Group Cohomology in the Lean Community Library

Amelia Livingston
King's College London, UK

Abstract

Group cohomology is a tool which has become indispensable in a wide range of modern mathematics, like algebraic geometry and algebraic number theory, as well as group theory itself. For example, it allows us to reformulate classical class field theory in cohomological terms; this formulation is essential to landmarks of modern number theory, like Wiles's proof of Fermat's Last Theorem. We explore the challenges of formalising group cohomology in the Lean theorem prover in a generality suitable for inclusion in the community library `mathlib`.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases formal math, Lean, mathlib, group cohomology, homological algebra

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.22

Supplementary Material *Software*: `https://github.com/101damnations/ITP2023SupplementaryMaterial`, archived at `swh:1:dir:d647ad13fc6cc27a6fe9f3b49691d0e1716a532c`

Funding This work was supported by the Engineering and Physical Sciences Research Council [EP/S021590/1]. The EPSRC Centre for Doctoral Training in Geometry and Number Theory (The London School of Geometry and Number Theory), University College London

Acknowledgements I am very grateful to Kevin Buzzard for his ongoing mathematical and Lean-related support and guidance. I am also indebted to Joël Riou for his explanation of the simplicial interpretation of group cohomology and his thorough reviewing of and advice regarding my work, and for his formalisation of some of the results I used. I also depended heavily on Scott Morrison's development of Lean's representation theory library and the category theory library more generally. Finally, thanks to anyone who answered my questions on the Xena Project Discord server and the Lean Zulip.

1 Introduction

1.1 Motivating group cohomology

There are many cohomology theories in mathematics. They associate simpler, "linear" invariants (vector spaces, or more generally modules, and linear maps between them) to more complicated objects, and analysing these invariants can answer questions about the complicated objects.

We want a cohomology theory for groups. They are ubiquitous in maths. Groups themselves often appear as invariants of more complex objects: we can study a topological space by studying its fundamental group, or field extensions by their Galois groups, or rings by their K-groups in algebraic K-theory. But they are still more complicated than "linear" invariants, and abstract group theory itself is not easy. The simpler invariants we obtain in group cohomology come from asking how a group acts on other objects, rather than analysing it internally.

This is the spirit of group cohomology – but there are multiple ways to actually define it. This is often the case in maths, and different definitions lend themselves to different exploits. There might be a particularly abstract formulation, expressing a concept as a special case of some more general category-theoretic notion. This perspective tends to give us access to powerful techniques for developing theory. But because the tools come from a more general setting, they are not specialised for the computation of examples, or for proving theorems that depend on the specificities of our situation. For these, we need a different, more down to earth approach, which will necessarily vary from setting to setting.

Because cohomology theories typically have a similar abstract foundation, there are many cohomological examples of this abstraction versus practicality dichotomy. For example, in algebraic topology, we find a variety of ways to compute singular cohomology, e.g. via simplicial, cellular or de Rham cohomology. More generally, we can calculate sheaf cohomology using Čech cohomology.

The situation is no different for group cohomology. Given a group G acting on an abelian group M, group cohomology is a family of groups $\mathrm{H}^n(G, M)$ for $n \in \mathbb{N}$. Mathematicians have analysed the groups of low degree, i.e. for $n \leq 2$, via explicit calculation since before "group cohomology" was a term [14, p. 10]. For example, H^1 appears in Hilbert's Theorem 90, originally proved by Kummer in 1855 [8, p. 213]; among many other applications, this result parametrises the solutions to certain Diophantine equations [7, p. 3]. Meanwhile, H^2 classifies group extensions, as explained by Baer and others in the early 20th century [14, p. 10]. Given an abelian group A and another group G, this means classifying the groups E having A as a normal subgroup and such that $E/A \cong G$, revealing the ways in which bigger groups can be built from smaller ones.

