Imperial College London

Mathematics

M4R Project

Group Cohomology in Lean Theorem Prover

Author
Shenyang Wu

With thanks to my supervisor
Professor Kevin Buzzard

June 8, 2020

Abstract

Lean Theorem Prover provides an interactive platform for users to state and prove mathematical concepts in a computer-readable manner. Since Lean first launched in 2013, many work to enrich its math library and to test the ability of Lean in understanding advance mathematical topics. This project aims to formalize in Lean the first general definition of group cohomology, $H^n(GM)$ for any $n \in \mathbb{N}$, group G and G-module M in any theorem provers. This report mainly consists of two parts. Part 1 is the blueprint of group cohomology, serving as a detailed mathematical guidance for Part 2, the formalization process in Lean. Definitions of cochain, differential in a complex, cocycle, coboundary and cohomology are given in a technical feasible and computational convenient manner and successfully formalized in Lean. Some progress to formalizing the induced long exact sequence of group cohomology are made. This project demonstrates Lean's capability of understanding group cohomology and Lean verifies the blueprint provided. More importantly, formalizing group cohomology in Lean opens Lean up to more advanced topics such as class field theory due to group cohomology's great usefulness in both local and global field theory.

Contents

1 Introduction

> "After World War II, developments in class field theory led to the stripping away of the algebras in proofs of class field theory as far as possible, leaving behind only cohomological formalism. "
>
> *KEITH CONRAD*

1.1 Overview

Lean Theorem Prover, launched by Leonardo de Moura at Microsoft Research Redmond in 2013, is an interactive platform where users can state definitions and prove theorems. The level of readability and verifiability of Lean with regards to mathematical concepts are of the interests of many mathematicians, and they test it by formalizing complex high-level mathematical topics in Lean. There are many on-going Lean projects on different aspects of mathematics, such as Patrick Massot's work on "Sphere Eversion in Lean", and "Formalizing the Solution to the Cap Set Problem" by Sander R. Dahmen, Johanne Holzl, and Robert Y. Lewis. Most of such Lean projects have two main steps,

- Firstly, a detailed mathematics document containing relevant explicit definitions and paper proofs of theorems is constructed, normally by mathematicians. This is called the *blueprint*.
- Then, following the guidance in the blueprint, is the formalization process in Lean, usually done by computer scientists.

This project aims to formalize a general definition of group cohomology in Lean, which is unprecedented in any theorem provers. Group cohomology is an important tool used to study groups. The particular way it re-assembles information about a group and its module makes group cohomology the correct language for class field theory.

Adapts similar methodology as other Lean projects, this report has two main parts,

- Part 1: The *blueprint* for group cohomology, focusing on detailed derivation of the definition and proofs of some properties. This is in Section 2.
- Part 2: The implementation of concepts into Lean, using the blueprint as a guidance. This is in Section 4, where Section 3 is a brief tutorial to Lean.
- Section 5 summaries the main results of this project and lists possible future researches.

It is remarkable that we have achieved both parts, providing the blueprint and achieving formalization in Lean for group cohomology in this single project.

1.1.1 Part 1: The Blueprint

There are multiple ways to define group cohomology. For technical feasibility and computational convenience, we choose the definition that arises from the standard complex.

Definition 1.1. *A complex is a sequence of abelian groups, $A^0, A^1, A^2, \cdots$, connected by differentials $d^n : A^n \to A^{n+1}$ such that $d^{n+1} \circ d^n = 0$. It can be written as*

$$A^0 \xrightarrow{d^0} A^1 \xrightarrow{d^1} A^2 \xrightarrow{d^2} \cdots$$

For a given group G and a G-module M, we will proceed based on the standard complex above,

- Define abelian groups $\{A^n\}$, where an element of A^n is called an n-cochain
- Define differentials $d^n : A^n \to A^{n+1}$
- Prove that $d^{n+1} \circ d^n = 0$
- Prove that d^n is a group homomorphism
- Define n-cocycle, $Z^n(G, M)$, as kernel of d^{n+1}
- Define n-coboundary, $B^n(G, M)$, as image of d^n
- Define n-cohomology, $H^n(G, M)$, as the quotient of $Z^n(G, M)$ over $B^n(G, M)$.

1.1.2 Part 2: Formalization in Lean

The structure of this part is analogue to that of Part 1 as we follow the blueprint for formalization. We will focus on the special properties of Lean as compared to a paper document, such as its *non-triviality, explicitness, dependence and type-specificity*. Rationales of construction, difficulties faced, solutions chosen as a result of these properties will be discussed thoroughly and final results will be presented.

The main goal of this project is to fill in an appropriate type and an explicit definition to replace `sorry` in the following,

```
def cohomology (n:ℕ )(G : Type*) [group G] (M : Type*) [add_comm_group M]
    [G_module G M]: SOME TYPE := sorry
```

The Lean repository for this entire project is available online at `https://github.com/Shenyang1995/M4R`.

The audience is assumed to know nothing about group cohomology or Lean, but with certain level of undergraduate mathematical knowledge, such as groups, subgroups, quotient groups, homomorphisms, kernels, ranges,and modules.

1.2 Literature Review

Mathematically speaking, group cohomology is well developed in homological algebra. Most textbooks relating this topic provide the definitions of complex, cochain, differential, cocycle, coboundary and cohomology. During the derivation of the definition, some authors would leave out certain details as practices to the reader. Namely, one property of the differentials d^n that is crucial to the definition of cohomology is that $d^{n+1} \circ d^n = 0$, which most texts state without proving. Further details have to be provided to produce a blueprint for group cohomology.

For the process of formalization in Lean, there is an established math library in Lean, consisting of many basic definitions and theorems that can be directly used. The documents that are highly relevant to this project includes `algebra.basic, group_theory, algebra.module, data.set, algebra.pi_instances, tactic.linarith, tactic.omega,`

`tactic.fin_cases`. These files will be imported to our repository and content will be used without proofs.

Previously, Anca Ciobanu had done the definitions of group cohomology for the first two cases, H^0 and H^1, and the corresponding exact sequences in Lean. She provided the definition of n-cocycles and n-coboundaries in the cases of $n = 0$ and $n = 1$ as explicit expression in terms of functions. Unfortunately, this approach fails for a general definition of n-cocycles, n-coboundaries and H^n of any natural number n.

1.3 Motivation

1.3.1 Usefulness of Group Cohomology

In homological algebra, group cohomology is a set of useful mathematical tools used to study groups. It provides insight into the structure of the group G and G-module M themselves. A well-known result formulated in group cohomology is Hilbert's Theorem 90, which states, if L/K is a finite Galois extension of fields with Galois group $G = Gal(L/K)$, then the 1-cohomology is trivial:

$$H^1(G, L^\times) = 1$$

One application of this theorem is when L/K is the quadratic extension $\mathbb{Q}(i)/\mathbb{Q}$. Hilbert's Theorem 90 provides the well-known parametrization for the rational points of the unit circle $x^2 + y^2 = 1$,

$$x = \frac{u^2 - v^2}{u^2 + v^2}, y = \frac{2uv}{u^2 + v^2},\ u, v \in \mathbb{Q}$$

This gives full characterization of all Pythagoras Triples, that is, the integral solutions to the equation $x^2 + y^2 = z^2$.

Another application is the Brauer group. For any arbitrary field k, The Brauer group $Br(k)$ of k is the abelian group whose elements are the equivalence classes of central simple algebras over k. The Brauer group $Br(k)$ is completely described by group cohomology, in particular Galois cohomology, as in

$$Br(k) = H^2(G, k_s^\times)$$

where k_s is a separable closure of k and $G = Gal(k_s/k)$ is the Galois group of the extension.

This is an example of 2-cohomology. Moreover, as the Brauer group plays an important role in the modern formulation of class field theory, study of group cohomology hence is essential and useful for both local and global field theory.

1.3.2 The First General Definition in any Theorem Prover

With Anca's work on H^0 and H^1, one asks whether it is possible to provide a general definition of group cohomology for any natural number n in Lean. On the other hand, there are concepts about group cohomology in GAP which enables users to compute explicit cohomology groups by providing specific inputs $n \in \mathbb{N}$, group G and G-module M. GAP, stands for Groups, Algorithms, Programming, is a system for computational discrete algebra with a programming language and an interactive platform similar to Lean. However, both Anca's and GAP's definitions of group cohomology requires an explicit value of $n \in \mathbb{N}$, and fail to provide a general definition of n-cohomology for an arbitrary unspecified $n \in \mathbb{N}$. As a result, both of their definitions cannot be used to state theorems about a general group cohomology, such as the induced long exact sequence, and give proofs. As to this point in

time, there is no well-known general definition of group cohomology in any theorem provers and the result from this project may well be the first ever definition of group cohomology for any natural number n in a theorem prover. This brings significance as once group cohomology is defined in Lean, it opens Lean up to a large number of possible research directions due to group cohomology's great usefulness. One can move on to prove properties of group cohomology, which further lead to the possibility of formalizing concepts of class field theory in Lean, a rather high level topic in mathematics.

This unprecedented nature brings challenging to this project, as

- Anca's methodology for defining group cohomology fails as we do not have explicit expressions for cocycles and coboundaries anymore.
- Lean does not have a very simple way to deal with an arbitrary unspecified general number n.
- One has to consider whether methods used in the definitions and proofs are programmable in Lean. There was no guarantee that Lean is able to understand a general notion of group cohomology at the beginning of the project.
- One also needs to look out for the best feasible solution in terms of computationally convenience for future uses of definitions and theorems.

Finding such a definition of group cohomology in Lean hence motivates this project.

1.4 Special Notes on $n \in \mathbb{N}$

For this report, $\mathbb{N}$ will taken to be the set of integers starting from 0. This is consistent with the set up in Lean.

In addition, we will avoid, anywhere possible, the use of subtraction in any term that represents a shift in n, such as $n-1$. This is because subtraction like $n-1$ will cause complexity in Lean when $n = 0$, as `0-1=0` in Lean by convention.

In terms of notation, for well-known concepts, such as differentials d^n, cocycle $Z^n(G, M)$, coboundary $B^n(G, M)$ and cohomology $H^n(G, M)$, the variable $n \in \mathbb{N}$ is placed as superscript to make sure consistency with most textbooks. This is to avoid confusion with those placing as subscripts that represent the dual concepts in homology. When there is no confusion, n might be placed as subscript.

2 Group Cohomology

2.1 Cochain

Definition 2.1. *Given a group G, a (left) G-Module M and a natural number $n \in \mathbb{N}$, an n-cochain is function from G^n to M.*

2.2 A Contraction Map

Definition 2.2. *Given a group G, a natural number $n \in \mathbb{N}$ and another natural number $j < n$, we can define a contraction map $F_{n,j}$, from $G^{(n+1)}$ to G^n simply as follow,*

$$F_{n,j} : G^{(n+1)} \to G^n$$

$$F_{n,j}(g_0, g_1, \cdots, g_n) = (g_0, g_1, \cdots, g_{j-1}, g_j g_{j+1}, g_{j+2}, g_{j+3}, \cdots, g_n)$$

where $g_i \in G \ \forall i \in \mathbb{N}$.

More explicitly, we can see that $F_{n,j}$ just multiplies g_j and g_{j+1} together and puts it into the place where g_j initially was. Hence, $F_{n,j}$ simply reduces the dimension by 1. One can also give the following equivalent definition of $F_{n,j}$ in terms of its explicit computation to each g_i,

$$F_{n,j}(g_0, g_1, \cdots, g_n) = (a_0, a_1, \cdots, a_{n-1})$$

where

$$\forall i \leq n-1,\ a_i = \begin{cases} g_i & i < j \\ g_j g_{j+1} & i = j \\ g_{i+1} & i > j \end{cases}$$

Remark 1: Notice that using the above explicit definition of $F_{n,j}$, the requirement that $j < n$ can be freed. For $j \geq n$, $F_{n,j}$ simply deletes that last entry g_n, i.e,

$$F_{n,j}(g_0, g_1, \cdots, g_n) = (g_0, g_1, \cdots, g_{n-1})$$

Hence, $F_{n,j}$ is actually well defined for all $j \in \mathbb{N}$. We will use this explicit definition of $F_{n,j}$ unless otherwise stated.

Remark 2: Notice that for the $i > j$ case in the explicit definition of $F_{n,j}$, there is a position shift of 1 of g_{i+1}. This is very important to keep in mind as we proceed to the next subsection.

