BEng Individual Project

Imperial College London

Department of Computing

# Group Cohomology in Lean

*Supervisor:*
Prof. Kevin Buzzard

*Author:*
Anca Ciobanu

*Second Marker:*
Prof. David Evans

July 10, 2019

**Abstract**

The Lean project is a new open source launched in 2013 by Leonardo de Moura at Microsoft Research Redmond that can be viewed as a programming language specialised in theorem proving. It is suited for formalising mathematical definitions and theorems, which can help bridge the gap between the abstract aspect of mathematics and the practical part of informatics. My project focuses on using Lean to formalise some essential notions of group cohomology, such as the 0th and 1st cohomology groups, as well as the long exact sequence. This can be seen as a performance test for the new theorem prover, but most importantly as an addition to the open source, as group cohomology has yet to be formalised in Lean.

# Contents

# Chapter 1

# Introduction

## 1.1 About theorem provers

Recent years have shown a significant increase in the interest in making mathematical definitions and proofs of theorems not only verifiable, but also readable by computers. The motivation behind this refers to the fact that even though technology has evolved tremendously and rapidly, mathematics in universities is still blackboard based. The sensible solution to this seems to be a tool that not only allows you to define mathematical concepts, but also to come up with proofs or to verify an existing one – we call this a theorem prover.

Providing a proof represents the base for supporting a mathematical claim and most conventional proof methods can be reduced to a small set of axioms and rules in any of a number of foundational systems. With this reduction, there are two ways that a computer can help establish a claim: it can help find a proof in the first place, and it can help verify that a purported proof is correct.

On one hand, automated theorem proving (ATP) is a subfield of automated reasoning and mathematical logic dealing with the "finding" aspect. Along the years, many people have contributed to the development of this field, from Aristotle, the parent of formalised logic, to Frege introducing propositional calculus and modern predicate logic (1879), to Mojżesz Presburger who came up in 1929 with an algorithm that could determine if a given sentence of a certain kind in the natural numbers was true or false. This topic has become more and more tempting and challenging in recent years due to the fact that automated reasoning over mathematical proof was a major impetus for the development of the computer science.

On the other hand, there is the verification part belonging to interactive theorem proving, that involves the use of computational proof assistants to verify that mathematical claims are correct, or to verify that hardware and software designs meet their formal specifications. The requirement here is that every claim is supported by a proof (every step has to be justified by appealing to prior definitions and theorems).

## 1.2 Lean as a solution

The Lean theorem prover aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. The goal is to support both mathematical reasoning and reasoning about complex systems, and to verify claims in both domains.

Lean seems to have indeed made a major change in how software could be used to formalise mathematical concepts, as it was developed taking into consideration both the key parts and flaws of famous theorem provers, such as Coq and Isabelle. One of the drawbacks of most of the previous systems is the fact that they use simple type theory, which limits the flexibility of handling abstract concepts. In addition to that, some of them also failed to provide a fairly natural mathematical language. These issues have been solved by the logical system that Lean is based on, a version of dependent type theory powerful enough to prove almost any mathematical theorem and expressive enough to do it in a natural way.

As Lean is still fairly new, thus still developing, there are only few (basic) parts of mathematics that have been touched, which encourages people to come up with their own implementation. This is one of the reasons behind me choosing to approach group cohomology as a field to formalise in Lean; not only is it a captivating part of mathematics, but it also missed from the maths library from the platform.

## 1.3 Achievements

The main goal of this project was to formalise mathematical definitions and theorems of group cohomology in Lean, which seemed quite intimidating at the beginning, as group cohomology is a part of mathematics that I have never approached before. The notion of a cohomology group has been generalised to an abstract $H^n(G, M)$, where $G$ is a group, $M$ is a module and $n$ is a natural number. This project presents the definitions of both 0th and 1st cohomology groups, $H^0(G, M)$ and $H^1(G, M)$, as well as the proofs of the essential properties that they come with (for example, the proof that these cohomology groups are also abelian groups).

The most challenging part was reasoning about the long exact sequence including $H^0(G, M)$ and $H^1(G, M)$ that is obtained when starting with an exact sequence of $G$-modules. Thus, the most impoortant achievement of this project is summarised in the lemma below:

If there is an exact sequence of $G$-modules

$$0 \to A \to B \to C \to 0$$

then there is the long exact sequence

$$0 \to H^0(G, A) \to H^0(G, B) \to H^0(G, C) \to \\ H^1(G, A) \to H^1(G, B) \to H^1(G, C).$$

In the upcoming chapters I will introduce some of the essential notions in group theory (Chapter2), that will help for a better understanding of the definitions and theorems I have chosen to present from group cohomology; as this is a vast domain, I will talk only about the 0th and 1st cohomology classes (Chapter 4). Chapter 3 will be dedicated to the basic aspects and tactics in Lean, which will appear later on in some proofs (Chapter 5) that will be thoroughly explained. Both interesting features and challenges of Lean will later be discussed (Chapter 6), as well as potential extension of the project and future work in Lean in general (Chapter 7).

# Chapter 2

# Mathematical Background

In order to define group cohomology, we first have to introduce basic definitions and properties of groups, as well as more advanced notions, such as homomorphisms and quotients.

## 2.1 Basic group theory

**Definition** (Group): A **group** is a pair $(G, \circ)$, where $G$ is a set and $\circ : G \times G \to G$, $(g,h) \mapsto g \circ h$ is a binary operation on $G$, called the group product, satisfying the following group axioms:

1. $g,\, h \in G \to g \circ h \in G$ (closure);
2. $\forall f, g, h \in G$: $(g \circ h) \circ f = g \circ (h \circ f)$ (associativity);
3. $\exists e \in G$ such that $\forall g \in G :\ e \circ g = g \circ e = g$ (identity);
4. $\forall g \in G\ \exists g^{-1} \in G$: $g \circ g^{-1} = g^{-1} \circ g = e$ (inverse).

**Definition** (Abelian group): A group $(G, \circ)$ is called **abelian** (or commutative) if $g \circ h = h \circ g$ for all $g, h \in G$.

**Definition** (Subgroup): Let $(G, \circ)$ be a group and $S \subseteq G$. Then $S$ is a **subgroup** of $G$, written $S \leq G$ if and only if $(S, \circ)$ is a group, i.e.:

1. $e \in S$;
2. $s, t \in S \Rightarrow s \circ t \in S$;
3. $s \in S \Rightarrow s^{-1} \in S$.

**Proposition**: Let $(G, \circ)$ be a group. Then:

- the identity element $e$ of $G$ is unique;
- every element $g \in G$ has a unique inverse $g^{-1}$;
- $(g^{-1})^{-1} =$ g for all $g \in G$;
- $(g \circ h)^{-1} = h^{-1} \circ g^{-1}$;

- for any $g_1, ..., g_n \in G$ the value of $g_1 \circ ... \circ g_n$ is independent of how the expression is bracketed (the generalised associative law).