On the other hand, the abstract story, which was developed in the mid-20th century [14, p. 11–12], gave us more tools to analyse $\mathrm{H}^n(G, M)$ for arbitrary n, and explore relationships between these groups as n, G and M vary, via homological techniques. It also connects group cohomology to topological cohomology. Thus we wish to formalise both the abstract and concrete approaches, and prove their equivalence. This equivalence is the focus of the paper.

1.2 Lean and `mathlib`

Lean is an interactive theorem prover that uses dependent type theory; every "object" in Lean is a term of a unique type. A Lean file mainly consists of definitions, lemmas and theorems, which the user must prove with the assistance of tactics that provide some degree of automation. We will meet some tactics and structural features of Lean during the paper, but try to explain these with as little code as possible. When this is unreasonable, we provide links to Lean files, indicated with the symbol 🔗, which illustrate details in more depth. An archive containing saved versions of the non-permanent links in the paper can be found at 🔗.

Like maths, formalisation is a collaborative process. In order to make progress, we must make use of work that has been done before; some of this work is collected in a library called `mathlib` 🔗. The library comprises folders for each mathematical subfield, and each file consists of a collection of definitions and facts, which we call an API, relating to a particular mathematical concept. But there are typically many ways to formalise the same object, and it is not always obvious which ways are "right": which implementation can feasibly be used in the formalisation of further material. Lean cannot work this out for us; many factors influence the extensibility of Lean code, and typically we must make an educated guess at the correct formalisation and potentially refactor when a future user runs into difficulties. This makes the task of writing `mathlib`-suitable code significantly harder than code which just compiles, and the `mathlib` library has a rigorous community review process to try and avoid having to refactor new work down the line. There is no algorithm for making sure the growth of `mathlib` is sustainable; it seems to require human insight.

Often this is because the library needs to meet the needs of humans. On the one hand, it should be organised coherently, so the user has some idea of where to find what they need. On the other hand, APIs should be fleshed out enough that a user does not need to know the specific implementation the author chose – to an extent, there should be support for any other user imagining a different implementation. It seems to the author that these aims can conflict.

But not all challenges in `mathlib` design are due to the limitations of humans. One reason for this is the complexity of the algebraic hierarchy (a portion of which can be seen in [10, p. 4]). Everything in `mathlib` should be stated in the maximum possible generality within this hierarchy, so that it can be used in any setting in which it applies. In simple examples this is easy to ensure – it is not hard to check if a lemma about groups actually applies to monoids too.

In more complicated settings, however, the pursuit of generality is less straightforward. Firstly, it often means needing to create more new API than previously anticipated: it took a surprisingly long time for real manifolds to enter `mathlib`, as contributors needed to develop a wide variety of more general material, like the Bochner integral. But this is not the only difficulty. In principle, abstraction should simplify code – but of course we will need to apply the general material to more specific situations, and this is where complexity arises. For instance, in our simple example, a lemma about groups becomes a lemma about monoids which we are applying to a group, so we have to use something extra: the fact that a group is a monoid. Obviously, this is fine. But the many iterations of this principle in complicated settings has been a factor in most of the challenges in this project, as it can slow Lean down and make errors more difficult to troubleshoot.

The project has taken a long time to develop considering its mathematical simplicity. The author has now contributed two definitions of group cohomology to `mathlib`, as well as proof of their equivalence. This required 10 sizeable pull requests, and often the material had been rewritten to increasing degrees of abstraction. The paper will illustrate in detail the development of this code.

Section 2 explains the essentials of the maths we wish to formalise, and explores some fundamental design decisions and Lean principles. Section 3 describes our formal version of a key object called the standard resolution, and Section 4 discusses how we use this object to define group cohomology in Lean. In Section 5 we conclude and detail the future of the project: the author is currently using the definitions explained to create further API and prove various group cohomological results, although the code is not yet prepared enough for presentation.