2.3 An Important Result on Double Sum

We are interested in a special property of using this contraction map $F_{n,j}$ twice, i.e. we look at $F_{n,i} \circ F_{n+1,j}$ for some pairs of $i, j \in \mathbb{N}$. ***Question:*** What will happen if we exchange the places of i and j? Intuition might tell us that there should be some relation between the two results that we obtained by performing this exchange, as they both represent some sort of dimension shrinking at places g_i and g_j, and doing at position i or position j first should give the same result. Well, this will true if we do not have any position shift. However, by *Remark 2*, we do have one and this leads to the following lemma,

Lemma 2.3. *Given a group G, for any natural number $n \in \mathbb{N}$, and any two natural numbers $j, k \in \mathbb{N}$ such that $j \leq k$, we have*

$$F_{n,j} \circ F_{n+1,k+1} = F_{n,k} \circ F_{n+1,j}$$

Proof. Consider the left-hand side (LHS) first. Using the explicit definition of $F_{n+1,k+1}$, we have

$$(F_{n,j} \circ F_{n+1,k+1})(g_0, g_1, \cdots, g_{n+1}) = F_{n,j}(a_0, a_1, \cdots, a_n)$$

where

$$\forall i \leq n,\ a_i = \begin{cases} g_i & i < k+1 \\ g_{k+1}g_{k+2} & i = k+1 \\ g_{i+1} & i > k+1 \end{cases}$$

According to the definition again, we then have to compare j with the indexes i of a_i. Thus we have,

$$(F_{n,j} \circ F_{n+1,k+1})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} a_i & i < j \\ a_j a_{j+1} & i = j \\ a_{i+1} & i > j \end{cases}$$

For the first case $i < j$, it is simple as $i < j \implies i < k+1$ because $j \leq k \implies j < k+1$. Hence we always have $b_i = a_i = g_i$ for $i < j$. For second case $i = j$, one special possibility is when a_{j+1} happens to be the product $g_{k+1}g_{k+2}$. This happens when $j+1 = k+1 \implies j = k$. In this situation, $i = j \implies b_i = a_j a_{j+1} = g_k g_{k+1} g_{k+2}$. Also, $i > j \implies i+1 > j+1 = k+1$, hence for the third case $i > j$, we have $b_i = a_{i+1} = g_{i+1+1} = g_{i+2}$. In summary, when $j = k$, we have the LHS as

$$(F_{n,j} \circ F_{n+1,k+1})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} g_i & i < j \\ g_k g_{k+1} g_{k+2} & i = j \\ g_{i+2} & i > j \end{cases}$$

Now we continue to compute LHS when $j < k$. Notice $j < k \implies j+1 < k+1 \implies b_i = a_j a_{j+1} = g_j g_{j+1}$ for the second case $i = j$. For the third case $i > j$, we have to compare the indexes $i+1$ of a_{i+1} with $k+1$, which separates into 3 possibilities:

- $i+1 < k+1 \implies i < k \implies b_i = a_{i+1} = g_{i+1}$

- $i+1=k+1 \implies i=k \implies b_i = a_{i+1} = g_{k+1}g_{k+2}$
- $i+1<k+1 \implies i<k \implies b_i = a_{i+1} = g_{i+1+1} = g_{i+2}$

Putting all together yields when $j<k$,

$$(F_{n,j} \circ F_{n+1,k+1})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} g_i & i<j \\ g_k g_{k+1} & i=j \\ g_{i+1} & j<i<k \\ g_{k+1}g_{k+2} & j<i=k \\ g_{i+2} & i>k>j \end{cases}$$

Similarly, we can construct the explicit computation of the right-hand side (RHS), as follow,

$$(F_{n,k} \circ F_{n+1,j})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} a_i & i<k \\ a_k a_{j+1} & k=k \\ a_{i+1} & i>k \end{cases}$$

and

$$\forall i \leq n,\ a_i = \begin{cases} g_i & i<j \\ g_j g_{j+1} & i=j \\ g_{i+1} & i>j \end{cases}$$

When $j=k$, we have

- $i<k \implies i<j \implies b_i = a_i = g_i$
- $i=k \implies i=j, k+1=j+1>j \implies b_i = a_j a_{j+1} = g_j g_{j+1} g_{j+2} = g_j g_{k+1} g_{k+2}$
- $i>k \implies i>j \implies b_i = a_{i+1} = g_{i+1+1} = g_{i+2}$

which means,

$$(F_{n,k} \circ F_{n+1,j})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} g_i & i < j \\ g_k g_{k+1} g_{k+2} & i = j \\ g_{i+2} & i > j \end{cases}$$

This is exactly the same as that of LHS when $j = k$.

Similarly for $j < k$,

- For $i < k$, we compare i with j for the following 3 possibilities:
  - $i < j < k \implies b_i = a_i = g_i$
  - $i = j < k \implies b_i = a_i = g_j g_{j+1}$
  - $j < i < k \implies b_i = a_i = g_{i+1}$
- $i = k \implies i > j, \implies b_i = a_j a_{j+1} = g_{k+1} g_{k+2}$
- $i > k \implies i > j \implies b_i = a_{i+1} = g_{i+1+1} = g_{i+2}$

This is exactly

$$(F_{n,k} \circ F_{n+1,j})(g_0, g_1, \cdots, g_{n+1}) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = \begin{cases} g_i & i < j \\ g_k g_{k+1} & i = j \\ g_{i+1} & j < i < k \\ g_{k+1} g_{k+2} & j < i = k \\ g_{i+2} & i > k > j \end{cases}$$

and it is identical to that of LHS. This completes the proof. □

One may argue that just by intuition that the lemma follows directly from that "The result obtained by contraction at j-th position and then contracts again at k-th position is the same as contracting at $(k+1)$-th position and then j-th position, where the $+1$ is just the result of the position shift of the given contraction." Surely, there are simpler convincing proofs for a well-trained mathematician, but this proof shows, as an example, how detailed and concrete we have to go in order to make it work in Lean. That is, we have to provide an explicit definition and use this definition to prove the lemma by considering all the possible cases.

Following this lemma, we arrive at the important result about double sum stated as follow,

Lemma 2.4. *Given a group G, for any natural number $n \in \mathbb{N}$, and for any natural number $m \in \mathbb{N}$ we have*

$$\sum_{i=0}^{m+1}\sum_{j=0}^{m}(-1)^{i+j}F_{n,j}\circ F_{n+1,i}=0$$

where 0 *here is the zero map.*

Proof. Here is an easy proof using induction on m. Fixing any $n\in\mathbb{N}$, we have base case $m=0$ as

$$(-1)^{0+0}F_{n,0}\circ F_{n+1,0}+(-1)^{0+1}F_{n,0}\circ F_{n+1,1}=0$$

since $F_{n,0}\circ F_{n+1,0}=F_{n,0}\circ F_{n+1,1}$ which is an immediate application of Lemma 2.3 setting $j=0$, $k=0$.

Assume that the lemma holds for $m=k$ for some $k\in\mathbb{N}$, i.e.

$$\sum_{i=0}^{k+1}\sum_{j=0}^{k}(-1)^{i+j}F_{n,j}\circ F_{n+1,i}=0$$

we have for $m=k+1$,

$$\begin{aligned}\sum_{i=0}^{k+1+1}\sum_{j=0}^{k+1}(-1)^{i+j}F_{n,j}\circ F_{n+1,i}=&\sum_{i=0}^{k+1}\sum_{j=0}^{k}(-1)^{i+j}F_{n,j}\circ F_{n+1,i}\\&+\sum_{j=0}^{k}(-1)^{k+2+j}F_{n,j}\circ F_{n+1,k+2}\\&+\sum_{i=0}^{k}(-1)^{k+1+i}F_{n,k+1}\circ F_{n+1,i}\\&+(-1)^{k+1+k+1}F_{n,k+1}\circ F_{n+1,k+1}+(-1)^{k+1+k+2}F_{n,k+1}\circ F_{n+1,k+2}\end{aligned}$$

Notice that the first term is the double sum that is 0 by the induction hypothesis. The second term is obtained by setting $i=k+2$, the third term is obtained by setting $j=k+1$, and adding the last two terms gives everything we need for the new double sum in the case of $m=k+1$.

For the last two terms, the sum is 0 because $F_{n,k+1}\circ F_{n+1,k+1}=F_{n,k+1}\circ F_{n+1,k+2}$ by applying Lemma 2.3, and of course $-1^{2k+2}=1$ and $-1^{2k+3}=-1$. □

We are remained to prove that

$$\sum_{j=0}^{k}(-1)^{k+2+j}F_{n,j}\circ F_{n+1,k+2}+\sum_{i=0}^{k}(-1)^{k+1+i}F_{n,k+1}\circ F_{n+1,i}=0$$

By simply changing variables we can combine the two sums as

$$\sum_{i=0}^{k} (-1)^{k+2+i} F_{n,i} \circ F_{n+1,k+2} + (-1)^{k+1+i} F_{n,k+1} \circ F_{n+1,i}$$

Note that as we are summing for i from 0 to k, we always have $i < k+1$, hence we can apply Lemma 2.3 to obtain $F_{n,i} \circ F_{n+1,k+2} = F_{n,k+1} \circ F_{n+1,i}$. Therefore, we have

$$\begin{aligned}
&\sum_{j=0}^{k} (-1)^{k+2+j} F_{n,j} \circ F_{n+1,k+2} + \sum_{i=0}^{k} (-1)^{k+1+i} F_{n,k+1} \circ F_{n+1,i} \\
&= \sum_{i=0}^{k} (-1)^{k+2+i} F_{n,i} \circ F_{n+1,k+2} + (-1)^{k+1+i} F_{n,k+1} \circ F_{n+1,i} \\
&= \sum_{i=0}^{k} (-1)^{k+2+i} F_{n,i} \circ F_{n+1,k+2} + (-1)^{k+1+i} F_{n,i} \circ F_{n+1,k+2} \\
&= \sum_{i=0}^{k} ((-1)^{k+2+i} + (-1)^{k+1+i}) F_{n,i} \circ F_{n+1,k+2} = 0
\end{aligned}$$

as $(-1)^i + (-1)^{i+1} = 0 \ \forall i \in \mathbb{N}$. We are done.

Remark 3: Notice that in the statement of Lemma 2.3, the numbers j, k are independent of the number n. This allows us to give the general from of Lemma 2.4 as stated above, where n and m can be chosen independently. If one is to avoid the case mentioned in *Remark 1* of Section 2.2, one should limit the choice of j, k to $j, k < N$ in Lemma 2.3 and correspondingly $m < n$ in Lemma 2.4. Actually, we will not be using this general form of Lemma 2.4, but only the case $m = n$.

Remark 4: Although the proof above using induction is rather easy to construct and understand, it is, on the contrary, rather complex to achieve in Lean. Firstly, by setting $m = n$, the induction step will change n to $n + 1$, and hence causing much more complexity. Secondly, there will problems in Lean when dealing with a general number n, which will be discussed further in Lean sections. Thirdly, the sums, especially double sums, are not quite well-established in Lean, as some seemingly trivial concepts (changing variables, combining sums, splitting sums, etc.) cannot be used unless you prove them first.

Remark 5: As for any function, surely same inputs yields same outputs, We can also integrate both Lemmas in this section with a function, giving a result about some double sum of the function evaluating at certain points. Namely, we can use the definition of an n-cochain and obtain the following Lemma.

Lemma 2.5. *Given a group G, a G-module M, a natural number $n \in \mathbb{N}$, an n-cochain $\phi : G^n \to M$, for any natural number $m \in \mathbb{N}$ we have*

$$\sum_{i=0}^{m+1} \sum_{j=0}^{m} (-1)^{i+j} (\phi \circ F_{n,j} \circ F_{n+1,i}) = 0$$

where 0 here is the zero map.

Proof. Notice that for $j, k \in \mathbb{N}$, with $j \leq k$, we have

$$\phi \circ F_{n,j} \circ F_{n+1,k+1} = \phi \circ F_{n,k} \circ F_{n+1,j}$$

simply by Lemma 2.3. We can proceed by induction analogue to the proof of Lemma 2.4. □

2.4 The Differential d^n

Definition 2.6. *Given a group G, a natural number $n \in \mathbb{N}$ and another natural number $j < n$, we can define another contraction map F_{first}, from $G^{(n+1)}$ to G^n the simply deletes the first entry, as follow,*

$$F_{first}(g_0, g_1, \cdots, g_n) = (g_1, g_2, \cdots, g_n)$$

or equivalently

$$F_{first}(g_0, g_1, \cdots, g_n) = (a_0, a_1, \cdots, a_{n-1})$$

where $\forall i \leq n-1$, $a_i = g_{i+1}$.

Definition 2.7. *Given a group G, a G-module M, for any natural number $n \in \mathbb{N}$, we define the map $d^n(G, M)$ (denoted as d^n when G, M are clear from the context) from n-cochain to $(n+1)$-cochain as:*

For any n-cochain $\phi : G^n \to M$,

$$\begin{aligned}(d^n \circ \phi)(g_0, g_1, \cdots, g_n) = {} & g_0 \cdot \phi(g_1, g_2, \cdots, g_n) \\ & + \sum_{j=0}^{n-1} (-1)^{(j+1)} \phi(g_0, g_1, \cdots, g_{j-1}, g_j g_{j+1}, g_{j+2}, g_{j+3}, \cdots, g_n) \\ & + (-1)^n \phi(g_0, g_1, \cdots, g_{n-1})\end{aligned}$$

where the operation $g \cdot m$ for $g \in G$ and $m \in M$ is the scalar multiplication.

In light of the contraction maps $F_{n,j}$ defined earlier and F_{first} above, we can simply this definition to

$$\begin{aligned}(d^n \circ \phi)(g_0, g_1, \cdots, g_n) = {} & g_0 \cdot \phi(F_{\text{first}}(g_0, g_1, \cdots, g_n)) \\ & + \sum_{j=0}^{n-1} (-1)^{(j+1)} \phi(F_{n,j}(g_0, g_1, \cdots, g_n)) \\ & + (-1)^n \phi(F_{n,n}(g_0, g_1, \cdots, g_n))\end{aligned}$$

Notice that last two terms can be further combined due to the fact that our contraction map $F_{n,j}$ is well defined for $j = n$ as mentioned in *Remark 1* in Section 2.2. Hence, we have the following definition of the differential d^n, of which we will normally use unless otherwise

stated.

$$(d^n \circ \phi)(g_0, g_1, \cdots, g_n) = g_0 \cdot \phi(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + \sum_{j=0}^{n} (-1)^{(j+1)} \phi(F_{n,j}(g_0, g_1, \cdots, g_n))$$

2.5 $d^{n+1} \circ d^n = 0$

Theorem 2.8. *Given a group G, a G-module M, for any natural number $n \in \mathbb{N}$, any n-cochain $\phi : G^n \to M$, we have*

$$(d^{n+1} \circ d^n) \circ \phi = 0$$

where 0 here is the zero map.

Proof. Using the definition of d^n, we have

$$\begin{aligned}
&((d^{n+1} \circ d^n)) \circ \phi)(g_0, g_1, \cdots, g_{n+1}) \\
&= g_0 \cdot (d^n \circ \phi)(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1})) + \sum_{j=0}^{n+1} (-1)^{(j+1)} (d^n \circ \phi)(F_{n+1,j}(g_0, g_1, \cdots, g_n)) \\
&= g_0 g_1 \cdot \phi\Big(F_{\text{first}}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1}))\Big) := A \\
&+ g_0 \cdot \sum_{i=0}^{n} \left\{ (-1)^{i+1} \phi(F_{n,i}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1}))) \right\} := B \\
&+ \sum_{j=1}^{n+1} \left\{ (-1)^{(j+1)} \left(g_0 \cdot \phi\Big(F_{\text{first}}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\Big) \right) \right\} := C \\
&+ (-1) \left(g_0 g_1 \cdot \phi\Big(F_{\text{first}}(F_{n+1,0}(g_0, g_1, \cdots, g_{n+1}))\Big) \right) := D \\
&+ \sum_{j=0}^{n+1} \left\{ (-1)^{(j+1)} \left(\sum_{i=0}^{n} (-1)^{(i+1)} \phi\big(F_{n,i}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\big) \right) \right\} := E
\end{aligned}$$

To explain this long equation, we label the 5 terms, obtained after expanding through the definition, with A to E as above, so that

$$((d^{n+1} \circ d^n)) \circ \phi)(g_0, g_1, \cdots, g_{n+1}) = A + B + C + D + E$$

Notice that

- A and B are simply the result of applying definition to the term

$$g_0 \cdot (d^n \circ \phi)(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1}))$$

- The other term (in the line right above expression A)

$$\sum_{j=0}^{n+1} (-1)^{(j+1)} (d^n \circ \phi)(F_{n+1,j}(g_0, g_1, \cdots, g_n))$$

will also gives two terms after applying definition of d^n. One of which will be a double sum, which is indeed E.