**Definition** (Cosets): Let $S$ be a subgroup of $G$ and $g \in G$. Then the sets $Sg = \{sg \mid s \in S\}$ and $gS = \{gs \mid s \in S\}$ are called respectively a **right coset** and a **left coset** of $S$ in $G$. Any element of a coset is called a **representative** for the coset; in general, $Sg$ and $gS$ are different sets.

## 2.2 Normal subgroups and homomorphisms

**Definition** (Normality): A subgroup $N$ of $G$ is said to be **normal**, denoted by $N \lhd G$, if for every $g \in G$ we have $Ng = gN$, or equivalently, if $g^{-1}Ng = N$.

**Definition** (Homomorphism): Let $(G, \circ)$ and $(H, \triangle)$ be two groups. Then a **homomorphism** from $G$ to $H$ is a mapping $\psi : G \to H, g \mapsto \psi(g)$ satisfying the following condition for all $g_1, g_2 \in G$:

$$\psi(g_1 \circ g_2) = \psi(g_1) \triangle \psi(g_2)$$

**Definition** (Image): Let $\psi : G \to H$ be a homomorphism of groups. Then the **image** of $\psi$ is $\text{Im}(\psi) := \{h \in H \mid h = \psi(g)\}$ for some $g \in G$.

**Definition** (Kernel): Let $\psi : G \to H$ be a homomorphism of groups. Then the **kernel** of $\psi$ is $\ker(\psi) := \{g \in G \mid \psi(g) = e_H\}$.

## 2.3 Quotient groups

**Definition** (Quotient group): Let $G$ be a group and $N$ a normal subgroup of $G$. Then the **quotient group** of $N$ in $G$, written $G/N$, is the set of all cosets of $N$ in $G$:

$$G/N = \{gN \mid g \in G\}$$

Without loss of generality, we will work with the left cosets. The elements of $G/N$ are the cosets of $N$ in $G$, where the group product is defined (for all $g, h \in G/N$) as:

$$(gN)(hN) = (gh)N$$

**Proposition**: $G/N$ forms a group.

1. As $gh \in G$, it follows that $(gh)N$ is a left coset, thus $G/N$ is closed.

2. $gN(hNkN) = gN(hkN) = g(hk)N = (gh)kN = (gh)NkN =(gNhN)kN$, so $G/N$ is associative.

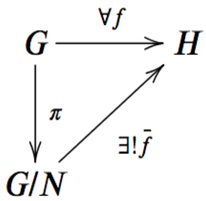
3. The left coset $eN = N$ serves as the identity: $\forall g \in G$, $N(gN) = (eN)(gN)$ $= (eg)N = gN$; similarly, $(gN)N = gN$.

4. We have $gN^{-1} = g^{-1}N$: $(gN)(g^{-1}N) = (gg^{-1})N = eN = N$; similarly, $(g^{-1}N)(gN) = N$.

**Definition** (Quotient epimorphism): Let $G$ be a group, $N$ a normal subgroup of $G$, and $G/N$ the quotient group of $N$ in $G$. Then the **quotient group epimorphism** from $G$ to $G/N$, also known as the **projection**, is the mapping $\psi : G \to G/N$ defined as:

$$\psi(x) = xN, \forall x \in G$$

**Theorem** (Universal property of quotient group): Let $G$ and $H$ be groups, $N$ a normal subgroup of $G$, $\pi : G \to G/N$ the projection, and $f : G \to H$ a group homomorphism with $N \subset \ker f$. Then there exists a unique group homomorphism $\overline{f} : G/N \to H$ such that $f = \overline{f} \circ \pi$.



# Chapter 3

# Tutorial in Lean

This chapter provides an insight to the basic functionality of Lean and to its natural syntax. Let's have a look at one of the simplest examples:

```
example very_easy : true :=
begin
  exact trivial
end
```

In the code above, the name of the example is `very_easy`, and its aim (written after the first use of ":") is to prove truthfulness. The proof starts after the use of ":=", and we enter tactic mode inside the `begin`(...)`end` statement. The proof is minor, and Lean provides us with the notion of triviality. The proof is finished in one line only, using the `exact` tactic, and the built in `trivial` notion.

## 3.1 Tactics for proving theorems

Tactics are commands or instructions that help in constructing proofs. For example, the steps in a mathematical proof might include applying a lemma (for which we have the corresponding instruction in Lean `apply`), simplifying some calculations (for which we could use `simp`) or introducing new variables (using the tactic `intro`, followed by an appropriate name for the variable that has not been used before in that scope). One can already tell that tactics in Lean have intuitive names and follow the usual steps of a mathematical proof.

When wanting to prove a theorem in Lean, the user works towards the final goal (which is the theorem itself at the beginning) through intermediate steps (as one would do in a mathematical proof) that change the goal along the way. The new goal at every step is displayed in a separate window by Lean when clicking after the last command, which provides the user with a clear image of the current state: all the givens and the desired outcome.

The most commonly found tactics throughout my project are the following:

- the `intro` tactic, used when the goal is of the form "$\forall x(statement)$" in order to introduce a new variable for which we want to prove that *statement* is true;

- the `cases` tactic, where the hypothesis is of the form "$\exists x(statement)$" and the type of $x$ is an inductive type;
- the `rw` (rewrite) tactic, used for applying substitutions to goals, providing a convenient and efficient way of working with equality;
- the `show` (or `change`) tactic, which simply declares the type of the goal that is about to be solved, while remaining in tactic mode; this is extremely useful when turning complicated goals into readable ones.

However, more tactics are available in Lean that help in following almost any conventional mathematical proof. A summary of these is included in the table below:

| **Type of term** | **To introduce** | **To eliminate** |
|---|---|---|
| $P \to Q$ | intro | apply |
| $P \wedge Q$ | split | cases |
| $P \vee Q$ | left, right | cases |
| $\exists x : T, H$ | use | cases |
| $\forall x : T, H$ | intro | apply |
| $P \leftrightarrow Q$ | split | cases |
| $\neg P$ | intro | apply |

## 3.2 Examples

```
example (P Q : Prop) (HP : P) (HPQ : P → Q) : Q :=
 begin
  apply HPQ,
  exact HP
 end
```

In the example above we have as givens the type of the variables P and Q (Prop) and two hypotheses: we know $P$ and $P \to Q$, and our goal ( written after ":") is to prove $Q$. We start the proof by entering tactic mode and we do so inside the `begin`(...)`end` statement. As illustrated in the table above, when wanting to eliminate an implication we use function application, a tactic called `apply`. The reasoning behind it is that in order to prove our goal $Q$, because we have that $P \to Q$, it is sufficient to prove $P$, thus after the `apply HPQ` line the goal has changed from $Q$ to $P$. As our new goal, $P$, is exactly one of the hypotheses that we already know, we will accomplish this goal by simply using the syntax `exact HP`.

```
import data.real.basic
import tactic.norm_num
example (a b c : ℝ) (Ha : a = 2) (Hb : b = 4) :
a ^ 2 = b ∧  b - a > 0 :=
  begin
    split,
      rw Ha,
      rw Hb,
      norm_num,
    rw Hb,
```

```
  rw Ha,
  norm_num,
end
```