2 Preliminaries

2.1 Mathematical background

▶ **Definition 1.** *Given an abelian category $\mathcal{C}$ (for example, the category of abelian groups, or the category of modules over a ring) a* ***cochain complex*** *X in $\mathcal{C}$ indexed by $\mathbb{N}$ is a sequence*

$$0 \to X_0 \xrightarrow{d_0} X_1 \xrightarrow{d_1} \dots \xrightarrow{d_{n-1}} X_n \xrightarrow{d_n} \dots$$

of objects $X_n \in \mathcal{C}$ and morphisms $d_n : X_n \to X_{n+1}$, $n \in \mathbb{N}$, satisfying $d_{n+1} \circ d_n = 0$ for all n.

We call the morphisms differentials. The condition $d_{n+1} \circ d_n = 0$ means the image of d_n is contained in the kernel of d_{n+1}, allowing us to define

▶ **Definition 2.** *The* ***nth cohomology*** *of X, $\mathrm{H}^n(X)$, is $\mathrm{Ker}(d_n)/\mathrm{Im}(d_{n-1})$.*

A **chain complex** is the same, but with the morphisms in the other direction: d_n is a morphism $X_{n+1} \to X_n$, and the analogous invariant is called **homology**, denoted $\mathrm{H}_n(X)$.

With this, we can explain the abstract and concrete perspectives on group cohomology, starting with the latter. The material can be found here [2, Chapter 4].

Given a group G, we call an additive commutative group M a G-module if it has a map $\cdot : G \times M \to M$ satisfying $gh \cdot m = g \cdot h \cdot m$ for $g, h \in G, m \in M$ and which also distributes over addition, i.e. $g \cdot (m + n) = g \cdot m + g \cdot n$ for $g \in G, m, n \in M$.

▶ **Definition 3.** *The **nth group cohomology** of G and M, denoted $\mathrm{H}^n(G, M)$, is the nth cohomology of the cochain complex*

$$0 \to M \to \mathrm{Fun}(G, M) \to \mathrm{Fun}(G^2, M) \to \mathrm{Fun}(G^3, M) \to \dots \tag{1}$$

where $\mathrm{Fun}(G^n, M)$ is the set of functions from G^n to M, and the differential d^n maps $f : G^n \to M$ to the function sending $(g_0, \dots, g_n)$ to

$$g_0 \cdot f(g_1, \dots, g_n) + \sum_{i=0}^{n-1} (-1)^{i+1} f(g_0, \dots, g_i g_{i+1}, \dots, g_n) + (-1)^{n+1} f(g_0, \dots, g_{n-1})$$

(and $d^0(m) = (g \mapsto g \cdot m - m)$).

The differential is slightly messy, but this is the formulation that in some sense "shows up in nature"; for example, it makes clearer the correspondence between $\mathrm{H}^2(G, M)$ and certain equivalence classes of group extensions. We call the $\mathrm{Fun}(G^n, M)$ **inhomogeneous cochains**.

The abstract perspective, meanwhile, is often introduced in a category of modules over a ring, rather than in the category of G-modules (where morphisms are group homomorphisms satisfying $f(g \cdot x) = g \cdot f(x)$ for all $g \in G$). Like `mathlib`, the average undergraduate student contains a far larger API for modules than for other concrete abelian categories, so when explaining group cohomology it is natural to exploit the equivalence between the category of G-modules and the category of modules over the following ring:

▶ **Definition 4.** *The **group ring** $\mathbb{Z}[G]$ is the free abelian group on G (functions $G \to \mathbb{Z}$ which are nonzero at finitely many elements of G), with multiplication induced by that of G.*

We denote its elements as sums $\sum n_i g_i$ for $n_i \in \mathbb{Z}, g_i \in G$. We can inject G into $\mathbb{Z}[G]$ by sending g to the function which is 1 at g and 0 everywhere else; we will often abuse notation and just denote this function by g.

Given this category equivalence G-Mod $\cong \mathbb{Z}[G]$-Mod, we can express group cohomology as objects called Ext groups. Ext is an example of a derived functor; these are abstract objects equipped with high-powered theory and are useful in many algebraic fields of maths. We summarise their definition; the material can be found in [13, Chapters 1, 2].