- The other term is of course $C + D$. However, since according to the definition, we need to separate into the two expressions given, as the first entry of $F_{n+1,j}(g_0, g_1, \cdots, g_n)$ depends on the value of j. Explicitly,
  - If $j = 0$, the first entry of $F_{n+1,j}(g_0, g_1, \cdots, g_n)$ is $g_0 g_1$, this gives D.
  - If $j \neq 0$, the first entry of $F_{n+1,j}(g_0, g_1, \cdots, g_n)$ is g_0, this gives C.

Our goal now is simply to prove $A + B + C + D + E = 0$. Consider E first,

$$\begin{aligned} E &= \sum_{j=0}^{n+1} \left\{ (-1)^{(j+1)} \left(\sum_{i=0}^{n} (-1)^{(i+1)} \phi\big(F_{n,i}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\big) \right) \right\} \\ &= \sum_{j=0}^{n+1} \sum_{i=0}^{n} \left\{ (-1)^{(j+1+i+1)} \phi\big(F_{n,i}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\big) \right\} \\ &= \sum_{j=0}^{n+1} \sum_{i=0}^{n} \left\{ (-1)^{(j+i)} \phi\big(F_{n,i}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\big) \right\} \end{aligned}$$

Observe this is exactly the double sum in Lemma 2.5 by setting $m = n$. Hence, we apply the Lemma and obtain $E = 0$.

Consider A and D. Notice that

$$F_{\text{first}}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1})) = (g_2, g_3, \cdots, g_{n+1}) = F_{\text{first}}(F_{n+1,0}(g_0, g_1, \cdots, g_{n+1}))$$

One simply deletes the first entry and then deletes again the first entry of the resulting sequence. The other composes the first two entries and then deletes this product afterwards. Both process indeed yields the same result. Hence, we have

$$A = g_0 g_1 \cdot \phi\Big(F_{\text{first}}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1}))\Big) = g_0 g_1 \cdot \phi\Big(F_{\text{first}}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1}))\Big) = -D$$

which gives $A + D = 0$. Hence remain to show that $B + C = 0$.

Notice that if we have the following statement,

$$F_{n,j}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1})) = F_{\text{first}}(F_{n+1,j+1}(g_0, g_1, \cdots, g_{n+1}))$$

for $0 \leq j \leq n$, then we will have

$$\begin{aligned}
C &= \sum_{j=1}^{n+1} \left\{ (-1)^{(j+1)} \left(g_0 \cdot \phi\Big(F_{\text{first}}(F_{n+1,j}(g_0, g_1, \cdots, g_{n+1})) \Big) \right) \right\} \\
&= \sum_{i=0}^{n} \left\{ (-1)^{(i+1+1)} \left(g_0 \cdot \phi\Big(F_{\text{first}}(F_{n+1,i+1}(g_0, g_1, \cdots, g_{n+1})) \Big) \right) \right\} \\
&= g_0 \cdot \left(\sum_{i=1}^{n} \left\{ (-1)^{i} \phi\Big(F_{\text{first}}(F_{n+1,i+1}(g_0, g_1, \cdots, g_{n+1})) \Big) \right\} \right) \\
&= (-1) g_0 \cdot \sum_{i=0}^{n} \left\{ (-1)^{i+1} \phi(F_{n,i}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1}))) \right\} = -B \\
&\implies B + C = 0 \implies A + B + C + D + E = 0 \\
&\implies (d^{n+1} \circ d^n) \circ \phi = 0
\end{aligned}$$

Hence, we remain to show the statement above, as follow

$$F_{n,j}(F_{\text{first}}(g_0, g_1, \cdots, g_{n+1})) = F_{n,j}(g_1, g_2, \cdots, g_{n+1}) = (a_0, a_1, \cdots, a_{n-1})$$

where

$$\forall i \leq n-1,\ a_i = \begin{cases} g_{i+1} & i < j \\ g_{j+1} g_{j+2} & i = j \\ g_{i+2} & i > j \end{cases}$$

Similarly,

$$F_{\text{first}}(F_{n+1,j+1}(g_0, g_1, \cdots, g_{n+1})) = (b_0, b_1, \cdots, b_{n-1})$$

where

$$\forall i \leq n-1,\ b_i = a_{i+1}$$

and

$$\forall i \leq n,\ a_i = \begin{cases} g_i & i < j+1 \\ g_j g_{j+1} & i = j+1 \\ g_{i+1} & i > j+1 \end{cases} \implies b_i = \begin{cases} g_{i+1} & i < j \\ g_{j+1} g_{j+2} & i = j \\ g_{i+2} & i > j \end{cases}$$

Hence the two are same and we are done. □

2.6 d^n is a Group Homomorphism

It is not too difficult to check that d^n as defined before is actually a group homomorphism.

Lemma 2.9. *The map d^n from n-cochain to $(n+1)$-cochain is an (additive) group homomorphism.*

Proof. We need to show $d^n \circ 0 = 0$,

$$d^n \circ 0 = g_0 \cdot 0 + \sum_{j=0}^{n} (-1)^{(j+1)} 0 = 0$$

and for all n-cochains a, b, we have $d^n \circ (a+b) = d^n \circ a + d^n \circ b$.

$$\begin{aligned}
&(d^n \circ (a+b))(g_0, g_1, \cdots, g_n) \\
&= g_0 \cdot (a+b)(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + \sum_{j=0}^{n} (-1)^{(j+1)} (a+b)(F_{n,j}(g_0, g_1, \cdots, g_n)) \\
&= g_0 \cdot a(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + g_0 \cdot b(F_{\text{first}}(g_0, g_1, \cdots, g_n)) \\
&+ \sum_{j=0}^{n} \left\{ (-1)^{(j+1)} (a(F_{n,j}(g_0, g_1, \cdots, g_n)) + b(F_{n,j}(g_0, g_1, \cdots, g_n))) \right\} \\
&= g_0 \cdot a(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + g_0 \cdot b(F_{\text{first}}(g_0, g_1, \cdots, g_n)) \\
&+ \sum_{j=0}^{n} (-1)^{(j+1)} a(F_{n,j}(g_0, g_1, \cdots, g_n)) + \sum_{j=0}^{n} (-1)^{(j+1)} b(F_{n,j}(g_0, g_1, \cdots, g_n)) \\
&= \left\{ g_0 \cdot a(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + \sum_{j=0}^{n} (-1)^{(j+1)} a(F_{n,j}(g_0, g_1, \cdots, g_n)) \right\} \\
&+ \left\{ g_0 \cdot b(F_{\text{first}}(g_0, g_1, \cdots, g_n)) + \sum_{j=0}^{n} (-1)^{(j+1)} b(F_{n,j}(g_0, g_1, \cdots, g_n)) \right\} \\
&= (d^n \circ a + d^n \circ b)(g_0, g_1, \cdots, g_n)
\end{aligned}$$

□

2.7 Cocycle

Since $d^n(G, M)$ is a group homomorphism by Lemma 2.9, it is sensible to define the notion of kernel to d^n.

Definition 2.10. *Given a group G, a G-module M, for any natural number $n \in \mathbb{N}$, we define $\underline{n\text{-cocycle}}$, denoted as $Z^n(G, M)$, as the kernel of the map d^{n+1}. That is,*

$$Z^n(G, M) = Ker(d^{n+1}) = \left\{ \phi : G^{n+1} \to M : d^{n+1}(G, M) \circ \phi = 0 \right\}$$

2.8 Coboundary

Since $d^n(G, M)$ is a group homomorphism by Lemma 2.9, it is sensible to define the notion of image(range) to d^n.

Definition 2.11. *Given a group G, a G-module M, for any natural number $n \in \mathbb{N}$, we*

define n-coboundary, *denoted as* $B^n(G,M)$, *as the range of the map* d^n. *That is,*

$$B^n(G,M) = Im(d^n) = \left\{\phi : G^{n+1} \to M : \exists\theta \ \in G^n \to M \text{ such that } d^n(G,M)\circ\theta = \phi\right\}$$

2.9 Cohomology

Theorem 2.8 actually tells that

$$\forall n \in \mathbb{N}, Im(d^n) \subseteq Ker(d^{n+1})$$

That is,

$$B^n(G,M) \subseteq Z^n(G,M)$$

Hence, we can define the quotient group $Z^n(G,M)/B^n(G,M)$.

Definition 2.12. *Given a group* G, *a* G*-module* M, *for any natural number* $n \in \mathbb{N}$, *we define* n-cohomology, *denoted as* $H^n(G,M)$, *as the quotient of* n*-cocycle over* n*-coboundary. That is,*

$$H^n(G,M) = Z^n(G,M)/B^n(G,M)$$

2.10 Long Exact Sequence of Group Cohomology

In order to properly states the theorem about the long exact sequence of group cohomology, we need a list of important definitions regarding a complex first.

Definition 2.13. *A* complex *is a sequence of abelian groups,* $A^0, A^1, A^2, \cdots$, *connected by differentials* $d^n : A^n \to A^{n+1}$ *such that* $d^{n+1}\circ d^n = 0$. *It can be written as*

$$A^0 \xrightarrow{d^0} A^1 \xrightarrow{d^1} A^2 \xrightarrow{d^2} \cdots$$

Notice that we can define A^n to be the group of n-cochains and d^n to the differentials we defined before. Hence for a group G and a G-module M, we can obtain a cochain complex.

Definition 2.14. *Given two complexes* A *and* B, *a* morphism, $f : A \to B$, *is a sequence of homomorphisms* $f^n : A^n \to B^n$ *which commutes with the differentials* d^n. *That is, it gives the following commutative diagram,*

$$\begin{array}{ccccccccccc}
\cdots & \xrightarrow{d_A^{n-2}} & A^{n-1} & \xrightarrow{d_A^{n-1}} & A^n & \xrightarrow{d_A^n} & A^{n+1} & \xrightarrow{d_A^{n+1}} & \cdots \\
 & & \downarrow{\scriptstyle f^{n-1}} & & \downarrow{\scriptstyle f^n} & & \downarrow{\scriptstyle f^{n+1}} & & \\
\cdots & \xrightarrow{d_A^{n-2}} & B^{n-1} & \xrightarrow{d_B^{n-1}} & B^n & \xrightarrow{d_B^n} & B^{n+1} & \xrightarrow{d_B^{n+1}} & \cdots
\end{array}$$

For a given G-module homomorphism $h : M \to N$ where M, N are G-modules, it is not difficult to define naturally induced homomorphisms f^n that maps an n-cochain of M to an n-cochain of N for all $n \in \mathbb{N}$. That is for $n \in \mathbb{N}$, $\phi : G^n \to M$,

$$(f^n \circ \phi))(g_0, g_1, \cdots g_{n-1})) = h(\phi(g_0, g_1, \cdots g_{n-1}))$$

Definition 2.15. *Given a sequence of groups* $\{G_i\}_{i=0}^{n+2}$ *and group homomorphisms* $\{f_i\}_{i=0}^{n+1}$ *for* $n \in \mathbb{N}$*, written as*

$$G_0 \xrightarrow{f_0} G_1 \xrightarrow{f_1} G_2 \xrightarrow{f_2} \cdots \xrightarrow{f_{n+1}} G_{n+2}$$

This sequence is exact if for all $k \leq n$*,*

$$Im(f_k) = Ker(f_{k+1})$$

As a special case of the above definition of exactness, we have a definition for a short exact sequence as

Definition 2.16. *The sequence*

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

is a short exact sequence if

- $Ker(f) = \{0\}$,
- $Im(g) = C$,
- $Im(f) = Ker(g)$.

Definition 2.17. *Given complexes* A, B, C*, and morphisms* $f : A \to B$ *and* $g : B \to C$*, we say*

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

is exact is for all $n \in \mathbb{N}$*, we have the following short exact sequence*

$$0 \longrightarrow A^n \xrightarrow{f^n} B^n \xrightarrow{g^n} C^n \longrightarrow 0$$

Theorem 2.18. *Given a group* G*, and* G*-modules* M, N, L*. Suppose we have* G*-module homomorphisms* $f : M \to N$ *and* $g : N \to L$ *such that we have the following short exact sequence*

$$0 \longrightarrow M \xrightarrow{f} N \xrightarrow{g} L \longrightarrow 0$$

Then there is an induced long exact sequence of cohomology,

$$0 \longrightarrow H^0(G, M) \longrightarrow H^0(G, N) \longrightarrow H^0(G, L) \longrightarrow H^1(G, M) \longrightarrow H^1(G, N) \longrightarrow H^1(G, L) \longrightarrow H^2(G, M) \longrightarrow \cdots$$

Proof. Omitted. Not in the scope of this project. □

3 An Introduction to Lean Theorem Prover

This report should not be treated as a thorough tutorial for new users of Lean (it surely is not). It only provides a quick tutorial to Lean such that the reader will be able to understand the second part of this report, and the online repository in Lean for this project. For individuals interested in studying Lean, on should consult online resources, some of which will be included in the references of this report.

3.1 What is Lean?

One can think Lean just as a programming language, on which you write write codes, functions, and perform computations. While others like Python, MATLAB, and R provides tools for explicit calculation, Lean focuses on theorem proving. One can state definitions, state theorems and prove those theorems in Lean, using given definitions, axioms and any formulated statements that have been already proved in Lean.

Lean creates an interactive platform for users in terms of theorem proving. Everything you entered in Lean is associated to a certain type. For anything you want to define in Lean (a definition, a theorem, etc.), Lean will perform a type check to ensure fundamental validity of the given statement. For example, for any equation, the types of both sides must match. When you want to prove a certain statement, Lean provides a messages summarizing your current stage including information on any variables you have, any assumptions made, and the current goal or goals (to prove). As you proceed with your logic and reasoning in the proof, Lean updates the message at each step, showing either the updated goal, or an error message indicating that you did something wrong or simply Lean cannot understand with its given knowledge.

I would prefer to think Lean just as digital mathematics student. As you put more definitions and theorems in Lean, it 'learns' those concepts. As the knowledge of Lean builds up, it then can use it to check every line of command as you proof a statement. The differences between Lean and a normal human mathematics learner are probably

- ***'Non-triviality'***: There is nothing as 'trivial' for Lean unless it is exactly the definition given in Lean. On the other hands, 'trivial' in a human sense can refer to simple proofs which has many underlying principles involved. Lean would require you (for the most of time) to explicitly list these principles out in the correct order.

