Group Theory in Lean

Mitchell Rowett

Interactive Theorem Proving

Contents

Abstract 1

1 Introduction 2

2 Lean Theorem Prover 2

2.1 Type Theory 2

2.2 Elaboration 3

2.3 Type Class Inference 3

2.4 Group Construction 4

3 Group Theory 5

3.1 Subgroups 5

3.2 Homomorphisms 5

4 Subgroups 6

5 Homomorphisms 6

6 Cosets 7

7 Quotient Groups 8

7.1 Setoids 8

7.2 Quotients in Lean 9

7.3 Construction of Quotient Groups 9

8 First Isomorphism Theorem 10

8.1 Images and Isomorphisms 10

8.2 The Theorem 10

8.3 to_fun 11

8.4 inv_fun 11

8.5 Proofs 12

Abstract

In this report, we describe a formalisation of elementary group theory in the proof assistant Lean. We begin with an introduction to interactive theorem proving, and to the Lean Theorem Prover in particular. We then detail the construction of this formalisation in Lean, culminating in a proof of the first isomorphism theorem.

1 Introduction

Computational formal verification of mathematical theorems comes in two main forms. Automated theorem proving focuses on proving assertions fully automatically, with little or no input from a user. Interactive theorem proving involves a user actively guiding the proof, and builds mathematical structures from a suitable axiomatic framework.

Lean is primarily an interactive theorem prover, with a variety of automated tools. It encodes a formal mathematical language, based on dependent type theory, as well as a method for checking the correctness of terms written in this language.

2 Lean Theorem Prover

Lean is a proof assistant which encodes a variety of dependent type theory (described below) and provides a method for checking derivations made in that encoding. Lean also provides many features which aid the user in defining objects and proving theorems, the most relevant of which are discussed later in this section.

2.1 Type Theory

A type theory is a formal system in which every term has a type, and operations are restricted to terms of a certain type. It is built by specifying judgments and rules. The basic judgment in type theory is of the form $a : A$, which is read as the statement 'a is a term of type A'. Further, operations can be defined with types. The judgment $f : A \to B$ is the statement that f has the type of a function with domain A and co-domain B.

Type theory allows the construction of further objects, given prior judgments. For example, the expression

```
4 * (5 + 7) : nat
```

is a type judgment states that $4 * (5 + 7)$ is of type `nat`, which is true because 4, 5, and 7 are of type `nat`, while $*$ and $+$ are operations on natural numbers.

The simple type theory described above is limited in that the only types which exist are the types built into the language and those which can be constructed using $\to$ (i.e. functions from one type to another). Simple type systems generally have other simple constructions as well – for example, a conjuction operator which gives types of the form `A ∧ B`.

There are many types which cannot be described using this system – for example, a list of terms where all the terms are of type `A`, expressed as the type `list A`. The construction of this type clearly depends on the type `A`, hence a type theory where constructions of this form are possible is called a dependent type theory.

Type systems are commonly used in programming languages to check the syntactic correctness of code. For example, if `x` is given as an argument to a function which takes an integer as an argument, then a type system as described above can be used to check whether x indeed has the type integer, and so can provide a compile-time check of the syntactic correctness of the program.

Proof assistants can use the same method of type checking to ensure the correctness of a mathematical proof of some theorem, by relying on the Curry-Howard isomorphism. The isomorphism simply amounts to a correspondence between two possible readings of the judgment

```
p : A
```

– p is a term (variable, function output) of the data type A
– p is a proof of the hypothesis A

To generalise, the former is how a programmer would understand the judgment, while the latter is the point of view of a proof theorist. Interactive theorem provers use both, in which a term (proof) of the correct type (hypothesis) is sought interactively by both the user and the system. In other words, a type checking algorithm can be used to check whether p is a correct proof of the hypothesis A, as this will be the case if and only if p is a correct term of the type A.

In this manner, Lean encodes a dependent type theory called the Calculus of Inductive Constructions (CIC).