▶ **Definition 5.** *A **morphism of chain complexes** $f : X \to Y$ is a family of morphisms $f_n : X_n \to Y_n$ making the resulting diagram commute – that is, $d_n^Y \circ f_{n+1} = f_n \circ d_n^X$ for each n.*

A chain complex morphism induces maps on each homology group $\mathrm{H}(f_n) : \mathrm{H}_n(X) \to \mathrm{H}_n(Y)$; we call f a **quasi-isomorphism** if each $\mathrm{H}(f_n)$ is an isomorphism. Analogous definitions can be made for morphisms of cochain complexes and cohomology.

To analyse an object X in an appropriate category $\mathcal{C}$, we can sometimes associate to it a chain complex of simpler objects, called a "resolution" of X, and study that instead. More precisely:

▶ **Definition 6.** *A **projective resolution of** X is a chain complex P and a quasi-isomorphism of chain complexes $f : P \to X[0]$ such that each P_n is **projective** (a certain "nice" property).*

By $X[0]$ we mean the complex whose 0th object is X, with every other object 0. All the f_n for $n > 0$ are necessarily 0. The requirement that f is a quasi-isomorphism means $\mathrm{H}^n(P)$ must be trivial for all $n > 0$; we say P is **exact except at the right**.

When we analyse what certain "nice" functors F do to X, we can learn more by applying them to all of P, and then taking the (co)homology of the resulting complex, which in general will no longer be trivial. However, due to the conditions in the definition of a projective resolution, $\mathrm{H}^0(F(P))$ will always be isomorphic to $F(X)$, so we do not lose information. The $\mathrm{H}^n(F(P))$ are independent of the projective resolution chosen, and can be extended to functors; we call them the derived functors of F.

Given an object Y, one such F is $\mathrm{Hom}(-, Y)$. This functor is contravariant, meaning it flips the directions of maps: a map $\phi : X_1 \to X_2$ is sent to the map $\mathrm{Hom}(X_2, Y) \to \mathrm{Hom}(X_1, Y)$ given by precomposition with ϕ. Because of this contravariance, the functor sends a chain complex to a cochain complex, giving us

$$0 \to \mathrm{Hom}(P_0, Y) \xrightarrow{- \circ d_0} \mathrm{Hom}(P_1, Y) \xrightarrow{- \circ d_1} \mathrm{Hom}(P_2, Y) \to \dots$$

▶ **Definition 7.** *For P a projective resolution of X, the nth cohomology of the above complex is called $\mathrm{Ext}^n(X, Y)$. It is independent of the resolution chosen.*

Now, returning to group cohomology, we can appeal to the undergraduate's module API to observe that

$$\begin{aligned}\mathrm{Fun}(G^n, M) &\cong \mathrm{Hom}_{\mathbb{Z}}(\mathbb{Z}[G^n], M) \\ &\cong \mathrm{Hom}_{\mathbb{Z}[G]}(\mathbb{Z}[G], \mathrm{Hom}_{\mathbb{Z}}(\mathbb{Z}[G^n], M)) \cong \mathrm{Hom}_{\mathbb{Z}[G]}(\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n], M) \qquad (2)\end{aligned}$$

Here Hom_R denotes morphisms in the category of R-modules, and the $\mathbb{Z}[G]$-module structure on $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ is given by $x \cdot (y \otimes z) = xy \otimes z$.