The second example illustrates how to deal with a goal involving simple calculations in the set of real numbers. To do so, we must first import the libraries including the notions of the reals, as well as the tactic that is commonly used in handling trivial equalities and inequalities, `norm_num`. In order to follow the steps of the proof, I will include the tactic state before and after each line of the code. Before writing anything in the proof, we start with the tactic state:

```
1 goal
a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ a ^ 2 = b ∧ b - a > 0
```

The `split` tactic transforms our goal into two separate goals, so we could handle each goal at a time.

```
2 goals
a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ a ^ 2 = b

a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ b - a > 0
```

The `rw` tactic is used here to substitute the numerical values of the variable into the equations, and then each goal is accomplished through the `norm_num` instruction that verifies simple equalities and inequalities in this case.

For example, when dealing with the first goal, we start by substituting the numerical value for $a$, thus obtaining:

```
2 goals
a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ 2 ^ 2 = b

a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ b - a > 0
```

We substitute the value for $b$ in the first goal by writing `rw b`, which changes the goal to `2 ^ 2 = 4`. This is a trivial arithmetic equality, which could be dealed with by using `norm_num`. After doing so, the first goal has been accomplished and thus the tactic state shows the remaining goal only:

```
1 goal
a b c : ℝ,
Ha : a = 2,
Hb : b = 4
⊢ b - a > 0
```

The second goal is solved by a similar approach. When all the goals have been achieved, the tactic state should display:

```
goals accomplished
```

# Chapter 4

# Definitions and Theorems of Group Cohomology

Throughout this chapter, I will introduce most of the mathematical concepts of group cohomology that have been approached in this project, along with some of their corresponding definitions in Lean.

## 4.1 Motivation

Cohomology theory is undoubtedly a powerful mathematical tool. Initially used in topology, it has now extended to a much wider area of interest. Group cohomology is now found in various fields of algebraic number theory, abstract algebra, homological algebra and algebraic topology. The key point of this domain is that we could make use of the **invariants of groups**, instead of handling complicated groups.

Galois theory is also connected to group cohomology; in fact, we can talk about Galois cohomology – the application of homological algebra to modules for Galois groups. Realising that the Galois cohomology of ideal class groups in algebraic number theory was one way to formulate class field theory came around 1950.

One interesting application is related to parametrising the rational points on the circle, which makes use of Hilbert's Theorem 90:

**Theorem**: Let $G$ be a Galois group of a finite extension K of a field k. Then:
$$H^1(G, K^*) = 0$$

**Corollary**: If $G$ is furthermore cyclic with generator $s$, then an element $a \in K$ has norm one if and only if it can be written as $\frac{t}{s(t)}$ for some $t \in K$.

Let $k = \mathbb{Q}$ and $K = \mathbb{Q}(i)$, $G = \mathbb{Z}/2$ generated by complex conjugation. Then by corollary we have that $x + iy \in K$ satisfies the equation $x^2 + y^2 = 1$ if and only if

$$x + iy = \frac{u+iv}{u-iv} = \frac{u^2-v^2}{u^2+v^2} + i\frac{2uv}{u^2+v^2}$$

This actually tells us that the rational points on the circle must be of the form $(\frac{u^2-v^2}{u^2+v^2}, \frac{2uv}{u^2+v^2})$.

## 4.2 Modules

In mathematics, given a group $G$, a $G$-module is an abelian group $M$ on which $G$ acts compatibly with the abelian group structure on $M$. Group cohomology provides an important set of tools for studying general $G$-modules.

**Definition** ($G$-module): Let $(G, *)$ be a group. A **left $G$-module** $M$ is an abelian group under addition with $\bullet : G \times M \to M$ a left action on $G$, satisfying the following, for all $g, h \in G$ and for all $m, n \in M$:

1. $1 \bullet m = m$;
2. $g \bullet (h \bullet m) = (g * h) \bullet m$;
3. $g \bullet (m + n) = g \bullet m + g \bullet n$).

The idea of a $G$-module can be expressed in Lean quite intuitively, as seen in the code below. The similarities between the mathematical way of expressing axioms and the way one could do it in Lean highlight the perceptive aspect of the programming language. Thus, for a new user, the syntax in Lean will seem natural, easy to read and understand.

```
class G_module (G : Type*) [group G] (M : Type*) [add_comm_group
    M] extends  has_scalar G M :=
(id : ∀ m : M, (1 : G) · m = m)
(mul : ∀ g h : G, ∀ m : M, g · (h · m) = (g * h) · m)
(linear : ∀ g : G, ∀ m n : M, g · (m + n) = g · m + g · n)
```

**Definition** ($G$-homomorphism): A function $f : M \to N$ is called a morphism of $G$-modules or a **$G$-homomorphism** if $f$ satisfies the following, for all $m, n \in M$ and for all $g \in G$:

1. $f(m + n) = fm + fn$;
2. $f(g \bullet m) = g \bullet (fm)$.

Similarly, in Lean we have:

```
 class G_module_hom (f : M → N)  : Prop :=
(add : ∀ a b : M, f (a + b) = f a + f b)
(G_hom : ∀ g : G, ∀ m : M, f (g · m) = g · (f m))
```

## 4.3 0th cohomology group

**Definition** (0th cohomology group): Let $G$ be a group and $M$ a left $G$-module. Then we define $H^0(G, M)$ as being the submodule of $M$ consisting of all the $G$-invariant elements:

$$H^0(G, M) = \{m \in M : \forall g \in G, g * m = m\}$$

The version of this definition in Lean takes as input arguments (the arguments mentioned in round brackets) a group $G$ and a $G$-module $M$:

```
definition H0 (G : Type*) [group G] (M : Type*) [add_comm_group M
    ] [G_module G M]
:= {m : M // ∀ g : G, g · m = m}
```

Let there be an exact sequence of $G$-modules

$$0 \to A \xrightarrow{f} B \xrightarrow{g} C \to 0$$

i.e. we have that $f$ and $g$ are $G$-module homomorphisms with $f$ injective and $g$ surjective, and $\mathrm{Im}(f) = \ker(g)$.