2.2 Elaboration

One issue proof assistants run into is that building more complex structures can require a lot of details. In 'pen-and-paper' mathematics, these details are omitted as they can be easily inferred.

To facilitate this, Lean has an elaborator, which allows the user to leave arguments implicit by putting them in curly brackets when defining a function. The elaborator will then convert this partially specified expression into a fully specified, type correct term. For example, Lean defines the identity function as

```
def id {α : Type} (a : α) : α := a
```

Unpacking this definition will show most of the essential lean syntax. The def keyword creates a function called id. The next two terms, {α : Type} and (a : α), are arguments to the function – α is a Type, while a is a term of type α. The syntax : α means that the result of the function will be a term of type α, and := a is the result of the function – given a type α and a term of that type a, return a.

Note, however, the difference between the syntax of the two arguments. The use of curly brackets allows the user to write id a rather than id α a. The elaborator can infer the type required.

The elaborator also supports overloading. For example, we later use the following: let a and b be elements of a group G, and S a subgroup of G. Then a*b is used to denote group multiplication, while a * S is used to denote the left coset of S by a. The elaborator also inserts coercions (for example, from nat to int) where necessary.

2.3 Type Class Inference

Another way in which the elaborator allows details to be omitted is through type class inference, which works as follows. First, a structure is marked as a class. Then, an instance of this class is declared, either by adding it as a hypothesis or by proving that it must be present due to existing hypotheses. Finally, in any definition/theorem/lemma in which this structure is used, we can mark the argument as implicit using square brackets rather than curly brackets, which informs the elaborator that these arguments should be inferred by the type class mechanism.

A relevant example of type class inference is the construction of the group structure.

2.4 Group Construction

Without any kind of implicit arguments, the statement `∀ a b : α, a * b : α` would be insufficient on its own. We would need to give as an argument to `*` the fact that multiplication on terms of type α is actually possible. However, using type class inference we can get around this:

```
class has_mul (α : Type u) := (mul : α → α → α)
```

This means that a proof or hypothesis `p : has_mul α` is the statement that α is a type with multiplication, and so we can write (for example) `∀ a b : α, a * b : α`. Further, since it is a class, we do not have to explicitly tell Lean that this is possible. So long as `p` is in the context, we can write `a * b`, and the elaborator will use type class inference to check that multiplication on terms of type α is possible.

Type class inference can also use the information that one type marked as a class inherits from another type marked as a class. For example:

```
class semigroup (α : Type u) extends has_mul α :=
    (mul_assoc : ∀ a b c : α, a * b * c = a * (b * c))
```

Since `semigroup` inherits from `has_mul`, we can think of `semigroup` as implicitly having the field `(mul : α → α → α)`. Given `[h : semigroup α]` in the context, we can write `a * b` and Lean will recognise that `semigroup` inherits from `has_mul`, and so can use type class inference to check that multiplication on terms of type α is possible.

We will now quickly show the remaining steps required to create a group structure. First, it is necessary for the type to have an identity element:

```
class has_one (α : Type u) := (one : α)
```

This allows us to create the monoid structure:

```
class monoid (α : Type u) extends semigroup α, has_one α :=
    (one_mul : ∀ a : α, 1 * a = a) (mul_one : ∀ a : α, a * 1 = a)
```

It is also necessary for every term of the type to have an inverse:

```
class has_inv (α : Type u) := (inv : α → α)
```

Finally we can create the group structure:

```
class group (α : Type u) extends monoid α, has_inv α :=
    (mul_left_inv : ∀ a : α, a⁻¹ * a = 1)
```

Writing these all separately is a lot more powerful than simply defining a group in its entirety. It allows us, for example, to prove theorems about multiplication which do not rely on the existence of an inverse. Then these theorems can be used whenever a type has multiplication, not just when a type is a group.