- ***'Dependence'***: Lean cannot create something new on its own (at least for now). You have to give everything it needs to understand a new statement or a new step in a proof. In the case of stating a definition of theorem, this means you have to carefully define all variables and any assumptions. In terms of a proof, this means that any lemmas or claims must be proved first before they can be used. Of course, you could also 'teach' Lean to find some answers itself by writing algorithms.

- ***'Explicitness'***: Lean cannot understand certain concepts that a human can. Most of these concepts involves the process of certain 'interpretation'. For example, we all understand what is meant by $1, 2, 3, \cdots, n$, but it you type this to Lean, it will not be able to tell what $\cdots$ means. Due to this fact, any definitions with any presence of $\cdots$ will face difficulties if one tries to put it in Lean.

- ***'Type-specificity'***: Lean categorizes everything explicitly while human mixes notations sometimes. For example, 2 to a human eye is a number, one rarely distinguishes between $2 \in \mathbb{N}$ and $2 \in \mathbb{R}$, but to Lean, there is a huge difference as $\mathbb{N}$ and $\mathbb{R}$ are defined

very differently. As such, one has to be very careful with anything you define in Lean, in the sense that you have to be clear which type it should fall into.

- There are many other notable properties of Lean. The following ones are listed for your interests, but will not be discussed in much details as they will not be highly relevant to this report.
  - One cannot use any means of proofs that is not programmable in Lean. For example, graphical method is unlikely to work as there is no graphical interface in Lean yet.
  - Lean does not make mistakes or confusions between concepts while human surely will.
  - Lean also do not have any memory issue. Everything it has 'learned' is present somewhere and you can always use it given that you state it correctly and that you are in correct circumstances to use.
  - And many more...

Note that we gave 'names' to the first four properties of Lean above. One should keep these features of Lean in mind as you proceed to the next section where we start to define notions and construct proofs in Lean. These four names will extensively appear in the next section as we proceed with the discussion, because indeed the way we have chosen highly depends on these properties.

Due to these differences, continuing this project of group cohomology in Lean, is somewhat similar, and at the same time very different, to teaching a mathematics student group cohomology. One has to be very explicit in giving statements (due to ***'Explicitness'***), and very careful and logically concise when constructing proofs(due to ***'Type-Specificity'***). You may imagine that you cannot jump any steps in a proof(due to ***'Non-triviality'***).

3.2 Conventions in Lean

3.2.1 Basic Set Up

A general definition in Lean takes the form of:

```
def name (variables: Type)[properties of variables](h: assumptions): Type
   of defined term := explicit definition
```

All the terms are self-explanatory. One note is that for a definition, it is acceptable to just give the type without an explicit expression.

A general lemma, theorem or example takes the similar form as that of definition,

```
theorem/lemma/example name (variables: Type)[properties of variables](h:
   assumptions): Statement := Proof
```

A proof must be provided for a lemma, theorem or example. However, one may replace it with `sorry` to fill in later.

One important note on a general definition/theorem/lemma/example, regarding on the inputs, is that

- For variables closed in (), it has to be provided as inputs.

- For variables closed in {}, it means this variable can be guessed by Lean from other inputs and must not be included in the inputs.

A structure is a special kind of definition that takes more than one items to define a notion. Such item could be an expression, or a proof of certain property. Similar to that of a definition, you have to provide variables and properties of variables needed.

A namespace creates a workspace where you can define some variables. After doing so, they do not have to appear in any following definition/lemma/theorem/example's statement, within the same namespace, again. In addition, the name of the definition/lemma/theorem/example can be shortened since when quoted anywhere, is named as '`the name of the namespace`'. '`the name of the theorem`'. One can close a namespace by `end +` '`the name of the namespace`'.

Similar to other programming languages, one can import other Lean files to the one you are working one by `import` '`name of the file`', normally at the beginning of your file. Once imported, you can use all the definitions/theorems/lemmas in the file, and only imported ones can be used. It is also important that files cannot cross import. If A imports B then B cannot import A.

A highly relevant notion to this report is the set `finset, fin(n)` in Lean which is the set of first n elements of natural numbers starting from 0 inclusively. By convention, if one makes a subtraction that results in less than 0, this will be set to 0. Hence, it is important to avoid using subtraction when using `fin(n)`, as any subtraction can cause unwanted results by the above convention.

By definition, the inequality in Lean means the negation of equality. That is, that equality implies false.

3.2.2 Functions

To define a function, one uses λ in Lean. `λ k, k^2` means the function that sends k to k^2. Generality, any expression involving k can be used, as well as another function. Note that there is also Π for defining functions, but we will not be using it.

A piece-wise function can be defined using, `if A then B else C`, where A is the condition under this if, B is the expression, and C is the expression of other cases apart from condition A. C can be incorporated with again an if-then-else set up. The short form of this is `ite(A)(B)(C)`.

Different to the usual mathematical set up, where a function takes all its inputs at once, functions in Lean can take arguments successively. For example, it is possible to have a function as `f: A → B → C` where A, B, C are types. After inserting an element of type $a \in A$, one gets `f a` as a function `B → C`. One can easily transfer between a mathematical multi-variable function to a function in this successive form.

Variables/inputs to a function, or the output of a function can have multiple entries where each entries can be expressions or a proof of certain property. For example, an element, `i ∈ fin(n)`, has two entries. The first is its value, which can be extracted by `i.1` or `i.val`. The second is a proof such that this element is indeed in this set. That is, it is a proof that `i.val < n`.

3.2.3 Bundled Compared to Un-bundled

A bundled file refers to a set of notions, including relevant definitions, theorems, lemmas, etc. grouped under the same namespace. On the contrary, un-bundled refers to any scattered ones. It is important for notions to be bundled for more applicable and convenient uses. First of all, any properties, theorems and lemmas of a given namespace can be easily cited for use. For example, with a bundled file for an additive group, say, once you define something to be an additive group, you can easily access the properties of groups proved in the bundle. In addition, you will automatically be granted with all the `instance` in the bundle. On the contrary, there is limited applicability when you only have a un-bundled statement like `is_group` to check whether it is a group. On the other hand, as types of variables are of significant importance in Lean, bundled files ensure proper applicability of statements by a much better structure on variables.

You will see many occasions of re-bundling in this report as many concepts required for this project were un-bundled in the Lean's math library.

3.3 Tactics in Lean

There are mainly two ways to provide a proof in Lean. The one we will generally use in this project is the tactic mode proof. One can enter the tactic mode by `begin` and ends simply by `end`. Once entered, you will be able to use a list of powerful tactics for you to solve your goal. From the very beginning of the proof, and as you move each step by commanding a tactic, Lean provides you with all information about the current state of the proof, including all the variables, assumptions and the current goals. Once this message displays `goals accomplished`, your proof is complete.

Here is a list of tactics highly relevant to this report. Some are clear by demonstrating the changes made to the goal before and after applying the tactic, as shown below in the table.

Tactic Name	Brief description	Goal before	Goal after
`funext`	adds an input to both side	`f = g`	`f x = g x`
`rw`	replaces `x` using equation `x = a`	`x > 0`	`a > 0`
`rw <-`	replaces `x` using equation `a = x`	`x > 0`	`a > 0`
`norm_num`	performs simple arithmetic calculations	`x = 1 + 1`	`x = 2`
`intro`	introduces an assumption	`P → Q`	`Q`

Others are explained in words as follow,

- `unfold`: Expands a definition as it is given in Lean.
- `simp only`: Simplifies the expression using the given set of theorems.
- `dsimp`: Simplifies the expression using definitions and basic conventions. For example, `⟨a, _⟩.1 = 0` will be simplifies to `a = 0` as by convention `.1` takes the first entry.
- `apply`: Applies a theorem to change the goal.
- `convert`: Converts the goal to a given form.
- `rwa`: Rewrites using the assumptions.

- `exact`: Claims that the goal is exactly the result of applying some theorem. This tactic can only be used for the last step in a proof.
- `show`: Claims that the goal is equivalent to a given form.
- `split`: Splits the goal into all separate parts (creating multiple goals) and considers each part individually.
- `cases`: Considers all possible cases and tackles each case separately. In particular, one can uses `cases` for an assumption of the form `∃ a, f(a)` to eliminate `∃`.
- `if_pos`: For an `ite (A)(B)(C)` situation, it checks that `(A)` is satisfies and change the goal to `(B)`.
- `if_neg`: For an `ite (A)(B)(C)` situation, it checks that `(A)` is false and change the goal to `(c)`.
- `have`: Makes an intermediate hypothesis, of which you have to provides a proof.
- `exfalso`: Changes the goal to `false`.
- `refl`: Solves the goal by reflectivity.
- `linarith`: Solves a goal of `false` by linear arithmetic on hypotheses.

This is not the full list of tactics available in Lean. It is also notable that some tactics can also be applied to hypothesis by `... at h`. For example, `rw T at h` means rewriting the hypothesis `h` using theorem `T`.

4 Group Cohomology in Lean

The Lean repository for this entire project is available online at `https://github.com/Shenyang1995/M4R`. You may wish to refer the these codes as you proceed in this section, as we will not provide a copy of all the codes here.

The structure of this section is designed to be similar to that of Section 2 due to the correspondence between content. It is suggested to familiarize yourself with definitions and proofs in Section 2.

4.1 Cochain

According to our definition of n-cochain, it is just a map from $G^n \to M$. However, this does not help us in giving a definition in Lean. This is as mentioned, a problem in Lean with the general number $n \in \mathbb{N}$. Of course, for $n = 1$, $n = 2$ etc, one can easily states the definitions explicitly as $f : G \to M$ or $f : G \times G \to M$. It is not possible to just write $f : G \times G \times \cdots \times G(\text{n times}) \to M$ in Lean as it will not understand $\cdots$. Hence, we need to turn to alternatives of the definition of a n-cochain, due to ***'Explicitness'*** of Lean.

Define $X_n \subset \mathbb{N}$ to be the set of first n elements of $\mathbb{N}$. For example, $X_5 = \{0, 1, 2, 3, 4\}$. One possibility is to consider $(g_0, g_1, \cdots, g_{n-1}) \in G^n$ as a map f from X_n to G such that $f(i) = g_i$ for all $i \in X_n$. As such, for each $n \in \mathbb{N}$, we can explicitly define the set X_n, and in consequence define the map f from X_n to G and n-cochain. It is also a standard mathematics result such that the group of all such fs is isomorphic to $G^{|X_n|} = G^n$.

This set, X_n, is already defined in Lean as `fin(n)`, which is indeed the set of first n elements of $\mathbb{N}$. We have the definition of an n-cochain as follow,

```
def cochain(n: ℕ)(G : Type*)[Group G] (M : Type*)[add_comm_group
   M][G_module G M]:= (fin n → G) → M
```

4.2 The Contraction map $F_{n,j}$

4.2.1 Definition

Due to ***'Explicitness'*** of Lean, one directly sees that the explicit definition of this contraction map $F_{n,j}$ we gave works better in Lean than the other definition which involves the use of '$\cdots$' that requires interpretation.

Notice that it is defined as a piece-wise function, which is achievable in Lean as follow,

```
def F{n: ℕ}(j: ℕ){G: Type*}[Group G] (g: fin(n+1) → G) :(fin n → G)
:= λ k,  if k.val < j then g ⟨k.val, lt_trans k.2 $ lt_add_one _⟩ else
         if k.val=j then g ⟨k.val, lt_trans k.2 $ lt_add_one _⟩ *g ⟨k.val+1,
   add_lt_add_right k.2 1⟩
         else g  ⟨k.val+1, add_lt_add_right k.2 1⟩
```

As mentioned before, each element of `fin(n)` consists of a value and a proof. As an example, in the above definition, for the case $k > j$, in addition to g_{k+1}, which is the value corresponding to that of the explicit definition given in Section 2, denoted above as `g ⟨k.val+1,...`, we have to also provide a proof that $k + 1 < n + 1$.

An easy way is to use `k ∈ fin(n)`, we have `k.2` is exactly the proof that $k < n$. The theorem stating that $k < n \implies k + c < n + c$ for c having the same type as k and n is

indeed called `add_lt_add_right` in Lean. This theorem takes 2 inputs in Lean, the first one is the proof that $k < n$, which in our case if simply `k.2`. The second one is the input c in the statement of the theorem which is 1 in this case, hence giving the above Lean code.

Remark: Note that the name of the theorem is chosen such that it relates to the content of the theorem for easy uses. `add_lt_add_right` stands for additive inequality added to the right. Also, the requirement that c has the same type as k and n is important due to ***'Type-Specificity'***. For example, if $k, n \in \mathbb{N}$ and $c \in \mathbb{R}$, then unfortunately `add_lt_add_right` cannot be applied!

4.2.2 Lemma 2.3

Following the definition, one immediately thinks of stating and proving Lemma 2.3. We first have to be careful with the inputs to function the F we defined in Lean. Notice that F takes two inputs, the number `(j: ℕ)` followed by the map `(g: fin(n+1) → G)`. Notice that `n: ℕ` and `G: type*` are not required and must not be included as they are defined using {} instead of (). As such, a standard use should be something like `F j g`.

Also, the map F outputs a map `(fin n → G)` which can be again used for F as the new g-input. However, we would see the occurrence of $n-1$ in that case as the output will be a map `(fin (n-1) → G)`. As discussed before, we should avoid the use of $n-1$ and we should then change the initial input g to F to `(g: fin(n+2) → G)`, yielding the statement as follow,

```
theorem degenerate{n:ℕ}{j:ℕ}{k:ℕ}{G : Type*}[group G](h:j≤ k)(g:fin
    (n+2)→ G):
F j (F (k+1) g) = F k (F j g):= sorry
```

Remark: Right after `:=` is where we start our proof to replace `sorry`.