This isomorphism suggests that our concrete group cohomology groups are actually Ext groups of some sort. Indeed, the modules $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ are not just projective, but free, a stronger property. A **free module** is a module with a basis; for example, a vector space is a module over a field, and since all vector spaces have a basis, every module over a field is free. Since the $\mathbb{Z}[G^n]$ are free $\mathbb{Z}$-modules, the $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ are free $\mathbb{Z}[G]$-modules, and hence projective. We will also apply an isomorphism $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n] \cong \mathbb{Z}[G^{n+1}]$; for our concrete group cohomology to agree with certain Ext groups, we then seek a projective resolution whose nth object is $\mathbb{Z}[G^{n+1}]$. Indeed:

▶ **Definition 8.** *The **standard resolution** is the chain complex*

$$\dots \xrightarrow{d_2} \mathbb{Z}[G^3] \xrightarrow{d_1} \mathbb{Z}[G^2] \xrightarrow{d_0} \mathbb{Z}[G] \to 0 \qquad (3)$$

with differential sending $g = (g_0, \dots, g_n) \in G^{n+1}$ to

$$\sum_{j=0}^{n+1} (-1)^j (g_0, \dots, g_{j-1}, g_{j+1}, \dots, g_n).$$

The cokernel of d_0, i.e. $\mathbb{Z}[G]/\mathrm{Im}(d_0)$, is $\mathbb{Z}$. Indeed, this is a projective resolution of $\mathbb{Z}$ considered as a trivial $\mathbb{Z}[G]$-module – that is, $g \cdot m = m$ for each $g \in \mathbb{Z}[G], m \in \mathbb{Z}$. We will use this resolution to show that the nth group cohomology of M is in fact $\mathrm{Ext}^n_{\mathbb{Z}[G]}(\mathbb{Z}, M)$, and thus connect the concrete and abstract interpretations of group cohomology.

2.2 Initial formalisation considerations

There are many respects in which our formalisation is not a direct translation of the maths just outlined. To start explaining the code, we must address the more fundamental of these, which appear at every point in the project.

2.2.1 Complexes in Lean

We first explain the `mathlib` definition of complexes, which is a little counterintuitive, and which illustrates one of the Lean community's adaptations to the quirks of dependent type theory.

We first note that complexes in Lean have always permitted a more general indexing type than $\mathbb{N}$, and although the issues we will detail can also arise for $\mathbb{N}$, the most natural examples are for $\mathbb{Z}$.

If we followed our nose, our definition of a cochain complex in Lean would involve a function `X:` $\mathbb{Z} \to \mathcal{C}$ and a function `d` sending $n \in \mathbb{Z}$ to a morphism `X(n)` $\to$ `X(n + 1)`. This is essentially how complexes were originally formalised, and is fine in set theory, but not when we translate to dependent type theory. For example, when $\mathcal{C}$ is the category of abelian groups, we might want to know whether a term `x :X(n)` is in the image of the differential. But the type of this differential, according to our setup, is `d:X(n - 1)` $\to$ `X(n - 1 + 1)`. Of course, $n - 1 + 1$ is equal to n, but the problem is that type theory has multiple notions of equality.

The simplest one is syntactic equality: two objects are syntactically equal if they are "the same characters in the same order". $A = A$ is a syntactic equality. Next we have definitional equality – when, after unfolding definitions, two objects reduce to being syntactically equal:

```
def X : ℕ := 5
def Y : ℕ := 5
example : X = Y := rfl
```

where `rfl` proves the statement by unfolding the definitions of X and Y and applying reflexivity of equality. We note that not all tactics unfold terms like `rfl`. Given an equality or an iff statement whose lefthand side is syntactically equal to something in the goal, the `rewrite` and `simp` tactics can replace that expression in the goal with the righthand side of the equality/iff statement. The "simplifier" tactic `simp` is an example of Lean's automation: it searches through all lemmas tagged `@[simp]`, looking for statements whose lefthand side is syntactically equal to something in the goal, and then rewrites those lemmas. If it were to unfold terms as well as search its library, it would perhaps be rendered uselessly slow.

Finally, we have propositional equality: when two objects can be proved to be equal. For example:

```
example (a : ℕ) : a + 0 = a := rfl -- succeeds
example (a : ℕ) : 0 + a = a := rfl -- fails
```

since $\mathbb{N}$ is an inductive type, and addition is defined by induction on the second variable, not the first; we can prove the second statement by inducting on a, at which point we can appeal to definitional equalities. Hence the first statement is a definitional equality, and the second only a propositional equality.