We can think of the group structure as having all of these fields, since it is an extension of previous structures. To write them all out explicitly:

```
class group (G : Type u) :=
    (mul : G → G → G)
    (mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
    (one : G)
    (one_mul : ∀ a : G, 1 * a = a)
    (mul_one : ∀ a : G, a * 1 = a)
    (inv : G → G)
    (mul_left_inv : ∀ a : G, a⁻¹ * a = 1)
```

3 Group Theory

In this section we will review the elementary group theory[2] which will be formalised in this report.

Definition 1 (Group). *A group is a set G together with an operation $* : G \times G \to G$, which we will call multiplication, such that:*

- *G is closed under multiplication: for all a and b in G, $a * b$ in G*
- *Multiplication is associative: for all a, b, c, in G, $(a * b) * c = a * (b * c)$*
- *G contains an identity element 1: for all a in G, $1 * a = a$ and $a * 1 = a$*
- *Every element a of G has an inverse, an element b such that $a * b = 1$ and $b * a = 1$*

We denote the inverse of a as a^{-1}.

3.1 Subgroups

Definition 2 (Subgroup). *A subset H of a group G is a subgroup if it has the following properties:*

- *Closure: If a and b are in H, then $a * b$ is in H*
- *Identity: 1 is in H*
- *Inverses: If a is in H, then a^{-1} is in H*

Definition 3 (Normal). *A subgroup N of a group G is a normal subgroup if for every a in N and every g in G, $g * a * g^{-1}$ is in N.*

Definition 4 (Trivial Subgroup). *The set containing only 1 is a subgroup, and is called the trivial subgroup.*

Definition 5 (Center). *The center of a group G is the set of elements that commute with every element of G:*

$$\{z \in G \mid z * x = x * z \text{ for all } x \in G\}$$

Definition 6 (Left Coset). *If H is a subgroup of a group G and a is an element of G, then the left coset of H by a is*

$$a * H = \{g \in G \mid g = a * h \text{ for some } h \in H\}$$

We similarly define the right coset of H by a and denote it $H * a$.

3.2 Homomorphisms

Definition 7 (Homomorphism). *Homomorphism: Let G and G' be groups. A homomorphism $\phi : G \to G'$ is a map from G to G' such that for all a, b in G:*

$$\phi(a * b) = \phi(a) * \phi(b)$$

Definition 8 (Kernel). *Suppose G and G' are groups, and let $\phi : G \to G'$ be a homomorphism. The kernel of ϕ is the set of elements of G which are mapped to the identity in G'.*

$$\{a \in G \mid \phi(a) = 1\}$$

Definition 9 (Quotient Group). *Let G be a group, and N a normal subgroup. Then there is an equivalence relation on elements of G, given by $a \sim b$ if $a * b^{-1} \in N$. The equivalence class of b given by this relation is denoted $[b]$, and the set of equivalence classes is denoted G/N. G/N is a group, and is called the quotient of G by N.*

Definition 10 (Isomorphism). *A homomorphism is called an isomorphism if it has a two-sided inverse. Two groups are said to be isomorphic if there exists an isomorphism between them.*

Theorem 1 (First Isomorphism Theorem). *Let G and G' be groups, and let $\phi : G \to G'$ be a homomorphism. Let K be the kernel of ϕ. Then G/K is isomorphic to the image of ϕ.*

4 Subgroups

Lean allows us to declare variables at the beginning of sections, which are then used as arguments in every definition/theorem in which they are necessary. For the rest of the report, assume that the definitions use the following variables:

```
variables {G : Type u} {H : Type v}
```

Given the definition of a subgroup described in section 3, we define a subgroup as follows.

```
class subgroup [group G] (S : set G) : Prop :=
    (mul_mem : ∀ {a b}, a ∈ S → b ∈ S → a * b ∈ S)
    (one_mem : (1 : G) ∈ S)
    (inv_mem : ∀ {a}, a ∈ S → a⁻¹ ∈ S)
```

subgroup takes as its only explicit argument a set of G, with which it will try to find an instance of the fact that G is a group. subgroup S is of type Prop, as it is the proposition that S is a subgroup.