In Section 2 we provided a detailed proof that is concrete enough to allow implementation in Lean. Now you can see the reasons, namely ***'Explicitness'*** and ***'Non-triviality'***, for giving a concrete proof of 2.3 even it can somehow be 'trivial'. For each `t ∈ fin(n)`, we simply consider all possibilities of t comparing with j and k to conclude in each case, both sides of the equation coincides. A general structure of the proof in Lean in presented below. (We will not go into details as the principles are exactly as that in Section 2. Some comments in the Lean code should be self-explanatory.)

```
theorem degenerate{n:ℕ}{j:ℕ}{k:ℕ}{G : Type*}[group G](h:j≤ k)(g:fin
    (n+2)→ G):
F j (F (k+1) g) = F k (F j g):=

begin
unfold F,                       -- apply definition of F
funext t,                       -- introduce t
cases t with t ht,              -- argue with cases of t
--some omitted lines of code
by_cases h1 : t < j,            -- comparing with j
--some omitted lines of code
by_cases h2:t=j,
--some omitted lines of code
by_cases h3 : t < k,            -- comparing with k
--some omitted lines of code
by_cases h4 : t = k,
--some omitted lines of code
```

```
end
```

Remark: One can practice the similar argument as before to find out what inputs we should present for the use of `theorem degenerate`, and also what should not be included. By distinguishing between {, and (, we have the only inputs in order is an assumption `(h:j≤ k)` and a function `(g:fin (n+2)→ G)`. Well, for theorems and lemmas, Lean will not be too strict about inputs. One may just try to rewrite using this theorem by giving no inputs and Lean will make a guess or ask you to provide proofs for the hypothesis afterwards.

4.3 The Result on Double Sum

As one can see from the proof of Theorem 2.8, the result on double sum that we need is actually a special case of Lemma 2.5 when $m = n$. As one may already see it really takes some effort to contract statements and proofs in Lean, we will just focus on the required results. That is, we will only prove the following statement in Lean, without proving the more general case:

Given a group G, a G-module M, a natural number $n \in \mathbb{N}$, an n-cochain $\phi : G^n \to M$, we have

$$\sum_{i=0}^{n+1}\sum_{j=0}^{n}(-1)^{i+j}(\phi \circ F_{n,j} \circ F_{n+1,i}) = 0$$

In Lean, this states as

```
theorem double_sum_zero1 (n':ℕ)(G : Type*)[group G](g:fin (n'+3)→ G)(M :
   Type*) [add_comm_group M] [G_module G M](v:cochain (n'+1) G M):
(range (n'+3)).sum(λ i, (range (n'+2)).sum(λ j, (F2 g v (i,j)))) =0:= sorry
```

Remark 1: This is our final result to prove in this section. Note we specially place $n'+1 = n$ to ensure our expression in the theorem will be of the exact form that we will use. You will see this again in the proof of $d^{n+1} \circ d^n = 0$.

Remark 2: It is also worth nothing how to express a sum in Lean using `finset`. As an example from above, it takes the form `(range n).sum(λ i, f i)` to mean $\sum_{i=0}^{n} f(i)$. Note $f(i)$ can be replaced by another sum to create a double sum.

Of course, there will many different approaches. For example, you can even think of proving the more general lemma and then apply to $m = n$ case. We assume this is not the best option. Here are some other possible methods one may naturally think of.

- Method 1: One may suggest we formalize the proofs of Lemma 2.4 and 2.5 in the case of $m = n$. However, we have discussed the limitations of this approach in Section 2.3. During the inductive step, we have to show for $k \in \mathbb{N}$,

$$\sum_{i=0}^{k+1}\sum_{j=0}^{k}(-1)^{i+j}(\phi \circ F_{k,j} \circ F_{k+1,i}) = 0 \implies \sum_{i=0}^{k+2}\sum_{j=0}^{k+1}(-1)^{i+j}(\phi \circ F_{k+1,j} \circ F_{k+2,i}) = 0$$

  Notice that the indexes of F changes from k to $k+1$, and $k+1$ to $k+2$. This will cause trouble when we try to use the inductive hypothesis in Lean, as Lean will look out for the **exact** appearance of the expression in inductive hypothesis, and will fail due to this

index change. As a result, we need to provide much more claims and lemmas for this to work. This then does not seem to be the best option.

- Method 2: When one tries to prove this by hand, it is quite straight forward as we can just explicitly write out all the terms in the sum. By trying small examples of n, one can easily see that we can pair up all the terms in a way such that for each pair, we can apply Lemma 2.3 and the sum of each pair will be 0 due to the indices of -1. For a general n, we can simply draw a graph or a table to discuss that the sum can be partitioned into unions of such disjoint pairs. However, from this argument itself, one can foreseen difficulties of employing the method in Lean due to ***'Explicitness'*** and Lean's lack of graphical interface. One has to think of an explicit way of describing this pairing, which is indeed the involution we are going to discuss.

Actually, due to ***'Dependence'***, it would be much better to turn to any method based on some proved lemma in Lean. Unfortunately, the library in Lean for double sum is not so well established in the sense that only limited number of lemmas about double sums can be directly used. If one has to formalize all the theorems we need, it will be very troublesome and time-consuming. For example, we used some fundamental properties of a double sum, like exchanging variables, combining two sums, partitioning a sum etc. These are trivial to a well-educated mathematician, but requires proving in Lean due to ***'Non-triviality'***. As a result, we proceed by trying to convert the double sum to a single sum, and then using a proved lemma in Lean using an involution.

4.3.1 Conversion to a Single Sum

This is achieve by the following statement,

Lemma 4.1. *For any sets S_i and S_j, and an expression $T(i,j)$ formulated in terms of $i \in S_i$ and $j \in S_j$, we have*

$$\sum_{i\in S_i}\sum_{j\in S_j} T(i,j) = \sum_{(i,j)\in S_i\times S_j} T(i,j)$$

This is stated in Lean as follow,

```
lemma sum_product {α : Type u} {β : Type v} {γ : Type w}{s : finset γ} {t :
    finset α} {f : γ×α → β} :
  (s.product t).sum f = s.sum (λx, t.sum $ λy, f (x, y)) := ...
  -- some ommited proof
```

We will emit the proof since it is trivial to a human eye and also already proved in Lean for the case we need. The lemma in Lean is originally stated in a multiplicative group setting, but analogy to additive group can and will be used.

In our case, $S_i = X_{n+2} =$ `fin (n+2)` , $S_j = X_{n+1} =$ `fin (n+1)` and $T(i,j) = (-1)^{i+j}(\phi \circ F_{n,j} \circ F_{n+1,i})$ is our summand.

After conversion, we need to prove that,

$$\sum_{(i,j)\in X_{n+2}\times X_{n+1}} (-1)^{i+j}(\phi \circ F_{n,j} \circ F_{n+1,i}) = 0$$

Before moving into involution, we note that at this point in time, well, Lean is still not clear enough in the sense that when we command it to apply something that have we already proved, it looks out for the exact appearance of the expressions appeared. Even with a small difference in the expression, Lean might just fail to apply what we want. One may note that there is some slight difference between the summand in the sum above, and the expression in the `theorem degenerate`. As such, we would better sort the -1 and ϕ in and make some adjustment to make it look exactly like the summand.

4.3.2 Summand in the Double Sum

First of all, there is an important change we have to make due to this conversion. Previously, we use `F k (F j g)` for $F_{n,k} \circ F_{n+1,j}$ in Lean, which is defined by `{j:ℕ}` and `{k:ℕ}` from `(h:j≤ k)`. After the conversion, the type of input changes! The input will be some sort of the form $(j,k) \in \mathbb{N} \times \mathbb{N}$. It is vital due to ***'Type-Specificity'***, we have to give a new formulation of $F_{n,k} \circ F_{n+1,j}$. As our primary goal is to make it exactly same as the summand, we also add in the corresponding -1 and ϕ.

We have the summand as

```
def F2{n:ℕ}{G : Type*}[group G](g:fin (n+2)→ G){M : Type*} [add_comm_group
   M] [G_module G M](v:cochain n G M): ℕ × ℕ → M:=
λ j, (-1:ℤ)^(j.1+j.2)• v (F (j.2) (F (j.1) g))
```

Remark 1: Here $j \in \mathbb{N} \times \mathbb{N}$, `j.1` simply refers to the first element and `j.2` the second. Also, it is important to give the type of `(-1:ℤ)` in this definition, due to ***'Type-Specificity'***, otherwise it will fail to compile. It is okay to specify the type in this way. The dot refers to the scalar multiplication.

Remark 2: More importantly, note the inputs needed are `(g:fin (n+2)→ G)` and `(v:cochain n G M)`, so a standard formulation is `F2 g v (j, k)`, where $(j,k) \in \mathbb{N} \times \mathbb{N}$. This expression gives an element of M.

4.3.3 Rephrasing Lemma 2.3

Now we will convert `theorem degenerate` properly to ensure smooth future use. This will be an excellent example to showcase how detailed you need to go in order to complete a proof in Lean even with a tiny change. We will present working and logical thinking in each step.

Firstly, it will not be difficult to state the new lemma using the new definition, as follow,

```
theorem F2_degenerate {n:ℕ}{j:ℕ}{k:ℕ}(h:j≤ k){G : Type*}[group G](g:fin
   (n+2)→ G){M : Type*} [add_comm_group M] [G_module G M](v:cochain n G M):
F2 g v (k+1, j)+F2 g v (j, k)=0:= sorry
```

- **Step 1:** First of all, we try to expand the definition of `F2` through the tactic `unfold`. This usually comes as the first step in a proof. Unfolding often gives us rather messy equation because of how Lean stores the definitions, so it is also common to simplify the expression by using the some tactics. Normally `dsimp`, or if it is just some numerical computation, `norm_num` works as well. In this case, we used `norm_num` and at this point the goal appears to be the following,

  After commanding (in order): `unfold F2, norm_num,`

```
⊢ (-1) ^ (j + k) • v (F k (F j g)) + (-1) ^ (j + (k + 1)) • v (F j (F
    (k + 1) g)) = 0
```

- **Step 2:** We see appearances of `F k (F j g)` and `F j (F (k + 1) g)` that are identical to those of `theorem degenerate`. This is a great sign as it tells us we can apply the proved statement in this proof. We can use `rw` to replace the left-hand side of `theorem degenerate` the right-hand side, which will gives us

  After commanding (in order): `rw degenerate,`

  ```
  2 goals
  ⊢ (-1) ^ (j + k) • v (F k (F j g)) + (-1) ^ (j + (k + 1)) • v (F k (F j
      g)) = 0
  ⊢ j ≤ k
  ```

  Remark: The second goal arises because in applying `theorem degenerate`, one needs an assumption that $j \leq k$. It is okay to provide a proof afterwards. Note that in this case this is just the assumption h we have, so if we put the command as `rw degenerate h,`, this goal will not appear. However, sometimes the assumptions will not be so clear, and such a way will give us a better structured proof. You will see this in the section of involution. This feature does not contradicts to the ***'Dependence'*** of Lean as nonetheless you have to prove those assumptions to use the stated theorem.

- **Step 3:** Note now we have a common term `F k (F j g)`, and the next step will be to combine them. This is NOT automatic! There is a underlying theorem we need, which is in Lean stated as

  ```
  theorem add_smul(α : Type u) (β : Type v) [semiring α]
      [add_comm_monoid β] [semimodule α β] (r s : α) (x: β) : (r + s) •
      x = r • x + s • x
  ```

  Due to ***'Type-Specificity'***, in order to apply this theorem, one has to make sure the inputs r, s, x has certain types with certain properties. This is why we mentioned earlier that the type of (-1) is very important. Note that we actually have the expression on the RHS and we want the LHS, so the tactic is `rw <-`, as follow,

  After commanding (in order): `rw <-add_smul,`

  ```
  2 goals
  ⊢ ((-1) ^ (j + k) + (-1) ^ (j + (k + 1))) • v (F k (F j g)) = 0
  ⊢ j ≤ k
  ```

- **Step 4:** Notice `(j + (k + 1))`, we want to remove the bracket of $(k + 1)$, this is again of course trivial but actually based on a theorem which has some requirements of inputs, as associative is not always true. It is called `add_assoc` in Lean with statement of the form, ... `:a+b+c=a+(b+c)`. So we need again `rw <-`,

  After commanding (in order): `rw <-add_assoc,`

  ```
  2 goals
  ⊢ ((-1) ^ (j + k) + (-1) ^ (j + k + 1)) • v (F k (F j g)) = 0
  ⊢ j ≤ k
  ```

- **Step 5:** We can now finish by arguing any the sum of any two consecutive integer powers of -1 is zero. However, this statement itself requires proving in Lean due to

'Non-triviality'. Noting that `v (F k (F j g))` is an element of M, we state the following lemma, together with its proof and explanation included as comments in the code.

```
theorem neg_one_power(n:ℕ )(G : Type*) [group G] (M : Type*)
    [add_comm_group M] [G_module G M]
: ∀ m : M, ((-1:ℤ)^n  + (-1:ℤ)^(n+1)) • m = 0 :=
begin
intro m,                          -- bring m in to eliminate ∀ m : M
induction n with h hd,            -- Induction on n
norm_num,                         -- Case 0: just by evaluation n=0
rw nat.succ_eq_add_one,           -- Inductive case
rw pow_add (-1:ℤ ) h 1,           -- Use a^(m + n) = a^m * a^n
rw pow_add (-1:ℤ ) h (1+1),       -- Use a^(m + n) = a^m * a^n again
norm_num,                         -- Evaluate to cancel out.
end
```

Remark: One may ask why did we add m inside. This is just for simplicity as the proof with or without this m does not make much difference. We can directly apply this theorem to accomplish the goal.

- **Step 6:** Use above proved theorem (you need to be careful with the inputs) and the assumption `(h:j≤ k)` to complete the proof.

  After commanding (in order): `rw rw neg_one_power (j+k) G M, exact h,`

  `goals accomplished`

In contrast, previously we concluded this result from Lemma 2.3 by simply arguing for any function, same inputs give the same outputs, and that $(-1)^i + (-1)^{i+1} = 0\ \forall i \in \mathbb{N}$. This two-line argument turns out to be in Lean, a 6-step proof which in addition contains an induction!

This exactly demonstrates that due to this ***'Non-Triviality'***, we will not be able to jump any steps, and for each step, we need to very carefully ensure that our inputs satisfies the requirements of the underlying theorems by ***'Type-Specificity'***. In addition, this 6-step proof is a result of a small change made to a proved theorem. Hence, we will try to state any of our statements in a way that favours future uses.

This is though, just an easy example, where as things get more complex, each step will requires much more complex proofs as compared to a single rewrite of some theorem. You may now have a better idea of the challenging nature of defining group cohomology in Lean. We will not be presenting all the steps to this detail, but only elaborate on important concepts in the following sections.

On the other hand, one sees that as we follow the flow of logic, there will be at some point that we realise we need something to be proved first in order to proceed. From now on, we will focus on presenting the results, as in we will prove all the required lemma first, so that we do not jump between proofs, though the thinking process does not follow this order.

