by the action. -/
^^Idef phi (g : G) : Equiv.Perm X :=
^^IsigmaPerm g

^^I-- `phi` agrees with the action on elements.
^^Ilemma phi_apply (g : G) (x : X) : phi g x = GroupAction.act g x := rfl

^^Ilemma phi_mul (g₁ g₂ : G) : phi (g₁ * g₂) = phi g₁ * phi g₂ := by
^^I-- Reduce equality of permutations to pointwise equality.
^^Iapply Equiv.ext
^^Iintro x
^^Icalc
^^Iphi (g₁ * g₂) x = GroupAction.act (g₁ * g₂) x := rfl
^^I_ = GroupAction.act g₁ (GroupAction.act g₂ x) := GroupAction.ga_mul g₁
 ↪  g₂ x
^^I_ = (phi g₁ * phi g₂) x := rfl

^^Ilemma phi_one : phi (1 : G) = (1 : Equiv.Perm X) := by
^^Iapply Equiv.ext
^^Iintro x
^^Icalc
^^Iphi (1 : G) x = GroupAction.act (1 : G) x := rfl
^^I_ = x := GroupAction.ga_one x
^^I_ = (1 : Equiv.Perm X) x := by simp [Equiv.Perm.one_apply]
```

Finally, package the existence statement:

```
^^I-- Existence of the permutation representation with the required
 ↪  properties.
^^Itheorem group_action_to_perm_representation :
^^I∃ (ψ : G → Equiv.Perm X),
^^I(∀ g x, ψ g x = GroupAction.act g x) ∧
^^I(∀ g₁ g₂, ψ (g₁ * g₂) = ψ g₁ * ψ g₂) ∧
^^I(ψ 1 = 1) := by
^^Iexact ⟨phi, ⟨phi_apply, ⟨phi_mul, phi_one⟩⟩⟩
```

6. Stabilizers

For a fixed $x \in X$, the stabilizer is the subset

$$G_x = \{g \in G \mid g \cdot x = x\}.$$

In Lean this is first defined as a set, then shown to be the carrier of a subgroup, and finally packaged as a `Subgroup`.

6.1. Proof sketch (informal). The identity element fixes every x, so $1 \in G_x$. If g_1 and g_2 fix x, then $(g_1 g_2) \cdot x = g_1 \cdot (g_2 \cdot x) = g_1 \cdot x = x$, so $g_1 g_2 \in G_x$. If g fixes x, then $g^{-1} \cdot x = g^{-1} \cdot (g \cdot x) = (g^{-1} g) \cdot x = x$, so $g^{-1} \in G_x$.

6.2. Lean code. Start from the set definition:

```
^^I/-- The stabilizer set
^^I`G_x = { g ∈ G | g • x = x }`. -/
^^Idef stabilizerSet (x : X) : Set G :=
^^I-- Membership means g fixes x.
^^I^^I{ g : G | GroupAction.act g x = x }
```

Package it as a subgroup by checking closure:

```
^^I/-- The stabilizer `G_x` as a subgroup of `G`. -/
^^Idef stabilizer (x : X) : Subgroup G := by
^^Iexact
^^I^^I{ carrier := stabilizerSet (G := G) (X := X) x
^^I^^I^^Ione_mem' := by
^^I^^I^^I-- The identity fixes every x.
^^I^^I^^Isimp [stabilizerSet, GroupAction.ga_one x]
^^I^^I^^Imul_mem' := by
^^I^^I^^Iintro g₁ g₂ hg₁ hg₂
^^I^^I^^Icalc
^^I^^I^^I-- Closure under multiplication uses the action axiom.
^^I^^I^^IGroupAction.act (g₁ * g₂) x = GroupAction.act g₁ (GroupAction.act
 ↪  g₂ x) := by
^^I^^I^^Isimpa using (GroupAction.ga_mul g₁ g₂ x)
^^I^^I^^I_ = GroupAction.act g₁ x := by
^^I^^I^^Irw [hg₂]
^^I^^I^^I_ = x := hg₁
^^I^^I^^Iinv_mem' := by
^^I^^I^^Iintro g hg
^^I^^I^^Icalc
^^I^^I^^I-- If g fixes x then g⁻¹ fixes x.
^^I^^I^^IGroupAction.act g⁻¹ x = GroupAction.act g⁻¹ (GroupAction.act g x)
 ↪  := by
^^I^^I^^Irw [hg]
^^I^^I^^I_ = GroupAction.act (g⁻¹ * g) x := by
^^I^^I^^Isimpa using (GroupAction.ga_mul g⁻¹ g x).symm
^^I^^I^^I_ = GroupAction.act (1 : G) x := by simp
^^I^^I^^I_ = x := GroupAction.ga_one x }
```

Then show the set is the carrier of a subgroup:

```
^^I/-- The stabilizer set is the carrier of a subgroup of `G`. -/
^^Itheorem stabilizer_set_is_subgroup (x : X) :
^^I∃ H : Subgroup G, (H : Set G) = stabilizerSet (G := G) (X := X) x := by
^^Irefine ⟨stabilizer (G := G) (X := X) x, rfl⟩
```

7. Main Theorem

The Lean theorem `group_action_to_perm_representation` packages the permutation representation together with its key properties. The proof uses the lemmas from the previous section: `phi_apply`, `phi_mul`, and `phi_one`.

8. What Lean Guarantees

Formalizing these constructions in Lean provides guarantees that informal proofs cannot match. When Lean accepts a definition or theorem, it has verified:

- **Types are correct.** The function `phi : G → Equiv.Perm X` has the claimed type. Lean checks that `sigmaPerm g` is actually a permutation (a bijection), not just a function. If we mistakenly defined a non-bijective map, Lean would reject it.
- **No hidden assumptions.** Every lemma explicitly lists its hypotheses. If a proof uses associativity, the code must invoke `mul_assoc` or the `ga_mul` axiom. There are no "clearly" or "it follows that" steps that skip justification.
- **Axioms match definitions.** The `GroupAction` class requires `ga_mul` and `ga_one`. If we omitted `ga_one`, the proof of `phi_one` would fail because Lean would have no way to show $1 \cdot x = x$.
- **Subgroup structure is verified.** When we package the stabilizer as a `Subgroup`, Lean checks that we provided proofs of closure under multiplication, inverses, and identity. The type system prevents us from claiming "the stabilizer is a subgroup" without proving it.

These checks prevent common errors: mistaken claims about injectivity, forgotten hypotheses, and gaps in subgroup proofs. An external examiner can trust that the Lean code genuinely establishes the mathematical claims, not just that it compiles.

9. Reflection

Writing the proofs in Lean forced me to make every step explicit. In particular:

- I had to specify exactly where associativity of the action is used.
- I had to show explicitly that $\sigma_{g^{-1}}$ is an inverse.
- I had to unfold the definition of permutation multiplication.

The resulting code is not shorter than a textbook proof, but it is more precise. An external examiner can read the surrounding explanations and then see that each Lean line matches a specific mathematical claim.
I estimate the formalisation took over fifteen hours, mostly due to understanding Lean's typeclass resolution and the mechanics of equivalences and permutations in mathlib. The final development is compact but captures the standard argument for the permutation representation and the stabilizer subgroup.

9.1. Specific challenges encountered. Several technical hurdles emerged during the formalization:

- **Typeclass resolution:** Ensuring Lean could automatically find the `GroupAction` instance in complex contexts required careful use of explicit type annotations (e.g., `(G := G)`).

- **Equivalence mechanics:** Constructing `sigmaPerm` as an `Equiv.Perm X` required providing both `left_inv` and `right_inv` proofs, which meant carefully applying the `ga_mul` axiom in both directions.
- **Coercions:** Working with subgroups required understanding how Lean coerces elements of type `H` (a subgroup) to type `G` (the ambient group) using `(h : G)`.
- **Function extensionality:** Proving equality of permutations required `Equiv.ext`, and equality of vector-valued functions required `ext i`, which were not immediately obvious.

9.2. Potential extensions. This development could be extended in several directions:

- **Orbit-stabilizer theorem:** Formalize the relationship $|G| = |\mathrm{Orb}(x)| \cdot |G_x|$ for finite groups, which requires defining orbits and proving Lagrange's theorem.
- **Burnside's lemma:** Count orbits using the formula involving fixed points.
- **Actions on cosets:** Show that a group acts on the left cosets of a subgroup, yielding the standard permutation representation.
- **Cayley's theorem:** Specialize the permutation representation to the case where G acts on itself, proving every group embeds in a symmetric group.

10. AI Assistance Statement

I used AI assistance to help draft Lean proofs and improve code readability. All AI-generated content was reviewed and verified by me to ensure mathematical accuracy.

11. References

John B. Fraleigh, Victor J. Katz, *A First Course in Abstract Algebra*, Addison–Wesley, 2003, Section 16 (Group Actions).