This distinction is important. subgroup does not take a subset and return a subgroup. It is simply the assertion that the set S is also a subgroup.

A term h : subgroup S is effectively the statement that the propositions mul_mem, one_mem, and inv_mem hold. Likewise, in order to prove subgroup S it is necessary to prove that all three of these propositions hold for S.

We can similarly define a normal subgroup, which extends the subgroup structure.

```
class normal_subgroup [group G] (S : set G) extends subgroup S : Prop :=
    (normal : ∀ n ∈ S, ∀ g : G, g * n * g⁻¹ ∈ S)
```

For example,

```
def trivial (G : Type u) [group G] : set G := {1}
```

We can define the set trivial, which it is then easy to prove is a normal subgroup.

```
instance trivial_in [group G] : normal_subgroup (trivial G) :=
    by refine {..}; simp
```

refine is a tactic (part of Lean's automation of theorem proving) which splits the goal of proving normal_subgroup (trivial G) into four goals: mul_mem, one_mem, inv_mem, and normal. simp is a powerful tactic which is used for automatically simplifying statements – in this case, it is sufficient to prove all four goals.

5 Homomorphisms

Given two types α and β, we can construct the type of functions between them, $\alpha \to \beta$. We can then define the assertion that a function is a homomorphism as follows.

```
class group_hom [group G] [group H] (f : G → H) : Prop :=
    (hom_mul : ∀ a b, f (a * b) = f a * f b)
```

This definition is a good example of the benefits of the elaborator. On the left side of the equality, a * b is multiplication of terms of type α, which is possible because the elaborator can determine that since α is a group, we have that has_mul α. On the right side of the equality, f a * f b is multiplication of terms of type β. Lean supports this overloading of the notation *, using type class inference to determine which operator to use.

We can also define certain structures based on the existence of homomorphisms.

```
def kernel [group G] [group H] (f : G → H) [group_hom f] : set G :=
    preimage f (trivial H)
```

`preimage` is already defined in Lean – it is the preimage of `trivial H`, where `f` is considered as a function on sets. We need to show that it is a normal subgroup of α, which we can do by showing the more general proposition that the preimage of a normal subgroup is a normal subgroup.

```
instance preimage_norm_in [group G] [group H] (f : G → H) [group_hom f]
(S : set H) [normal_subgroup S] : normal_subgroup (preimage f S) :=
        by refine {..}; simp [hom_mul f, one f, inv f]
```

The terms in square brackets after `simp` are additional lemmas which the `simp` tactic can use to simplify the goals.

We also prove some important lemmas to do with kernels.

Lemma 2 *Let G and G' be groups, and $f : G \to G'$ a homomorphism. Then for all $a, b \in G$, $f\ a = f\ b$ if and only if $a * b^{-1} \in \ker(f)$.*

```
lemma one_iff_ker_inv [group G] [group H] (f : G → H) [group_hom f]
(a b : G) : f a = f b ↔ f (a * b⁻¹) = 1
```

Lemma 3 *Let G and G' be groups, and $f : G \to G'$ a homomorphism. Then f is injective if and only if $\ker(f)$ is the trivial subgroup of G.*

```
lemma trivial_kernel_of_inj [group G] [group H] {f : G → H} [group_hom f]
(h : function.injective f) : kernel f = trivial G
```

The proofs of these lemmas are fairly simple, and the proofs in Lean closely follow the intuitive proofs.

6 Cosets

Given `[has_mul G] (a : G) (S : set G)`, we can construct the left and right cosets of `S` by `a`.

```
def lcoset [has_mul G] (a : G) (S : set G) : set G := image (λ x, a * x) S
def rcoset [has_mul G] (S : set G) (a : G) : set G := image (λ x, x * a) S
```

Here we introduce another powerful feature of Lean's dependent type theory – lambda expressions. The expression `(λ x, a * x)` is a function, with the arguments to the left of the comma and the result to the right. In this case, it is a function which takes an element `x` of the set `S` and returns `a * x`. The image of this function is clearly the left coset of `S` by `a`, as we require.