4.3.4 The Involution

With `theorem F2_degenerate` in hand, we are well-prepared for involution. We define an involution as follow,

Definition 4.2. *Define the involution map, denoted as invo, as*

$$invo : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

$$invo(j,k) = \begin{cases} (k+1, j) & j \leq k \\ (k, j-1) & otherwise \end{cases}$$

Remark: This definition is define is a way such that by performing the involution map, the output and input forms a pair (as described in Method 2 earlier) such that the sum of `F2 g v` evaluating at this pair equals to 0 by `theorem F2_degenerate`.

In Lean, this definition is simply states as

```
def invo : ℕ × ℕ → ℕ × ℕ :=
λ jk, if jk.1 ≤ jk.2 then ⟨jk.2 + 1, jk.1⟩ else ⟨jk.2, jk.1 - 1⟩
```

We prove some general facts about this involution map first. The reason is simply that we are going to use these in our final proof and ***'Dependence'***.

Lemma 4.3. *1 : For $j \leq k$, we have $invo(j,k) = (a,b)$ where $\neg a \leq b$*

2 : For $\neg j \leq k$, we have $invo(j,k) = (a,b)$ where $a \leq b$

3 : For $\neg j \leq k$, we have $j - 1 + 1 = j$

4 : For all $(j,k) \in \mathbb{N} \times \mathbb{N}$, $invo(invo(j,k)) = (j,k)$

Proof.

- By definition, $a = k+1$, $b = j$. Hence

$$j \leq k \implies k+1 > j \implies \neg j \leq k+1 \implies \neg a \leq b$$

- By definition, $a = k$, $b = j-1$. Hence

$$\neg j \leq k \implies j > k \implies k \leq j-1 \implies a \leq b$$

- This is NOT trivial as $0 - 1 = 0$ in Lean ! Hence,

$$\neg j \leq k \implies j > k \implies j > 0 \implies j - 1 + 1 = j$$

- We can just argue by cases.
  - If $j \leq k$, we have$j \leq k \implies k+1 > j \implies \neg k+1 \leq j$, Hence $invo(invo(j,k)) = invo(k+1, j) = (j, k+1-1) = (j,k)$.
  - If $\neg j \leq k$, we have$\neg j \leq k \implies j > k \implies k \leq j-1$, Hence $invo(invo(j,k)) =$

$invo(k, j-1) = (j-1+1, k) = (j, k)$ by 4.3.3 above.

□

Remark: Each one of the implies signs above in the proofs actually corresponds to one underlying fundamental arithmetic theorem which is proved and has a unique name in Lean. One just need to look out for the correct names and applied to the proof. We will just present the result without going into the details as the reasoning in clear from above. The statements are in the exact order corresponding to that of Lemma 4.3.

```
lemma invo_ineq1 {jk : ℕ × ℕ}
  (h : jk.1 ≤ jk.2) :
  ¬(invo jk).1 ≤ (invo jk).2 :=
begin
  unfold invo,
  rw if_pos h,
  exact not_le.2 (nat.lt_succ_of_le h),
end

lemma invo_ineq2 {jk : ℕ × ℕ}
  (h : ¬jk.1 ≤ jk.2) :
  (invo jk).1 ≤ (invo jk).2 :=
begin
  unfold invo,
  rw if_neg h,
  rw not_le at h,
  exact nat.pred_le_pred h,
end

lemma invo_aux {j k : ℕ} (h : ¬j ≤ k) : j - 1 + 1 = j :=
nat.succ_pred_eq_of_pos $ lt_of_le_of_lt (zero_le _) (lt_of_not_ge h)

lemma invo_invo (jk : ℕ × ℕ) : invo (invo jk) = jk :=
begin
  unfold invo,
  split_ifs,
  { exfalso, linarith},
  { ext, refl, simp},
  { ext, dsimp, exact invo_aux h, refl},
  { exfalso,
    rw not_le at h_1,
    rw nat.lt_iff_add_one_le at h_1,
    rw invo_aux h at h_1,
    linarith,}
end
```

4.3.5 Final Result on the Sum

We are ready to prove the following, Given a group G, a G-module M, a natural number $n \in \mathbb{N}$, an n-cochain $\phi : G^n \to M$, we have

$$\sum_{(i,j)\in X_{n+2}\times X_{n+1}} (-1)^{i+j}(\phi \circ F_{n,j} \circ F_{n+1,i}) = 0$$

The theorem we are going to use is stated in Lean as

```
lemma sum_involution{α : Type u} {β : Type v} {s : finset α} {f : α → β}
   :
  ∀ (g : Π a ∈ s, α)
  (h₁ : ∀ a ha, f a + f (g a ha) = 0)
  (h₂ : ∀ a ha, f a ≠ 0 → g a ha ≠ a)
  (h₃ : ∀ a ha, g a ha ∈ s)
  (h₄ : ∀ a ha, g (g a ha) (h₃ a ha) = a),
  s.prod f = 0 := ...some omitted proof
```

Remark: The detailed proof in Lean uses strong induction on the sum and is omitted. One can see that the 4 assumptions stated actually guarantee a partition of the sum into disjoint pairs where each pair sums to 0. Hence the original sum is 0, similar to the argument as in Method 2 before.

Hence, applying this lemma to our case, where g is taken to be the involution map define before, yields 4 goals, which are exactly those 4 assumptions. Note from above, the notion `a ha` means $a \in \mathbb{N} \times \mathbb{N}$ while `ha` is the assumption that $a \in X_{n+2} \times X_{n+1}$. We can introduce `a ha` simply by `intros jk hjk`, so these will be omitted in the 4 goals stating below,

1 : `F2 g v jk +F2 g v (invo jk) = 0`

2 : `F2 g v jk ≠ 0 → invo jk ≠ jk`

3 : `invo jk ∈ finset.product (range (n' + 3)) (range (n' + 2))`

4 : `invo (invo jk) = jk`

Remark: In goal 3, `finset.product (range (n' + 3)) (range (n' + 2))` means exactly $X_{n'+3} \times X_{n'+2}$.

For goal 1, it is exactly applying `theorem F2_degenerate` after considering the two possible cases of $jk = (j, k)$, that is $j \leq k$ or $\neg j \leq k$ using the definition of *invo*. Lemma 4.3.3 is used.

For goal 2, one introduces the assumption that `F2 g v jk ≠ 0` and `invo jk = jk` and tries to derive a contradiction. This is easily done by considering the two cases in the definition of *invo*. Lemma 4.3.1 and 4.3.2 are used.

For goal 3, one has to use the definition of *invo* and argues by cases. One important notion in Lean that is used is that

```
theorem mem_range : m ∈ range n ↔ m < n := mem_range
```

where `range n := fin(n)`.

For goal 4, this is exactly Lemma 4.3.4!

Therefore, we proved all the 4 goals, and one can finish proving the final result on the sum. The statement in Lean with a general structure of the proof is presented as follow,

```
theorem double_sum_zero1 (n':ℕ)(G : Type*)[group G](g:fin (n'+3)→ G)(M :
   Type*) [add_comm_group M] [G_module G M](v:cochain (n'+1) G M):
(range (n'+3)).sum(λ i, (range (n'+2)).sum(λ j, (F2 g v (i,j)))) =0:=
```

```
begin
  rw <-sum_product,                                  --This is the conversion
  apply sum_involution (λ jk h, invo jk),            --Applying involution
  { intros jk hjk,                                   --Goal 1
    by_cases hin : j ≤ k,                             --Consider by cases
    { rw add_comm,
      convert F2_degenerate hin g v,                 --Use theorem F2_degenerate
      exact invo_def1 hin},
     --some omitted lines of proof
  },
  { intros jk hjk _ h,                               --Goal 2: introduce all
   assumptions
    dsimp at h,
    by_cases hin : jk.1 ≤ jk.2,                       --Consider cases
    { apply invo_ineq1 hin,                          --Apply Lemma 4.3
      rwa h},
     --some omitted lines of proof
  },
  { intros jk hjk,                                   --Goal 3
    --some omitted lines of proof
    cases mem_product.1 hjk with hj hk,              --Consider cases
    --some omitted lines of proof
  }
  { intros jk hjk,                                   --Goal 4
    dsimp,
    exact invo_invo jk,  }                           --Exactly Lemma 4.3.4
end
```

Remark: When there is more than 1 goal, and knowing that each of the proof would not be short, using {} to group the lines for proving each goal together, as shown in the above example, helps to structure the proof much better.

4.4 Definition of d^n

For the definition of d^n, we can define the F_{first} in Lean similar to the way of defining $F_{n,j}$, but much simpler.

```
def F_first{n:ℕ} {G : Type*}[group G](g:fin (n+1)→ G):fin n→ G
:= λ k,   g  ⟨k.val+1, add_lt_add_right k.2 1⟩
```

This leads us to the definition of d^n in Lean, as follow,

```
def d.to_fun {n:ℕ}{G : Type*} [group G] {M : Type*} [add_comm_group M]
   [G_module G M]
(φ: cochain n G M): (cochain (n+1) G M):= λ(gi: fin (n+1) → G),
gi ⟨0, (by simp)⟩ · φ (λ i, gi ⟨i.val + 1, add_lt_add_right i.2 1⟩)
+(range (n+1)).sum(λ j,(-1:ℤ  )^(j+1)· φ (F j gi))
```

Remark 1: Almost every single term in this definition has a similar form that we have thoroughly explained before. This definition is long but should not be difficult to understand itself. Note that we did not use F_{first} actually in the definition, though the part corresponding to it has exactly the same expression as that of F_{first}. This is chosen because we can then avoid unfolding F_{first} for every time we use this definition.

Remark 2: You may notice that the name is `d.to_fun`. This is because finally we are going

to give a structural definition of d^n as a group homomorphism. You have seen in Section 2 such that d^n is indeed a group homomorphism. The reason for this choice of structural definition is explained before in Section 3 under Bundled vs. Un-bundled.

Remark 3: One should be quite familier with how to decide what inputs should we give for a definition. For `d.to_fun`, it is simply one input `(φ: cochain n G M)`.

Before proceeding to prove that $d^{n+1} \circ d^n = 0$ using this definition, one thinks about checking whether this definition indeed provides exactly what we want. Because it is a rather long definition, we do not want to waste time working on a wrong definition for hours solving an extremely complex but wrong equation. An idea is to check it with an simple example. We can choose 1-cocycle to test this definition, as follow,

```
example (G : Type*) [group G] (M : Type*) [add_comm_group M] [G_module G M]
(φ : cochain 1 G M) (hφ : d.to_fun φ = (λ i, 0)) (g h : G) : φ (λ _, g *
   h) = φ (λ _, g) + g • φ (λ _, h) :=
begin
  unfold d.to_fun at hφ,                          --Use the definition before
  let glist : fin 2 → G := λ i, if i.val = 0 then g else if i.val = 1 then
   h else 1,                                        --Introduce a 2-cochain
  have h2 : (λ (gi : fin (1 + 1) → G),
       gi ⟨0, _⟩ • φ (λ (i : fin 1), gi ⟨i.val + 1, _⟩) +
         finset.sum (finset.range (1 + 1)) (λ (j : ℕ), (-1: ℤ) ^ (j + 1) •
   φ (F j gi))) glist = 0,
    rw hφ,
                                                  --Rephrase the assumption
  --some omitted lines of proof about rewriting h2,
  convert h2,                                     --Use h2 to prove the goal
  {                                               --3 goals to proof
    ext,
    cases x with x hx,                            --argue by cases
    cases (nat.sub_eq_zero_of_le hx),
    refl,
  },
    --some omitted lines of proof similar to the one above
end
```

In the statement, it states that with the hypothesis that $d^\circ\phi = 0$ where 0 is the 0 map that sends everything to 0, then ϕ must satisfy $\phi(gh) = \phi(g) + g \cdot \phi(h)$. As shown in the structure of the proof, one uses the definition we gave and indeed arrives at the desired result, hence showing that our definition is most likely to be true.

Remark: Note that above we have a long rewriting of `hφ`. It is sometimes necessary to just write out the expression yourself, and give it a proof even though it might be a trivial one-line proof. Lean has the limitation that sometimes it cannot identify certain expressions and does not know how to and where to apply some tactics when it is not very explicit, and it will return an error.

4.5 $d^{n+1} \circ d^n = 0$

Well, to begin with, we always give the statement in Lean first. This is not difficult at all, once we know what the input does `d.to_fun` take and what type of output it generates. We have the following,

```
theorem d_square_zero{n:ℕ}{G : Type*} [group G] {M : Type*} [add_comm_group
   M] [G_module G M]
(φ: cochain n G M):d.to_fun (d.to_fun φ )=λ(gi: fin (n+2) → G), 0:= sorry
```

Note the RHS simply represents the 0 function of $G^{n+2} \to M$.

However, one may expect a very long proof by inspecting the length of definition and the complexity of the proof presented in Section 2. Indeed, just to have a taste, after expanding the definition of `d.to_dun`, and some simplifying, we obtain the goal as,

```
⊢ gi ⟨0, _⟩ ·
       sum (range (n + 1)) (λ (x : ℕ), (-1) ^ (x + 1) · φ (F x (λ (i :
   fin (n + 1)), gi ⟨1 + i.val, _⟩))) +
      (sum (range (n + 2))
           (λ (x : ℕ),
              (-1) ^ (x + 1) ·
                (F x gi ⟨0, _⟩ · φ (λ (i : fin n), F x gi ⟨i.val + 1, _⟩)) +
                   sum (range (n + 1)) (λ (j : ℕ), (-1) ^ (j + 1) · φ (F j
   (F x gi))))) +
        (gi ⟨0, _⟩ * gi ⟨1, _⟩) · φ (λ (i : fin n), gi ⟨i.val + 2, _⟩)) =
    0
```

We discuss everything that we need to complete this proof first. You may wish to refer back to the proof of Theorem 2.8 to see where does each of this seemingly trivial statement is used.

4.5.1 Facts about Double Sums

After we expand through the definition of `d.to_fun`, we are definitely going to get a term that is a double sum. This is exactly the reason of all the work we have done so far, namely, the important the result as stated in `theorem double_sum_zero1`. Unfortunately, we are not going to get the exact same expression as in the theorem after we expand. There is going to be some rearranging for us to find a match. This rearranging also appears in the proof of Theorem 2.8 but they are so trivial that we do not need to explicitly point out. However, due to ***'Non-Triviality'*** of Lean, we have to proof any statements we want to use for rearranging. Luckily, there are many that are already proved in Lean.