We would like to reason about this short exact sequence after

applying the 0th cohomology group to it – what could we deduce about:

$$0 \to H^0(G, A) \xrightarrow{H^0(f)} H^0(G, B) \xrightarrow{H^0(g)} H^0(G, C) \to 0 \; ?$$

The first instinct would urge us to say that this, too, forms an exact sequence. However, this is only true for most common examples, but not true in general. The exact sequence that we can deduce so far is the following:

$$0 \to H^0(G, A) \xrightarrow{H^0(f)} H^0(G, B) \xrightarrow{H^0(f)} H^0(G, C)$$

Notice that the maps used in the exact sequence involving the 0th cohomology groups are not definitionally identical to the maps we started with, $f$ and $g$. This is because of the difference between a module $M$ and its $G$-invariant, $H^0(G, M)$: the latter is a subgroup of $M$, not necessarily equal to $M$. This subtlety is highlighted in the code by having a separate definition for the maps that connect two 0th cohomology groups:

```
def H0_f (f : M → N) [G_module_hom G f] : H0 G M → H0 G N :=
λ x, ⟨f x.1, λ g, H0.G_module_hom G f g x.1 x.2⟩
```

We take as an input argument the $G$-module homomorphism $f$, the return type (written after ":") being a map of the form $H^0(G, M) \to H^0(G, N)$. We define this new map to be a pair having the function itself as first element, and the second element being a proof that the image is $G$-stable. The proof is listed below as a lemma, using the property of a $G$-homomorphism applied to $f$ (first `rw` in the proof), as well as the given hypothesis `hm` (second `rw` in the proof).

```
lemma H0.G_module_hom (f : M → N) [G_module_hom G f] (g : G) (m
    : M) (hm : ∀ g : G, g · m = m): g · f m = f m :=
  begin
    rw ←G_module_hom.G_hom f g,
    rw hm g,
  end
```

## 4.4 First cohomology group

**Definition** (1-cocycle): Let $G$ be a group, $M$ a $G$-module. Then a **1-cocycle** is a function $\xi : G \to M$ satisfying the cocycle identity, for all $g, h \in G$:

$$\xi(gh) = g \bullet \xi(h) + \xi(g)$$

**Definition** (1-coboundary): Let $G$ be a group, $m$ an element of a $G$-module M. A coboundary is a map $\xi : G \to M$ satisfying the following property for all $g \in G$:

$$\xi(g) = g \bullet m - m$$

**Proposition**: A 1-coboundary is a 1-cocycle.

The corresponding definitions in Lean are as follows:

```
def cocycle (G : Type*) [group G] (M : Type*) [add_comm_group M]
    [G_module G M] :=
{f : G → M // ∀ g h : G, f (g * h) = f g + g · (f h)}

def coboundary (G : Type*) [group G] (M : Type*) [add_comm_group
    M] [G_module G M] :=
  {f : cocycle G M | ∃ m : M, ∀ g : G, f g = g · m - m}
```

**Proposition**: The 1-coboundaries form a subgroup of the 1-cocycles.

The mathematical proof of this proposition follows easily from the properties of 1-coboundaries and 1-cocycles. We will have a look at the proof in Lean, even though it might look complicated, but it goes just as smoothly:

```
instance : is_add_subgroup (coboundary G M) :=
{ zero_mem := begin
      use 0,
      intro g,
      rw g_zero g,
      simp,
      refl,
    end,
  add_mem := begin
      intros a b,
      intros ha hb,
      cases ha with m hm,
      cases hb with n hn,
      use m+n,
      simp [hm, hn],
    end,
  neg_mem := begin
      intro a,
      intro ha,
      cases ha with m hm,
      use -m,
      intro g,
      show - a g = _,
      simp [hm],
      rw g_neg g,
```

```
end }
```

We see that the code contains 3 proofs (one for each `begin`(...)`end` statement), each of them corresponding to the axioms that need to be satisfied by a subgroup, denoted here as `zero_mem`, `add_mem` and `neg_mem`. Having a detailed look at the proof for `neg_mem` would be sufficient to understand the other two, as well. Clicking just after the `begin` word for the `neg_mem` proof we find in the tactic state both the givens and our goal:

```
1 goal
G : Type u_1,
_inst_1 : group G,
M : Type u_2,
_inst_2 : add_comm_group M,
_inst_3 : G_module G M
⊢ ∀ {a : cocycle G M}, a ∈ coboundary G M → -a ∈ coboundary G M
```

We introduce a variable `a` to eliminate the for all statement at the beginning of the goal, as well as a hypothesis `ha` in order to eliminate the implication. We are left with:

```
1 goal
G : Type u_1,
_inst_1 : group G,
M : Type u_2,
_inst_2 : add_comm_group M,
_inst_3 : G_module G M,
a : cocycle G M,
ha : a ∈ coboundary G M
⊢ -a ∈ coboundary G M
```

We know want to make use of the `ha` hypothesis. Saying that `a` is a coboundary is the same as saying that ∃`m` such that the coboundary property is satisfied. To eliminate the there exists statement, we use the `cases` tactic, so we get exactly what we want in `m` and `hm`:

```
1 goal
G : Type u_1,
_inst_1 : group G,
M : Type u_2,
_inst_2 : add_comm_group M,
_inst_3 : G_module G M,
a : cocycle G M,
m : M,
hm : ∀ (g : G), ⇑a g = g · m - m
⊢ -a ∈ coboundary G M
```

In order to prove that −`a` is also a coboundary, we want to find an element of $M$ that satisfies the property; that element is −`m` (`use` −`m`). We then need to check that for this element of $M$ we have that $\forall g \in G$ (so we introduce a variable `g`) we have that:

```
1 goal
```

```
G : Type u_1,
_inst_1 : group G,
M : Type u_2,
_inst_2 : add_comm_group M,
_inst_3 : G_module G M,
a : cocycle G M,
m : M,
hm : ∀ (g : G), ⇑a g = g · m - m,
g : G
⊢ ⇑-a g = g · -m - -m
```

From here the goal seems pretty straightforward: we use the given property `hm`, simplify calculations by using the `simp` tactic and then the goal is accomplished by using the fact that $g \bullet -m = -(g \bullet m)$.