We can then introduce some notation to mirror the informal notation `aS`, which is commonly used to denote the left coset of `S` by `a`.

```
namespace coset_notation
    infix * := lcoset
    infix * := rcoset
end coset_notation
```

By overloading the notation `*`, we now enable ourselves to write `a * S` and `S * a` to denote the left coset and right coset respectively. Lean's type class inference is able to determine when `*` means multiplication, and when it means a coset.

In addition to several simple lemmas about cosets, we prove two main lemmas.

Lemma 4 *Let G be a group and S a subgroup. Then the relation $a \sim b \iff aS = bS$ is an equivalence relation.*

```
def lcoset_equiv (S : set G) (a b : G) := a * S = b * S
```

Equivalence relations are defined as expected in Lean. Given a relation `r : G → G → Prop`, and using the notation `a ≺ b := r a b`, we define:

```
def reflexive := ∀ x, x ≺ x

def symmetric := ∀ {x y}, x ≺ y → y ≺ x

def transitive := ∀ {x y z}, x ≺ y → y ≺ z → x ≺ z

def equivalence := reflexive r ∧ symmetric r ∧ transitive r
```

We also have a constructor `mk_equivalence`, which takes proofs of `reflexive r`, `symmetric r`, and `transitive r`, and creates a proof that `r` is an equivalence relation. With this in mind, we can provide a very simple proof:

```
lemma lcoset_equiv_rel (S : set G) : equivalence (lcoset_equiv S) :=
 mk_equivalence (lcoset_equiv S) (λ a, rfl) (λ a b, eq.symm) (λ a b c, eq.trans)
```

The simplicity of these proofs relies on the fact that cosets are defined as the image of multiplication, which allows Lean to use lemmas about multiplication in the proofs.

Lemma 5 *Let G be a group and S a subgroup of G. Then S is a normal subgroup if and only if for all $g \in S$, $gS = Sg$.*

```
theorem normal_iff_eq_cosets : normal_subgroup S ↔ ∀ g, g * S = S * g
```

7 Quotient Groups

Suppose we have a type α and an equivalence relation r on α. Then we can consider the ”quotient” $\alpha \ / \ r$, the set of equivalence classes of α modulo r. Given a function $f : \alpha \to \beta$, if we prove *for all* $a, b \in \alpha$, $a \sim b \implies f \ a = f \ b$ (in other words, that f respects the equivalence relation), then f ”lifts” to a function $f' : \alpha/\text{r}\!\rightarrow \beta$, defined on each equivalence class $[x]$ with $f' \ [x] = f \ x$.

The Calculus of Inductive Constructions that Lean is based on has no in-built way to define quotients, so they are added using constants. Before we discuss these constants, however, we must introduce setoids.

7.1 Setoids

A setoid is simply a set with an equivalence relation on it. In Lean, a setoid is defined as

```
class setoid (α : Sort u) :=
    (r : α → α → Prop)
    (iseqv : equivalence r)
```

7.2 Quotients in Lean

A constant in Lean is simply the statement that a certain identifier has a certain type. This allows to construct new types. Effectively, it is the addition of certain axioms into Learn.

The following constants are built into Lean:

```
constant quotient {α : Type u} (s : setoid α) : Type u

constant quotient.mk {α : Type u} [s : setoid α] (a : α) : quotient s

constant quotient.lift {α : Sort u} {β : Sort v} [s : setoid α] (f : α → β)
    : (∀ a b, a ∼ b → f a = f b) → quotient s → β

constant quotient.ind {α : Sort u} [s : setoid α] {β : quotient s → Prop}
    : (∀ a, β [a]) → ∀ q, β q

constant quotient.sound {α : Sort u} [s : setoid α] {a b : α}
    : a ∼ b → [a] = [b]
```

`quotient` constructs a new type given any setoid `s`. The axioms following it (`sound` in particular) make this new type exactly the quotient we would expect: the set of equivalence classes of the relation `setoid.r`.