Firstly, we need the additive distributive law of a sum.

$$\sum_{i=0}^{n-1} (f(i) + g(i)) = \sum_{i=0}^{n-1} f(i) + \sum_{i=0}^{n-1} g(i)$$

This is already proved in Lean, stated as

```
lemma sum_mul_distrib : s.sum (λx, f x + g x) = s.sum f + s.sum g := sorry
```

Next, we need to be able to take out the first term of the sum, that is, when $i = 0$, and then changes the variable of the sum accordingly. Mathematically, it states,

$$\sum_{i=0}^{n-1} f(i) = f(0) + \sum_{i=0}^{n-2} f(i+1)$$

In Lean, it is also formulated and proved as follow,

```
lemma sum_range_succ' {β : Type v}[add_comm_monoid β] (f : ℕ → β) :
  ∀ n : ℕ, (range (nat.succ n)).sum f = (range n).sum (f ∘ nat.succ) + f
   0:= sorry
```

Similarly, one will think of removing the last term.

$$\sum_{i=0}^{n-1} f(i) = f(n) + \sum_{i=0}^{n-2} f(i)$$

and in Lean, the corresponding lemma is,

```
lemma sum_range_succ {β : Type v}[add_comm_monoid β](f : ℕ → β) (n : ℕ) :
  (range (nat.succ n)).sum f = f n * (range n).sum f := sorry
```

Remark 1: `nat.succ(n)` is defined as the next natural number after n, and is proved that `nat.succ(n)=n+1`. This is how natural numbers are define in Lean. With an initial number 0 and iteratively uses this `nat.succ(n)` function. Any natural number n in Lean hence can be consider to be either 0 or the result of `nat.succ(n')=n` for another natural number n'. This also forms the set up for induction.

Remark 2 **Important**: Notice there is an occurrence of $n-2$ in the mathematical term! The theorem in Lean is formulated to avoid the possible appearance of any $n-1$. However, if we stick to $n \in \mathbb{N}$ as our normal set up, we will face problem as one will see when we apply this to n for the LHS, we will have a set `fin (n-1)` on the RHS as the range of the sum. This will be problematic for $n = 0$. This is exactly why, as you will see in the final proof, that we separate the case of n to either 0 or `nat.succ(n')=n`, which indeed gives the appearance of $n' + 1 = n$ as you see before in earlier sections.

The next statement will be used in the case $n = 0$, as there will every explicit calculations there.

$$\sum_{i=0}^{0} f(i) = 0$$

In Lean,

```
lemma sum_range_zero (f : ℕ → β) :
 (range 0).sum f = 0 := sorry
```

The last lemma we need concerns the scalar multiplication relating a sum. One can directly see the use of this in the proof of Theorem 2.8 as we move -1 in and out of the sum.

$$\sum_{i=0}^{n-1} \lambda \cdot f(i) = \lambda \cdot \sum_{i=0}^{n-1} f(i)$$

```
def finset.sum_smul2 {α : Type*} {R : Type*} [semiring R] {M : Type*}
   [add_comm_monoid M]
  [semimodule R M] (s : finset α) (r : R) (f : α → M) :
    finset.sum s (λ (x : α), (r · (f x))) = r · (finset.sum s f) := sorry
```

Remark **Important**: Notice carefully the type of `(r : R)` where `[semiring R]`. Due to

'Type-Specificity' of Lean, this theorem about scalar multiplication only applied to any λ from a semi-ring. In particular, we can applied this to -1 but not $g \in G$! This is very important, as shortly in the proof we will see that we need an analogue statement about scalar multiplication as above but for $\lambda = g \in G$ where G is just a group. Hence, we need to produce a new lemma for this. See the subsection on Facts about G-module.

We will not be discussing the proofs of these properties as they are already done in Lean and are not part of this project. Nevertheless, we need to formally state them as they constitute important steps in the final proof we are going to construct in Lean.

4.5.2 Facts about $F_{n,j}$

As one may notice, apart from the double sum, there is still some forms of `F x g` appearing in the goal shown earlier. We would require certain properties of $F_{n,j}$ for us to cancel some terms out, just like the proof of Theorem 2.8.

As discussed in the section before, we may be interested in taking out the first term, giving some expressions evaluated at value 0. Thus, we can categorize our results to the 0 case of the general case, giving us the following 4 claims.

Let $F_{n,j}(g_0, g_1, \cdots, g_{n-1})_i$ be the $(i+1)$-th entry of the result for $i \in \mathbb{N}$ and $i \leq n-1$.

Lemma 4.4. *For $n > 1$,*

$$F_{n,0}(g_0, g_1, \cdots, g_{n-1})_0 = g_0 g_1$$

Proof. For $n > 1$, $n - 1 > 0$ so g_1 exists. Then the result follows from the definition when $j = 0$ and $i = 0 = j$. □

In Lean, it is stated as follow,

```
theorem F_0_0(n':ℕ){G : Type*}[group G](g:fin (n'+1+1+1)→ G):F 0 g ⟨0,
   nat.succ_pos (n'+1)⟩ = g⟨ 0, nat.succ_pos (n'+1+1) ⟩ *g⟨
   1,nat.lt_of_sub_eq_succ rfl⟩ :=
begin
refl,
end
```

Remark: Within the definition, we also have to provide proofs that an element is in the set `fin (n'+1+1+1)`. This expression of $+1+1+1$ is used instead of just $+3$ is to favour some proofs that uses the function `nat.succ`.

Similarly, we fixed $j = 0$, and varies $i > 0$ to get,

Lemma 4.5. *For $n > 1$,$i \in \mathbb{N}$*

$$F_{n,0}(g_0, g_1, \cdots, g_{n-1})_{i+1} = g_{i+2}$$

Proof. For $n > 1$, $n - 1 > 0$ so g_1 exists. Then the result follows from the definition when $j = 0$ and $i > j$. There is indeed an $+1$ position shift. □

The corresponding lemma in Lean is,

```
theorem F_0_j{n':ℕ}{j:fin (n'+1)}{G : Type*}[group G](g:fin (n'+1+1+1)→
   G):F 0 g⟨ j.val+1,nat.lt_succ_iff.mpr j.is_lt
⟩ =g⟨ j.val+2, add_lt_add_right j.is_lt 2⟩:=
begin
refl,
end
```

Now, fix $i = 0$, and varies $j > 0$. We have,

Lemma 4.6. *For $n > 1$, $j \in \mathbb{N}$,*

$$F_{n,j+1}(g_0, g_1, \cdots, g_{n-1})_0 = g_0$$

Proof. The result follows from the definition when $j > 0$ and $i = 0 < j$. □

In Lean, it is

```
theorem F_x_0(n' x:ℕ)(h:x>0){G : Type*}[group G](g:fin (n'+1+1+1)→ G):F x g
   ⟨0, nat.succ_pos (n'+1)⟩ = g⟨ 0, nat.succ_pos (n'+1+1) ⟩:=
begin
unfold F,
rw if_pos h,
end
```

Finally, without fixing anything, we have,

Lemma 4.7. *For $n > 1$, $j \in \mathbb{N}$, $i \in \mathbb{N}$,*

$$F_{n,j}(g_1, g_2, \cdots, g_n)_i = F_{n,j+1}(g_0, g_1, \cdots, g_{n-1})_{i+1}$$

Proof.

$$LHS = F_{n,j}(g_1, g_2, \cdots, g_n)_i = \begin{cases} g_{i+1} & i < j \\ g_{j+1}g_{j+2} & i = j \\ g_{i+2} & i > j \end{cases}$$

$$RHS = F_{n,j+1}(g_1, g_2, \cdots, g_n)_{i+1} = \begin{cases} g_{i+1} & i+1 < j+1 \implies i < j \\ g_{j+1}g_{j+2} & i+1 = j+1 \implies i = j = LHS \\ g_{i+2} & i+1 > j+1 \implies i > j \end{cases}$$

□

In Lean, it is

```
theorem F_x_j(n' x:ℕ){G : Type*}[group G](g:fin (n'+1+1+1)→ G):F x (λ
   (j:fin (n'+1+1)), g ⟨j.val+1, nat.lt_succ_iff.mpr j.is_lt,⟩ )
= λ (j:fin (n'+1)), F (x+1) g⟨j.1+1, nat.lt_succ_iff.mpr j.is_lt,⟩:=
```

```
begin
unfold F,
funext,
split_ifs,                                  --Use def of F to give 9 cases
{refl},
{exfalso,linarith},                         --In most cases contradiction
{exfalso,linarith},
{exfalso,linarith},
{refl},
{have h5: k.val+1=x+1,
rw h_1,
contradiction,
},
{exfalso,linarith},
{have h5: k.val=x,
exact nat.succ_inj h_3,
contradiction},
{refl},
end
```

Apart from the last one, the rest are really trivial lemmas. However, we need to state them correctly, and explicitly as the same as that will appear in the final proof. Otherwise, Lean will fail to identify the instance to apply a specific lemma, especially when there is a long expression.

4.5.3 Facts about G-Module

As discussed before, we will need an analogue of linearity of scalar multiplication for a sum for some $\lambda = g \in G$. This is achievable because we have our function f in the summand mapping to M, which is a G-module, and on which we have linearity of scalar multiplication with regards to the G-module.

Here is the analogue statement,

Lemma 4.8. *Given a group G, a G-module M, and a map $f : \mathbb{N} \to M$, we have for $n \in \mathbb{N}$, $g \in G$*

$$\sum_{i=0}^{n-1} g \cdot f(i) = g \cdot \sum_{i=0}^{n-1} f(i)$$

Proof. Induction on n. Base case is trivial as both side coincides. For the inductive hypothesis, we will use the linearity of scalar multiplication of G in M which states that $\forall g \in G$, $\forall m, n \in M$,

$$g \cdot (m + n) = g \cdot m + g \cdot n$$

Together with `lemma sym_range_succ` which allows us to take out the last term of the sum. That is,

$$\sum_{i=0}^{k-1+1} g \cdot f(i) = \sum_{i=0}^{k-1} g \cdot f(i) + g \cdot f(k) = g \cdot \sum_{i=0}^{k-1} f(i) + g \cdot f(k) = g \cdot \sum_{i=0}^{k-1+1} f(i)$$

□

Correspondingly, in Lean we have

```
lemma G_module.G_sum_smul {G : Type*} [group G] {M : Type*} [add_comm_group
   M]
[G_module G M] (n:ℕ )(g : G)(f: ℕ → M):finset.sum  (finset.range (n+1))(λ
   (x : ℕ ), g · f x) = g · finset.sum (finset.range(n+1)) f:=
begin
induction n with d hd,                          --Induction
norm_num,
rw finset.sum_range_succ,                       --First = in math proof
rw hd,                                          --Second =
rw <-G_module.linear,                           --Third =
rw <-finset.sum_range_succ _ (d+1),             --Third =
end
```

Here is a question, what if we have simultaneously $-1 \in \mathbb{Z}$ and some $g \in G$ present? Can we use either of the two lemmas stated about scalar multiplication be used? True, one may say that we can take out the left most one first by the appropriate lemma. However, the problem goes with whether Lean is able to successfully identify the left most one and to move out correctly. Even Lean can, there will be problem in the following situation:

- we want to move g out but having for some $k \in \mathbb{N}$

$$\sum_{i=0}^{n-1} (-1)^k \cdot g \cdot f(i)$$

The left-most approach would fail to achieve our goal. Indeed, the two lemmas are not enough, due to again ***'Type-Specificity'***. We need, in addition, the following lemma to exchange the places of -1 and g.

Lemma 4.9. *Given a group G, a G-module M, and a map $f : \mathbb{N} \to M$, we have for $n \in \mathbb{N}$, $g \in G$, $m \in M$,*

$$(-1)^n \cdot g \cdot m = g \cdot (-1)^n \cdot m$$

Proof. By induction on n. Base case trivial as both sides equal to 0. For inductive step, we have

$$(-1)^{k+1} \cdot g \cdot m = (-1)(-1)^k \cdot g \cdot m = (-1)g \cdot (-1)^k \cdot m = g \cdot (-1)^{k+1} \cdot m$$

□

In Lean, it has the following form,

```
lemma G_module.neg_one_pow_mul_comm {G : Type*} [group G] {M : Type*}
   [add_comm_group M]
[G_module G M] (n:ℕ  )(g:G)(m:M): (-1:ℤ )^n · g · m = g · (-1:ℤ)^n · m:=
begin
induction n with d hd,
norm_num,
rw nat.succ_eq_add_one,
```

```
rw pow_add,
norm_num,
exact hd,
end
```

4.5.4 Proof of the Final Result

We are finally ready to prove our final result. As mentioned before, we will separate to two cases $n = 0$ and nat.succ(n′)=n. For the first case $n = 0$, one can explicitly write up all the expression in the definition of d.to_fun and using suitable arithmetic tools already proved in Lean to solve the goal. We will not venture too much into that case.

For the second case, we will make use of all the theorems and lemmas proved before. The proof below gives the general structure of the proof.

```
theorem d_square_zero{n:ℕ}{G : Type*} [group G] {M : Type*} [add_comm_group
   M] [G_module G M]
(φ: cochain n G M):d.to_fun (d.to_fun φ )=λ(gi: fin (n+2) → G), 0:=
begin
  unfold d.to_fun,                                  -- Apply definition
  funext,
  dsimp,
  norm_num,
  cases n with n',                                   -- Divides to two cases
  { norm_num,                                       -- Case n=0
    rw sum_range_succ,                              -- Use facts on sum
    rw sum_range_succ,
    rw sum_range_zero,
--some omitted lines of proof for case n=0
},                                                  --  Case n=n'+1,
rw ←double_sum_zero1 n' G gi _ φ,                    --  Use result on double sum
unfold F2,                                          --  Apply def of F2
dsimp,
simp only [smul_add],
rw sum_add_eq_add_sum,                              -- This is sum_distributive
rw sum_range_succ' _ (n'+2),                        -- Facts on sum
rw F_0_0,                                           -- Lemma \ref{F00}
simp only [F_0_j],                                  -- Lemma \ref{F0j}
rw [zero_add, pow_one, neg_one_smul],
--some omitted lines of proof --arithmetic rewrites
simp only [F_x_0 _ _ (nat.succ_pos _)],             -- Lemma \ref{Fx0}
simp only [(F_x_j n' _ gi).symm],                    -- Lemma \ref{Fxj}
rw sub_eq_add^neg,
convert add_comm _ _,
{
  simp only [(pow_add' _ _ _).symm, nat.succ_eq_add_one],
  simp only [(G_module.G_sum_smul _ _ _).symm],     --Lemma \ref{Glinear}
  norm_num,
  simp only [G_module.neg_one_pow_mul_comm],        --Lemma \ref{Glinear2}
},
{
  simp only [(finset.sum_smul2 _ _ _).symm], }      --Facts about sum
--some omitted lines of proof --arithmetic rewrites
end
```

We have successfully proved one of the most important results for defining group cohomology

in Lean. Though this proof is not difficult mathematically speaking, but formalising in Lean requires a clear mind when dealing with an extremely long string of sums, trying to figure out the exact theorems to use, and filling every logical gap in the entire process. It is a significant work achieved in Lean.