**Definition** (1st cohomology group): The first cohomology group is defined as the quotient group:

$$H^1(G, M) = \text{1-cocycles} \ / \ \text{1-coboundaries}$$

```
def H1 (G : Type*) [group G] (M : Type*) [add_comm_group M] [
    G_module G M] :=
  quotient_add_group.quotient (coboundary G M)
```

# Chapter 5

# Group Cohomology in Lean

Choosing the appropriate platform to formalise abstract concepts is a challenge in itself. Lean fulfills all of the requirements needed to reason about group cohomology in its environment, whilst other theorem provers would struggle with its indefiniteness. As seen before, the definitions of the cohomology groups follow their corresponding mathematical definitions and are easy to understand. I will now focus on the key part of this project – the long exact sequence obtained from the cohomology groups. We will start with the assumption that $A, B$ and $C$ are $G$-modules and $f : A \to B$, $g : B \to C$ are $G$-homomorphisms such that the following is an exact sequence (so we have that $f$ is injective, $g$ is surjective and $\text{Im}(f) = \ker(g)$):

$$0 \to A \xrightarrow{\text{f}} B \xrightarrow{\text{g}} C \to 0$$

## 5.1 Exact sequence with $H^0(G, M)$ only

Given the assumption, we want to prove that the next sequence is exact:

$$0 \to H^0(G, A) \xrightarrow{H^0(f)} H^0(G, B) \xrightarrow{H^0(g)} H^0(G, C)$$

To prove this, it would be sufficient to prove that:

1. the mapping $H^0(f) : H^0(G, A) \to H^0(G, B)$ is injective;
2. the sequence $H^0(G, A) \xrightarrow{H^0(f)} H^0(G, B) \xrightarrow{H^0(f)} H^0(G, C)$ is exact, or equivalently that $\text{Im}(H^0(f)) = \ker(H^0(f))$.

I will go through the reasons why the aforementioned are indeed true by commenting on both the mathematical aspects and the code for them in Lean. I will provide the tactic state quite often, because it is of great help in guiding the user through the proof. We will see that, in fact, every step in the code comes logically when thinking about the current state (givens and goals).

1. The mathematical argument for the injectivity of $H^0(f)$ is simple: $H^0(G, A)$ is a subset of $A$, and $H^0(f)$ acts the same as $f$ on the elements of its domain. We already know that $f$ is injective, so $H^0(f)$ must also be injective. These properties are not predefined in Lean, so we will prove this as one would normally approach showing that a function is injective: starting with $H^0(f)(x) = H^0(f)(y)$ for some $x, y \in H^0(G, A)$, we need to show that $x = y$.

```
lemma H0inj_of_inj {A B : Type*} [add_comm_group A] [
G_module G A] [add_comm_group B] [G_module G B] (f : A
→ B) (H1 : injective f) [G_module_hom G f] : injective
(H0_f G f) :=
  begin
    intro x,
    intro y,
    intro H,
    unfold H0_f at H,
    simp at H,
    have H3 : x.val = y.val,
      exact H1 H,
    exact subtype.eq H3
  end
```

Because the definition of injectivity contains for all statements, we first have to introduce the variables $x$ and $y$. The definition takes the form of an implication, so we also introduce the hypothesis $H$ confirming that $H^0(f)(x) = H^0(f)(y)$, and now the goal is as mentioned before: $x = y$. Recall the definition of $H0\_f$: a pair containing the value of the function and a proof. We want to get to the function; for that, we use the `unfold` tactic to unfold the definition of $H^0(f)$ that appears in hypothesis $H$. After doing so, the current state becomes:

```
1 goal
G : Type u_1,
_inst_1 : group G,
A : Type u_2,
B : Type u_3,
_inst_6 : add_comm_group A,
_inst_7 : G_module G A,
_inst_8 : add_comm_group B,
_inst_9 : G_module G B,
f : A → B,
H1 : injective f,
_inst_10 : G_module_hom G f,
x y : H0 G A,
H : ⟨f (x.val), _⟩ = ⟨f (y.val), _⟩
⊢ x = y
```

The `simp` tactic here uses the fact that for two pairs to be equal, we must have that the first elements of the pairs are also equal, so $H$ becomes just $\mathtt{f}(\mathtt{x.val}) = \mathtt{f}(\mathtt{y.val})$.
We can now see the use of a new tactic, `have H3 : prop`, that is the same as claiming that `prop` is true. For this reason, an extra goal is added that is identical to `prop`, in our case being `x.val` = `y.val`. This is the case indeed, as the given $H1$ tells us that $f$ is injective.
After solving this new goal, hypothesis $H3$ becomes part of the givens. We are again left with the goal `x` = `y`, but we are only one step away from it because of $H3$. The last line in the code uses exactly the property in `subtype.eq` (`x.val` = `y.val` → `x` = `y`), so we now have reached our goals.

2. Proving that $\mathrm{Im}(H^0(f)) = \ker(H^0(g))$ is equivalent to proving that $\mathrm{Im}(H^0(f))$

$\subseteq \ker(H^0(g))$ and that $\ker(H^0(g)) \subseteq \mathrm{Im}(H^0(f))$, thus we will have 2 goals instead of one, which are highlighted in the code below with curly brackets:

```
lemma h0_exact {A B C : Type*} [add_comm_group A] [G_module
    G A] [add_comm_group B] [G_module G B] [add_comm_group C
    ] [G_module G C] (f : A → B) (g : B → C)  (H1 :
    injective f) [G_module_hom G f] [G_module_hom G g] (H2 :
     is_exact f g) : is_exact (H0_f G f) (H0_f G g) :=
  begin
    change range f = ker g at H2,
    apply subset.antisymm,
    { intro x,
      cases x with b h,
      intro h2,
      cases h2 with a ha,
      cases a with a propa, -- point1
      rw mem_ker,
      apply subtype.eq,
      show g b = 0,
      rw ←[mem_ker g, ←H2],
      use a,
      injection ha,
    },
    { rintros ⟨x,h⟩ hx,
      rw mem_ker at hx,
      unfold H0_f at hx,
      injection hx with h2,
      change g x = 0 at h2,
      rw ← mem_ker g at h2,
      rw ← H2 at h2,
      cases h2 with a ha, -- point2
      have h2a : ∀ g : G, g • a = a,
      { intro g,
      apply H1,
      rw G_module_hom.G_hom f,
      rw ha,
      exact h g,
      apply_instance,},
      use ⟨a, h2a⟩,
      apply subtype.eq,
      unfold H0_f,
      exact ha,
    }
  end
```

The proof inside the first pair of curly brackets introduces some variables needed and also extracts the relevant information from each variable representing of type $H^0(G, M)$ (we used `cases` on both `x` and `a`). At *point 1* we have the new givens, along with the goal:

```
b : B,
h : ∀ (g : G), g • b = b,
a : A,
```

```
propa : ∀ (g : G), g · a = a,
ha : H0_f G f ⟨a, propa⟩ = ⟨b, h⟩
⊢ ⟨b, h⟩ ∈ ker (H0_f G g)
```