`quotient.mk` maps α to `quotient s`. Given a term `a : α`, `quotient.mk a` is the term of type `quotient s` corresponding to the equivalence class of `a`.

If we are given any function `f : α → β`, and given a proof `h` that `f` respects the quotient, the function `lift f h : quotient s → β` is the corresponding function on `quotient s`.

`ind` is the statement that in order to prove a statement of the form

for all t in quotient s, p s

it is sufficient to prove

for all a in α*, p [a].*

Finally, `sound` is the axiom that ensures the quotient construction indeed acts how we expect it to – that if `a ∼ b`, then `[a] = [b]`. (The converse of this is encoded as a lemma called `quotient.exact`.)

7.3 Construction of Quotient Groups

To construct the quotient group in Lean given a particular normal subgroup, we must

1. Define an appropriate equivalence relation on the group
2. Use Lean's quotient constructor to build a quotient type
3. Prove that this quotient type is a group

We begin by defining an equivalence relation based on normal subgroups.

```
def norm_equiv [group G] (N : set G) (a b : G) := a * b⁻¹ ∈ N
```

It is not difficult to show that this is an equivalence relation – in fact, it is fairly simple to show that this is in fact the same equivalence relation as the coset relation described in section 6.

We can then define use this equivalence relation to define a setoid, and with that setoid we can give our definition of a quotient group.

```
def quotient_group (G) [group G] (N : set G) [normal_subgroup N] :=
    quotient (quotient_group.setoid N)
```

We also define an easier notation:

```
notation G ‘/‘ N := quotient_group G N
```

Finally, we prove

```
instance [group G] (N : set α) [normal_subgroup N] : group (G / N)
```

which involves proving each of the seven sub-propositions listed at the end of section 2. These can all be proven fairly simply by lifting the proof from the base group.

8 First Isomorphism Theorem

8.1 Images and Isomorphisms

Before we can prove the first isomorphism theorem, we first need to define what we mean by the image of a homomorphism, and what we mean by an isomorphism.

```
def image [group G] [group H] (f : G → H) : set H := image f univ
```

`univ` is simply the set of all terms of a particular type (in this case `G`), so it is clear that this definition fits with our common understanding of the image of a homomorphism. Terms of the type `image f` have the form

```
b ∧ h
```

where `b : H` and `h : ∃ a, f a = b` (a proof that `b` is in the image of `f`).

The definition of a group isomorphism is similarly simple; it is a function, its two-sided inverse, proofs that the two functions are two-sided inverses, and a proof that the original function is a homomorphism.

```
class group_isomorphism (G : Type u) (H : Type v) [group G] [group H] :=
    (to_fun    : G → H)
    (inv_fun   : H → G)
    (left_inv  : left_inverse inv_fun to_fun)
    (right_inv : right_inverse inv_fun to_fun)
    (hom_fun : group_hom to_fun)
```

We also give ourselves a more convenient notation for this.

```
infix ≅ := group_isomorphism
```

8.2 The Theorem

We are finally ready to give the type signature of the first isomorphism theorem.

```
noncomputable theorem first_isomorphism_theorem [group G] [group H]
(f : G → H) [group_hom f] : G / kernel f ≅ image f
```

The most interesting part of this type signature is the statement that it is noncomputable. As will be explained later, this is because the proof given in this report requires the axiom of choice!

8.3 to_fun

In informal mathematics, the function $g : G/\ker f \to \operatorname{im} f$ used in this isomorphism is as follows: we can think of each element of $G/\ker f$ as corresponding to a particular subset of G. These subsets of G are exactly the preimages of each element of $\operatorname{im} f$. Let g map the element of $G/\ker f$ corresponding to the preimage of b to b.