At this point, the next goal will be to define d^n as a group homomorphism, define kernel and image(range) for a group homomorphism and hence define n-cocycle, n-coboundary and n-cohomology accordingly. In order to progress, we need some basic notions about additive group homomorphism, additive subgroup, kernel and image. Unfortunately there is no well-structured documents in Lean for these concepts. Before this project, there is only certain un-bundled lemmas that could be used for identification, for example, `def is_subgroup` for checking whether a set is a subgroup. We have discussed the benefits of having bundled files on such concepts as compared un-bundled ones, as such we will first make some well formulated bundle for the topics interested.

4.6 d^n as a Group Homomorphism

4.6.1 Bundle: Additive Subgroup

In this bundle, we introduce basic definitions, and prove important theorems about an additive subgroup. As kernel and image are both subgroups, we construct some special subgroups for later definitions of these two in the bundle of additive group homomorphism.

First important notion is the definition of an additive subgroup, which is defined as a set and a proof that this set is indeed a subgroup for a given group G. It appears in Lean as a structure, as follow,

```
structure add_subgroup (G : Type*) [add_group G] :=
(carrier : set G)
(is_add_subgroup : is_add_subgroup carrier)
```

For kernel, we define a special subgroup of a given group G to be set of one element containing only $0 \in G$ as we are in the additive group setting. One can easily prove that this is a subgroup. It is called `bot` in Lean,

```
def bot {G : Type u} [add_group G] : add_subgroup G :=
{ carrier := {x| x=0},
is_add_subgroup := sorry  --proof omitted
```

For range, we similarly define a special subgroup of a given group G to be set of all elements of G. It is called `top` and proved to be a subgroup.

```
def top {G : Type u} [add_group G] : add_subgroup G :=
{ carrier := {x | true},
  is_add_subgroup := sorry --proof omitted
```

The rest of the bundle consists of basic properties of a subgroup H of G, including $0 \in H$, $g \in H \iff -g \in H$, $g, h \in H \implies g + h \in H$ and the fact that H itself is an additive group. In addition, if G is an additive abelian group, then so is H. The statement and proofs can be found online and will not be discussed in detail.

One important thing to note is the type of H when it is treated as a group. Certainly it is not G, and not H either as H has type as additive subgroup. It is noted as H lifted up to G.

4.6.2 Bundle: Additive Group Homomorphism

This bundle imports the previously finished bundle `add_subgroup.basic`. Most of its content comes from bundled additive monoid homomorphism, but due to ***'Type-Specificity'*** , we have to create this new bundled additive group homomorphism based on previously defined additive subgroup.

The important results of this bundle are the definitions of image and co-images of a given set (group) with respect to a group homomorphism. These are formalized as `map` and `comap` in Lean. Depending on these two definitions, we can easily define kernel and range.

```
def ker (f : add_group_hom G H) :add_subgroup G:= add_group_hom.comap f ⊥

def range (f : add_group_hom G H) :add_subgroup H:= add_group_hom.map f top
```

The remaining section concerns with quotients which is just re-formalisation of scattered theorems about quotients in math library to make it a nice bundled file.

4.6.3 d^n as a Group Homomorphism

With bundled additive group homomorphism, we can proceed to define d^n as a group homomorphism. It is a structure where we have to provide a function f, a proof that $f(0) = 0$, and another proof that $f(a + b) = f(a) + f(b)$ for all a, b in the domain.

It is defined as follow with some new lemmas explained in the comments.

```
def d : add_group_hom (cochain n G M) (cochain (n + 1) G M) :=
{ to_fun := d.to_fun,                          --function is as before
  map_zero' := begin
    unfold d.to_fun,
    funext,
    have h1: (0:cochain n G M)=0, refl,
    have h2: (0:cochain (n+1) G M)=0, refl,
    simp only [zero_cochain n G M h1],
    --zero_cochain says 0 ∈ cochain maps everything to 0
    rw zero_cochain (n+1) G M h2,
    norm_num,
    rw G_module.zero,                    --says g· (0:M) = 0
  end,

  map_add' := begin
    unfold d.to_fun,
    intros,
    funext,
    simp only [sum_cochain, G_module.linear,
    G_module.neg_one_linear_applied, sum_add_distrib],
    --This is just applying linearity of cochain and G-module, and that
   (-1:ℤ) is also linear in scalar multiplication of G-module.
    rw [add_left_comm, <-add_assoc, add_left_comm, <-add_assoc,add_assoc],
   --arithmetic rearrangeing of a equation.
  end }
```

4.7 Definition of Cocycle

With kernel define, this is direct, as follow,

```
def cocycle (n:ℕ )(G : Type*) [group G] (M : Type*) [add_comm_group M]
   [G_module G M]
(N : Type*) [add_comm_group N] [G_module G N]:= (add_group_hom.ker (d (n+1)
   G M))
```

4.8 Definition of Coboundary

With range define, this is direct, as follow,

```
def coboundary (n:ℕ )(G : Type*) [group G] (M : Type*) [add_comm_group M]
   [G_module G M]
(N : Type*) [add_comm_group N] [G_module G N]:= add_group_hom.range (d n G
   M)
```

4.9 Definition of Cohomology

Due to ***'Type-Specificity'*** , we note that a normal quotient is defined as a group over another group. However, it our definition of n-cohomology, it is actually a subgroup(kernel) over another subgroup(range), with the additional property that this range is a subgroup of the kernel. There is a type difference. Instead of using the definition of quotient already in Lean, we need to define a notion for this kind of 'quotient', which leads us to this new bundle of additive sub-quotient.

4.9.1 Bundle: Additive Sub-quotient

The definition of the newly introduced concept `add_subquotient` is,

```
structure add_subquotient (A : Type*) [add_comm_group A] :=
(bottom : add_subgroup A)
(top : add_subgroup A)
(incl : bottom ≤ top)
```

One immediately proves that the `top`, `bottom` of an `add_subquotient` and moreover the `add_subquotient` itself are abelien groups. This is useful as we define cohomology using `add_subquotient`, we immediate gets that $H^n(G, M)$ forms an abelian group.

In addition to the above definition, one also proves in this file an important definition for the section on induced map on cohomology. That is, given abelian groups A, B, `add_subquotient` of A, B as Q and R, and a homomorphism $f : A \to B$, with the properties that f maps cocycles to cocycles and coboundaries to coboundaries. Then there is an induced homomorphism from Q to R.

4.9.2 Definition of Cohomology

With `add_subquotient`, we can simply give the definition of n-cohomology in Lean as

```
def cohomology (n:ℕ )(G : Type*) [group G] (M : Type*) [add_comm_group M]
   [G_module G M]: add_subquotient (cochain (n + 1) G M) :=
{bottom := coboundary n G M,
top := cocycle n G M,
incl := cob_sub_of_coc n G M}

where the inclusion can be proved very simply by playing around definitions
   and applying \lstinline{theorem d_square_zero1} that we have already
   done in Lean.
```

4.10 Partially Finished: Long Exact Sequence

A definition and some fundamental facts of G-module homomorphism will form the basic as we moves towards the long exact sequence of group cohomology, which is the next concept we want to formalize in Lean. The plan can be segmented into the following parts:

- Step 1: For a given G-module homomorphism `f: M\to N`, define a cochain map from `cochain n G M` to `cochain n G N`.
- Step 2: Prove this cochain map commutes with the differential `d.to_fun` as defined in Lean.
- Step 3: Define the induced homomorphism on cohomology, from `cohomology n G M` to `cohomology n G N`.
- Step 4: Define exactness in Lean.
- Step 5: Prove that given an short exact sequence of G-modules M, N, L and G-module homomorphism $f : M \to N$, $g : N \to L$, we have the an short exact sequence of on cohomology with the induced homomorphism defined in Step 3.
- Step 6: Define the connecting homomorphism from $H^n(G, L)$ to $H^{n+1}(G, M)$.
- Step 7: Prove that the long sequence induced on cohomology is indeed exact.

However, as many dependencies essential for proving this theorem, such as the zig-zag lemma in algebraic topology, is still not in place in Lean yet, we will not be able to finish all the steps. This project finish with the completion of step 3 above.

4.10.1 Bundle: G-Module and G-Module Homomorphism

The bundle `G_module.basic` contains basic definition and results of the scalar multiplication for a G-module M. The definition is given as follow. Some properties include those we have already discussed in Section 4.5.3.

```
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)
```

The bundle `G_module.hom` contains definition of a G-module homomorphism, that is, a homomorphism that is G-linear, and its properties. The definition is constructed as follow, and denoted by `M →[G] N`.

```
structure G_module_hom :=
(f : M →+ N)
(smul : ∀ g : G, ∀ m : M, f (g · m) = g · (f m))
```

4.10.2 Step 1: Induced Cochain Map

Given a G-module homomorphism f, we have an induced cochain map, just as discussed in Section 2.

```
def cochain.map {n : ℕ} (f : M →[G] N) : cochain n G M → cochain n G N :=
λ b c, f (b c)
```

It is indeed a homomorphism with simple checks in Lean,

```
def cochain.hom (f : M →[G] N) : cochain n G M →+ cochain n G N :=
{ to_fun := cochain.map f,
  map_zero' := begin unfold cochain.map, ext, apply
   add_monoid_hom.map_zero, end,
  map_add' := begin intros, unfold cochain.map, funext, apply
   add_monoid_hom.map_add,end }
```

4.10.3 Step 2: A Commutative Diagram

The important result is,

```
theorem d_map (n : ℕ) (f : M →[G] N) (c : cochain n G M) :
cochain.map f (d.to_fun c) = d.to_fun (cochain.map f c) :=
begin
  ext gs,
  unfold d.to_fun,
  unfold cochain.map,
  rw f.map_add,        --says f(a+b)=f(a)+f(b)
  rw f.map_smul,       -- says f(gm)=gf(m)
  rw f.map_sum,        -- says linearity of a sum
  congr',
  ext x,
  show (f.f) _ = _,
  exact add_monoid_hom.map_gsmul _ _ _,
end
```

It is not a mathematically difficult proof. It is achieved by rearranging and simply applying linearity everywhere. Of course, these facts about linearity have to be proved first, but most of which are already equipped in Lean.

4.10.4 Step 3: Induced Map on Cohomology

This is where we use the important definition of the induced homomorphism in the bundled file `add_subquotient.basic`, we have the induced map on group cohomology as

```
def cohomology.map (f : M →[G] N) : cohomology n G M →+ cohomology n G N:=
add_subquotient.to_add_monoid_hom (cocycle.map_incl f) (coboundary.map_incl
   f)
```

where the two inputs are the proofs that the induced cochain map sends cocycles of M to cocycles of N, and similarly for coboundaries. One can also find out explicit expressions for the induced homomorphism on these cocycles and coboundaries but it is unnecessary.

Notice that in any of the definition of these induced homomorphisms, the number n is never explicit specified. Actually, one can change the value of this n freely as long as it is consistent in the corresponding definition of cochain, cocycle, coboundary and cohomology. This is an essential fact as this actually shows that these induced maps can be created for any $n \in \mathbb{N}$, hence forming the required complexes and morphisms as defined in Section 2 which is important for the proof of the long exact sequence of group cohomology. We stop here for this project.

5 Conclusion and Further Researches

5.1 Main Results

In summary, we have successfully achieved in both providing the blueprint and formalizing the first general definition of group cohomology in Lean. Moreover,

- The error-free and sorry-free Lean repository on the definition of group cohomology verifies that our blueprint is correct.
- The new introduced definitions in Lean, such as n-cochain, differential d^n, n-cocycle, n-coboundary, and n-cohomology are checked by examples and theorems such that they are defined correctly.
- We have shown that Lean is capable of understanding group cohomology.
- We have restructured and bundled some concepts of additive groups in Lean which make future applications smoother.
- We have made some progress to formalizing the long exact sequence of group cohomology in Lean. Once the dependencies are in place in Lean, such as definition of exactness, the proof of the zig-zag lemma, and formalization of long exact sequence from short exact sequence that are essential to the proof of the theorem, one can continue from our progress to finish the induced long exact sequence in Lean based on the definitions given in this project.

5.2 Implication in Lean

As a result of group cohomology's usefulness, our main achievement of formalizing the first general definition of group cohomology in Lean has a significant implication to Lean. It sets up the basics in this area of study of groups and opens up countless possible directions of future projects in Lean. For example, it makes all the formalization of theorems about group cohomology possible. Actually more generally, it makes any theorems using group cohomology in its statement (with other notions in the statement already defined in Lean) possible. For example, one may think of formalizing Hilbert's Theorem 90 and Brauer groups in Lean. In addition, due to the high relevance of group cohomology in modern formulation of class field theory, our definition of group cohomology in Lean makes class field theory a targeted topic in advanced mathematics that to be formalized in Lean.

On the other hand, Lean was not created to just to verify proofs. In a holistic point of view, it aims to achieve automation of proof gradually, that is, providing a proof by Lean itself. One can already see that Lean at this point is capable of finding some simple proofs by itself through algorithms such as `library_search`, and `linarith`. We may foreseen in the future, as the library of Lean builds up, and as Lean 'learns' more and more concepts, that it is capable of performing machine learning and creating something new on its own. This project enriches the math library in Lean by adding an important concept and opening possibilities for many more important topics, and hence makes contribution towards this automation objective of Lean.

5.3 Possible Directions of Future Researches

As discussed before, there are abundant possible directions of researches made available by our achievements. Some include,

- Continue to work on theorems about group cohomology in Lean, starting with the induced long exact sequence.
- Define Brauer groups in Lean using the definition of group cohomology and hence proceed to formalizing class field theory in Lean.
- Cross comparison with non-general definitions of group cohomology. For example, one can proof our definition in Lean is the same as that of in GAP to prove our definition is computationally efficient.
- Check other definitions of group cohomology in Lean. For example, prove that the cohomology groups $H^n(GM)$ are derived functors, or choose another resolution to obtain a new formula for the differentials d^n and compare which is more convenient to work with.

6 References