We use the property of `mem_ker` (`x ∈ ker(f) ⇔ f(x) = 0`) to work on the equality. In our case, this is equivalent to `g(b) = 0` – that's what the line using `show` in the code reflects. Next, we want to make use of the fact that $\mathrm{Im}(f) = \ker(g)$, so we use `mem_ker` again, but in the different direction. Now, it suffices to show that `b ∈ range f`. We need to provide a value `t` for which `f(t) = b`; this is, in fact, `a` (`use a`), which is verified by `ha`. We use `injection ha` instead of `exact ha`, as the equality in `ha` refers to the pairs, and we are only interested in the first values of the pairs.
The other inclusion, $\ker(H^0(g)) \subseteq \mathrm{Im}(H^0(f))$, starts with similar rewritings and alterations of the given hypotheses. Just before the `have` statement (at *point* 2) we have:

```
b : B,
h : ∀ (g : G), g · b = b,
hb : ⟨g (⟨b, h⟩.val), _⟩ = 0,
a : A,
ha : f a = b
⊢ ⟨b, h⟩ ∈ range (H0_f G f)
```

The current goal requires to come up with an element of $H^0(G, A)$ such that, when $H^0(f)$ is applied to it, we get `b`. We have a hint of a choice in `ha`; we want to use `a` as an element of $A$, but first we need to check that the property of $H^0(G, A)$ is indeed true for `a`. This is because when wanting to suggest an element of $H^0(G, A)$ we must provide a pair, with both the element and the proof that it satisfies the property for the 0th cohomology group. The proof of `a` satisfying that property is inside the curly brackets of the `have` statement. Thus, we can now `use` the pair consisting of `a` and `h2a`, the hypothesis obtained after the `have` tactic. The last 3 lines in the proof handle the unfolding of the pairs, but the goal follows naturally after the choice of `a`.

## 5.2 Long exact sequence

We want to prove that, given the assumption made at the beginning of this chapter, the following sequence is exact:

$$0 \to H^0(G,A) \xrightarrow{H^0(f)} H^0(G,B) \xrightarrow{H^0(g)} H^0(G,C) \xrightarrow{\delta}$$
$$H^1(G,A) \xrightarrow{H^1(f)} H^1(G,B) \xrightarrow{H^1(g)} H^1(G,C),$$

where the mappings have the following domains and codomains: $H^0(f) : H^0(G,A) \to H^0(G,B)$, $H^0(g) : H^0(G,B) \to H^0(G,C)$, $H^1(f) : H^1(G,A) \to H^1(G,B)$, $H^1(g) : H^1(G,B) \to H^1(G,C)$, and the function $\delta : H^0(G,C) \to H^1(G,A)$ is the connecting homomorphism.

To prove this, having the result in the previous section, means to prove that all the following sequences are short exact sequences:

1. $H^0(G,C) \xrightarrow{\delta} H^1(G,A) \xrightarrow{H^1(f)} H^1(G,B)$

2. $H^1(G,A) \xrightarrow{H^1(f)} H^1(G,B) \xrightarrow{H^1(g)} H^1(G,C)$

3. $H^1(G,A) \xrightarrow{H^1(f)} H^1(G,B) \xrightarrow{H^1(g)} H^1(G,C)$

Similar arguments are used in the proofs of the exactness of the sequences above. All the 3 proofs in Lean for the sequences mentioned above can be found in the appendix. I will briefly go through the mathematical reasoning of the third sequence.

We want to prove that $\mathrm{Im}(H^1(f)) = \ker(H^1(g))$.

1. Proving that $\mathrm{Im}(H^1(f)) \subseteq \ker(H^1(g))$ is equivalent to proving that for every element $a \in H^1(G,A)$ we have $H^1(g)(H^1(f)(a)) = 0$. We want to transform the problem into one involving cocycles, so we lift $a$ to $c \in Z^1(G,A)$, where $c : G \to A$ is a cocycle. Easy check: the element of $H^1(G,C)$ is represented by a cocycle : $G \to C$, and it would be sufficient to prove that this cocycle is 0. We know from the short exact sequence of $G$-modules in the assumption made at the beginning of this chapter that $\mathrm{Im}(f) = \ker(g)$, so the path from $A$ to $B$ to $C$ is equivalent to a map from $A$ to $C$ that sends everything to 0. Thus, the cocycle from $G$ to $C$ is equivalently obtained through the path from the cocycle $c : G \to A$ and then the mapping from $A$ to $C$ that sends everything to 0. We can conclude that the cocycle from $G$ to $C$ is 0.

2. The proof for $\ker(H^1(g)) \subseteq \mathrm{Im}(H^1(f))$ follows the same idea as in the inclusion that was just proved. The main idea is to change the problem from quotients to cocycles and coboundaries, which has been highlighted in the code from the appendix.

# Chapter 6

# Evaluation

## 6.1 First impressions on Lean

The Lean theorem prover is currently used by a constantly increasing community that "gathers" on the Zulip chat to discuss various topics related to the system. Extremely helpful and always willing to help, the more experienced users guide the others through their first encounters with Lean. Questions are asked frequently, from the most basic ones to more complicated topics – there is always someone answering or giving a hint.

My first contact with Lean was when meeting people from the Xena project, mostly being undergraduate students, part of the mathematics department, at Imperial College London. The project was initiated by my supervisor, professor Kevin Buzzard, with the purpose of showing students how to use formal proof verification software to manipulate problems that appear in their courses. Being someone who studies both mathematics and computer science, I instantly became engaged, especially when seeing the dedication of both the professor and the students involved.

I started by solving some basic examples in logic, which I tried on the web browser interface. The first steps seemed straightforward, but moving on to more complicated problems has proven to be a pain, as the interface became slower and slower. Installing Lean on my laptop was well-explained by experienced users in an article and I could quite rapidly start coding in Visual Studio Code. However, when trying to approach the topic of group cohomology, there were a lot of setbacks, due to my lack of experience.

Professor Buzzard was the one who guided me at every step before getting the gist of dealing with Lean. From syntax to built in functions and lemmas, everything was new and I felt in the dark, but the joy of reaching the `goals accomplished` tactic state was extremely motivating. I think that most students that take part in the Xena project have had similar experiences, but we all consider that Lean has great perspective. We believe that Lean could play a tremendous role in improving mathematics research by having computers helping humans, for example, by filling in proofs.

## 6.2 Reflection on Lean

One instantly noticeable characteristic of the programming language is that it resembles in detail the mathematical syntax, especially when it comes to defining new structures. This improves readibility and helps the user in writing down intuitive code easier. However, when it comes to filling in proofs, one must be aware of the tactics and their functionality, as well as the names of relevant lemmas. This makes proofs not as easy to unravel for the new comers.

We can definitely deduce that the learning curve is steep and that without proper guidance, Lean might seem unapproachable. This idea has also been highlighted by Thomas Hales from the University of Pittsburgh in one of his blog posts about Lean: "It is very hard to learn to use Lean proficiently. Are you a graduate student at Stanford, CMU, or Pitt writing a thesis on Lean? Are you a student at Imperial being guided by Kevin Buzzard? If not, Lean might not be for you".