To construct this in Lean for a function `f : G → H`, we need a way to construct a function `f' : G / kernel f → H` such that `f' [a] = f a` for all `a : G`.

Fortunately, `quotient.lift` does exactly this, as described above. Note that `quotient.lift` also requires us to give a proof that `a ∼ b → f a = f b`. To save ourselves the necessity of proving this in the construction of the first isomorphism theorem, we create the function

```
def qgroup_lift [group G] [group H] {N : set G} [normal_subgroup N] (f : G → H)
  [group_hom f] (h : ∀ x ∈ N, f x = 1) (q : G / N) : H := quotient.lift f _ q
```

where the underscore is a proof of `a ∼ b → f a = f b`, given `h`.

By doing this, we make it so that in the context of the first isomorphism theorem, we only need to prove `∀ x ∈ N, f x = 1`, which is very easy to do.

We also need a way to take a function `f : G → H` and get a function `g : G → image f`. This can be done very simply. Recall that terms of type `image f` are most commonly expressed as `b ∧ h`, written in Lean as `⟨b, h⟩`.

```
def im_lift [group G] [group H] (f : G → H) [group_hom f] (c : G) : image f
:= ⟨f c, image_mem f c⟩
```

To recap: `qgroup_lift` takes a function with type `G → H` and gives a function with type `G / kernel f → H`. `im_lift` takes a function `f` with type $G \to H$ and returns a function with type `G → image f`.

To get our required function, which has type **G / kernel f → image f**, we can write

```
to_fun := qgroup_lift (im_lift f) _
```

where the underscore is a proof of `∀ x ∈ N, f x = 1` (as required by `qgroup_lift`).

8.4 inv_fun

In informal mathematics, the function $g^{-1} : \operatorname{im} f \to G/\ker f$ which is the inverse to the function described in the previous section is as follows: for each $b \in \operatorname{im} f$, choose some $a \in G$ such that $f\ a = b$. Let $g^{-1}(b) = [a]$.

This can be written very simply in Lean as

```
inv_fun := λ b, [classical.some b.property]
```

Recall from section 6 that this is a function which takes `b : image f` as an argument. `classical.some` does exactly what would be expected; given `b : image f`, it returns some `a` such that `f a = b`.[1]

`classical.some` is a direct application of the axiom of choice as implemented in Lean, which causes this theorem to be noncomputable.

The following statement is equivalent to the first isomorphism theorem as presented in this report: if $f : G \to G'$ is surjective, then $\bar{f} : G/\ker f \to G'$ is injective. This statement would

[1] Recall that since **b** has type **image f**, it is of the form ⟨**b', h**⟩. We can refer to these using **b.val := b'** and **b.property := h**.

certainly be possible to prove without the axiom of choice, and so it may be possible to prove the first isomorphism theorem without the axiom of choice.

However, we may run into the axiom of choice when proving that the two statements above are equivalent. Lean already contains a proof of this equivalence, but it is a noncomputable proof.

8.5 Proofs

The proofs of `left_inv` and `right_inv` are informally very simple; since the two functions are explicitly designed as inverses. The most complex part is proving a lemma that $f(a) = f(b) \iff [a] = [b]$, which in itself is not particularly difficult.

The Lean proofs are mostly simplifying definitions and applying the above lemma.

Similarly, the proof of `hom_fun` simply requires showing that the property of being a homomorphism lifts. This was already proven in the proof that the quotient group is a group.

References

[1] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. *The Lean Theorem Prover (system description).* Microsoft Research and Carnegie Mellon University. Research Showcase @ CMU. 2015.

[2] M. Artin. *Algebra, 2nd Edition.* Pearson. 2010.

[3] H. Geuvers. *Introduction to Type Theory.* Radboud University Nijmegen and Technical University Eindhoven. 2008.

[4] J. Avigad, L. de Moura, and S. Kong. *Theorem Proving in Lean.* 2018.