Even though having a better understanding of how to use Lean's features takes time, I still believe it is worthwhile. Lean was designed after having the past decade to research into the Coq system, an interactive theorem prover initially developed in 1984, in order to significantly improve it. When it comes to the topic of this project, group cohomology, Lean was by far the best choice. Coq, for example, would not have been able to deal with quotients as straightforward, one would have to do it with equivalence classes. In contrast, Lean has quotients built into its kernel.

Another important aspect is that the system is designed to take on abstract concepts and long proofs. It is based on Calculus of Constructions, a version of dependent type theory. This provides a powerful language that allows the user to express complicated mathematical assertions, write complex hardware and software specifications, while also reasoning about both in an uniform way. It is a major asset when thinking about other systems, such as Isabelle and HOL-Light, that use simple type theory, which are not as expressive as Lean.

# Chapter 7

# Conclusion and Future Work

This paper presented the Lean theorem prover, whose framework supports user interaction and the construction of fully specified axiomatic proofs. It is appealing because of its open source, in-depth documentation, a small trusted kernel, and most importantly due to its support for both classical and constructive mathematics. It can be equally viewed as a programming language, because of its underlying logic. In addition, the system is its own metaprogramming language: one can extend the functionality of Lean using Lean itself.

Formalising domains such as group cohomology in Lean is a challenge, but the abstractness in the notions that the system handles makes it a unique experience. The dependent type theory, along with the way Lean is able to infer types are essential in this project. It is worth mentioning that Lean does not know anything about group cohomology and that the structures were built from scratch. This was one of the main reasons I chose to formalise group cohomology instead of a topic that is already part of the maths library of Lean. I can confirm that the process is slow at the beginning, but once the important design choices are made, every proof follows quite naturally.

When it comes to future work, there are various extensions that could be done. Group cohomology is a vast area, and I have only managed to touch the 0th and 1st cohomology groups, together with important theorems. One potential augmentation of the code would be proving the restriction-inflation sequence:

**Proposition**: Let $N$ be a normal subgroup of $G$ and $M$ a $G$-module. Then the following sequence is exact:

$$0 \to H^1(G/N, M^N) \xrightarrow{\text{Inf}} H^1(G, M) \xrightarrow{\text{Res}} H^1(N, M)$$

Another extension would be approaching higher order cohomology groups, such as $H^2(G, M)$, or even the general case, $H^n(G, M)$. The long exact sequence could be extended further, by introducing in the sequence $H^2(G, A)$, $H^2(G, B)$ and $H^2(G, C)$, and then the higher cohomology groups (the sequence is, in fact, infinite). Thus, multiple lemmas and properties might be furtherly proved, and the code that has already been written would serve as a basis.

# Appendix A

# Appendix

These are the proofs in Lean for the 3 short exact sequences that are sufficient to prove the continuation of the long exact sequence after $H^0$:

1. $H^0(G, C) \xrightarrow{\delta} H^1(G, A) \xrightarrow{H^1(f)} H^1(G, B)$

```
lemma h0b_hoc_h1a_exact (G : Type*) [group G]
{A : Type*} [add_comm_group A] [G_module G A]
{B : Type*} [add_comm_group B] [G_module G B]
{C : Type*} [add_comm_group C] [G_module G C]
{f : A → B} [G_module_hom G f]
{g : B → C} [G_module_hom G g]
(hf : injective f)
(hg : surjective g) (hfg : range f = ker g)
: is_exact (H0_f G g) (delta G hf hg hfg) :=
  begin
    apply subset.antisymm,
    { intros fc h,
    rw mem_ker,
    cases h with b hb,
    cases fc with c fc,
    cases b with b propb,
    injection hb,
    change g b = c at h_1,
    unfold delta,
    suffices : delta_cocycle G hf hg hfg ⟨c, fc⟩ ∈ ker (
  quotient_add_group.mk),
      rw mem_ker at this,
      exact this,
      swap, apply_instance,
      swap, apply_instance,
    rw quotient_add_group.ker_mk,
    let b' : B := delta_b G hg ⟨c, fc⟩,
    have hb' : b' - b ∈ range f,
      have hb'' : b' - b ∈ ker g,
        rw mem_ker,
        rw is_add_group_hom.map_sub g,
        rw delta_im_b G hg ⟨c, fc⟩,
        simp,
      rw h_1,
    simp,
```

```
  rw hfg,
  exact hb'',
  cases hb' with a ha,
  unfold delta_cocycle,
  use a,
  intro γ,
  apply hf,
  rw is_add_group_hom.map_sub f,
  rw G_module_hom.G_hom f,
  swap, apply_instance,
  swap, apply_instance,
  show f (delta_cocycle_aux G hg hfg ⟨c, fc⟩ γ) = _,
  rw delta_cocycle_aux_a G hg hfg ⟨c, fc⟩ γ,
  rw ha,
  rw G_module.map_sub,
  change γ · b' - b' = γ · b' - γ · b - (b' - b),
  rw propb,
  simp,
},
{ intros x h,
  cases x with c fc,
  rw mem_ker at h,
  unfold delta at h,
  replace h := (mem_ker quotient_add_group.mk).2 h,
  rw quotient_add_group.ker_mk (coboundary G A) at h,
  cases h with a ha,
  let b' : B := delta_b G hg ⟨c, fc⟩,
  let b : B := f a,
  use b'-b,
  intro γ,
  have h1 := ha γ,
  have h2 := congr_arg f h1,
  change f (delta_cocycle_aux G hg hfg ⟨c, fc⟩ γ) = _ at
 h2,
  rw delta_cocycle_aux_a G hg hfg ⟨c, fc⟩ γ at h2,
  change γ · b' - b' = _ at h2,
  rw is_add_group_hom.map_sub f at h2,
  rw G_module_hom.G_hom f at h2,
  change γ · b' - b' = γ · b - b at h2,
  rw ←sub_eq_zero at h2,
  rw G_module.map_sub,
  rw ←sub_eq_zero,
  swap,
  apply_instance,
  swap,
  apply subtype.eq,
  unfold H0_f,
  show g (b' - b) = c,
  rw is_add_group_hom.map_sub g,
  show g (delta_b G hg ⟨c, fc⟩) - g b = _,
  rw delta_im_b G hg ⟨c, fc⟩,
  have hb' : b ∈ range f,
    use a,
```

```
    have hb : b ∈ ker g,
      rw ←hfg,
    exact hb',
    rw mem_ker at hb,
    show c - g b = c,
    rw hb,
    simp,
    rw ←sub_add,
    rw ←sub_add at h2,
    rw add_comm (γ · b' - γ · b - b') b,
    rw add_comm (γ · b' - b' - γ · b) b at h2,
    rw ←sub_add_eq_sub_sub_swap (γ · b') b' (γ · b),
    rw ←neg_neg (γ · b),
    change b + (γ · b' - (b' - (-(γ · b)))) = 0,
    rw ←sub_add,
    exact h2,
  },
end
```

2. $H^1(G,A) \xrightarrow{H^1(f)} H^1(G,B) \xrightarrow{H^1(g)} H^1(G,C)$

```
lemma h0c_h1a_h1b_exact (G : Type*) [group G]
{A : Type*} [add_comm_group A] [G_module G A]
{B : Type*} [add_comm_group B] [G_module G B]
{C : Type*} [add_comm_group C] [G_module G C]
{f : A → B} [G_module_hom G f]
{g : B → C} [G_module_hom G g]
(hf : injective f)
(hg : surjective g) (hfg : range f = ker g)
: is_exact (delta G hf hg hfg) (H1_f G f) :=
  begin
    apply subset.antisymm,
    { intros fa h,
    rw mem_ker,
    cases h with c hc,
    rw ←hc,
    unfold delta,
    rw cocycle.map_mk G f (delta_cocycle G hf hg hfg c),
    suffices : (cocycle.map G f (delta_cocycle G hf hg hfg
   c)) ∈ ker (quotient_add_group.mk),
      rw mem_ker at this,
      exact this,
      swap, apply_instance,
      swap, apply_instance,
    rw quotient_add_group.ker_mk,
    use (delta_b G hg c),
    intro γ,
    change f (delta_cocycle_aux G hg hfg c γ) = _,
    exact delta_cocycle_aux_a G hg hfg c γ,
    },
    { intros x h,
    rw mem_ker at h,
    induction x,
```

```
  swap,
  refl,
  unfold H1_f at h,
  change quotient_add_group.mk (cocycle.map G f x) = 0
at h,
  rw ←mem_ker quotient_add_group.mk at h,
  swap,
  apply_instance,
  swap,
  apply_instance,
  rw quotient_add_group.ker_mk at h,
  cases h with b hb,
  change ∀ (g : G), f (x g) = g · b - b at hb,
  let c : C := g b,
  use c,
  intro γ,
  rw ←sub_eq_zero,
  rw ←G_module_hom.G_hom g,
  rw ←is_add_group_hom.map_sub g,
  rw ←mem_ker g,
  rw ←hfg,
  rw ←hb,
  use x γ,
  apply_instance,
  change delta G hf hg hfg ⟨c, _⟩ = quotient_add_group.
mk x,
  unfold delta,
  apply quotient_add_group.eq.2,
  have hc : ∀ γ : G, γ · c = c,
    intro γ,
    rw ←sub_eq_zero,
    change γ · g b - g b = 0,
    rw ←G_module_hom.G_hom g,
    rw ←is_add_group_hom.map_sub g,
    rw ←mem_ker g,
    have h1 : γ · b - b ∈ range f,
      rw ←hb,
      use x γ,
    convert h1,
    exact hfg.symm,
    apply_instance,
  let b' : B := delta_b G hg ⟨c, hc⟩,
  have hb' : b - b' ∈ range f,
    have hb'' : b - b' ∈ ker g,
      rw mem_ker,
      rw is_add_group_hom.map_sub g,
      rw delta_im_b G hg ⟨c, hc⟩,
      simp,
    rw hfg,
    exact hb'',
  cases hb' with a ha,
  use a,
  intro g',
```

```
      change - (delta_cocycle_aux G hg hfg ⟨c, hc⟩ g') + x g
    ' = g' · a - a,
      apply hf,
      rw is_add_group_hom.map_add f,
      rw is_add_group_hom.map_neg f,
      rw is_add_group_hom.map_sub f,
      rw G_module_hom.G_hom f,
      rw delta_cocycle_aux_a G hg hfg ⟨c, hc⟩ g',
      rw hb,
      rw ha,
      change - (g' · b' - b') + (g' · b - b) = _,
      rw G_module.map_sub,
      simp,
      apply_instance,
  },
end
```

3. $H^1(G, A) \xrightarrow{H^1(f)} H^1(G, B) \xrightarrow{H^1(g)} H^1(G, C)$

```
lemma h1a_h1b_h1c_exact (G : Type*) [group G]
{A : Type*} [add_comm_group A] [G_module G A]
{B : Type*} [add_comm_group B] [G_module G B]
{C : Type*} [add_comm_group C] [G_module G C]
{f : A → B} [G_module_hom G f]
{g : B → C} [G_module_hom G g]
(hf : injective f)
(hg : surjective g) (hfg : range f = ker g)
: is_exact (H1_f G f) (H1_f G g) :=
  begin
  apply subset.antisymm,
  { intros fb h,
  rw mem_ker,
  cases h with fa hfa,
  induction fa,
  swap,
  refl,
  change H1_f G f (quotient_add_group.mk fa) = _ at hfa,
  rw ←hfa,
  rw cocycle.map_mk G f fa,
  rw cocycle.map_mk G g,
  suffices : (cocycle.map G g (cocycle.map G f fa)) ∈ ker
  (quotient_add_group.mk),
    rw mem_ker at this,
    exact this,
    swap, apply_instance,
    swap, apply_instance,
  rw quotient_add_group.ker_mk,
  use 0,
  intro x,
  cases fa with fa pfa,
  show g (f (fa x)) = _,
  rw g_zero,
  rw sub_zero,
```

```
    rw ←mem_ker g,
    rw ←hfg,
    use fa x,
    },
    { intros x h,
    induction x,
    swap,
    refl,
    rw mem_ker at h,
    unfold H1_f at h,
    change quotient_add_group.mk (cocycle.map G g x) = 0 at
    h,
    rw ←mem_ker quotient_add_group.mk at h,
    swap,
    apply_instance,
    swap,
    apply_instance,
    rw quotient_add_group.ker_mk at h,
    cases h with c hc,
    rcases (hg c) with ⟨b, rfl⟩,
    let y := x - cocycle.mk b,
    have hy : range y ⊆ range f,
    {
      rintros b' ⟨γ, rfl⟩,
      rw hfg,
      rw mem_ker,
      change g ( x γ - (γ • b - b)) = 0,
      rw is_add_group_hom.map_sub g,
      rw sub_eq_zero,
      convert hc γ,
      rw is_add_group_hom.map_sub g,
      congr',
      rw G_module_hom.G_hom g,
      apply_instance,
    },
    let z : cocycle G A := cocycle.lift G f hf y hy,
    use quotient_add_group.mk z,
    change quotient_add_group.mk (cocycle.map G f (cocycle.
    lift G f hf y hy)) = quotient_add_group.mk x,
    rw cocycle.lift_eq,
    apply quotient_add_group.eq.2,
    use b,
    have hb : -y + x = cocycle.mk b,
    {
      show -(x - cocycle.mk b) + x = _,
      simp,
    },
    intro g',
    rw hb,
    refl,
  },
end
```

# Bibliography