Thus our issue is that $n - 1 + 1$ is not definitionally equal to n; its proof is the lemma `int.sub_add_cancel`. And in dependent type theory, everything is a term of a unique type; $d(y)$ cannot have type `X(n - 1 + 1)` and type `X(n)` simultaneously. If $n - 1 + 1$ had been definitionally equal to n, Lean could unify `X(n - 1 + 1)` and `X(n)` by unfolding definitions, but since these expressions are only propositionally equal, the statement $d(y) = x$ will not typecheck.

We can, of course, compose with an isomorphism `X(n - 1 + 1)` $\cong$ `X(n)`; this is the approach taken in Domínguez and Rubio's formalisation of chain complexes in Coq [4, p. 6]. The UniMath library also uses the intuitive definition of chain complexes 🔗, since the univalence axiom means that such an isomorphism is equivalent to an equality. We note that in simple type theory, meanwhile, one must take a different approach: Isabelle/HOL defines exact sequences inductively as a certain kind of set of pairs of objects and functions 🔗.

However, carrying around these extra isomorphisms is unwieldy (in Lean, at least), and to deduce things about the resulting maps the user must prove heterogeneous equalities: equalities between terms of different types, denoted $==$.

Discussion here 🔗 concerns similar issues raised by commutative differential graded algebras. Over a commutative ring R, these are families of R-modules A_n indexed by $\mathbb{N}$ with, among other things, a family of R-bilinear "multiplication" maps $A_i \times A_j \to A_{i+j}$ for $i, j \in \mathbb{N}$ satisfying certain axioms. But the natural statement of associativity, for example, does not typecheck.

Ultimately, for complexes, the Lean community settled on a different implementation: define cochain complexes (and chain complexes) to have a differential between *every* pair of indices i, j, and require a proof that these are equal to 0 unless $i + 1 = j$, as well as a proof that any two differentials compose to give zero. This was first suggested by Johan Commelin here 🔗 on March 9th, 2021. Whilst this definition seems strange, it means we only have to identify non-definitionally equal types or check $i + 1 = j$ during proofs, and not when defining data. This is much easier to work with, as demonstrated here 🔗: we can use the `cases` tactic on hypotheses like $i + 1 = j$ and replace the lefthand side with the right in the types involved in the goal. The definition of complexes was refactored to use this approach in 2021, by Scott Morrison, in this pull request 🔗.

2.2.2 The right generality

Secondly, when we apply the `mathlib` tenet of maximum generality, we notice something: our exposition of group cohomology all works if we replace $\mathbb{Z}$ by an arbitrary commutative ring k, and ask that M is a $k[G]$-module, where $k[G]$ is defined analogously to $\mathbb{Z}[G]$; the rest of this paper will always use general k. We made a choice here – the alternative would be to continue developing the theory over $\mathbb{Z}$, and then just tensor with k when we want a more general result [12, Tag 0DVD]. Likewise, we could define additive commutative groups as the special case of k-modules when $k = \mathbb{Z}$. The "hierarchy of generality" seems to contain cycles. In this latter case it is easier to define additive commutative groups on their own, as the definition of module naturally extends the definition of additive commutative group. However, in our case the other approach is preferable, as it is simpler to replace $\mathbb{Z}$ with k than to have to involve tensor products.

2.2.3 Exploiting typeclass inference

Thus we are now concerned with the group k-algebra $k[G]$ and $k[G]$-modules; this raises further choices. In maths we denote any "scalar-like" action by $\cdot$. We can emulate this in Lean using typeclasses – a strategy also used in other theorem provers, like Agda, Coq and Isabelle [1, p. 1]. Maths is built from complex hierarchies of structures; typeclasses make it easier to formalise these hierarchies in an efficient and usable way. They enable us to reuse API for simple structures when reasoning about more complicated superstructures; for example, we can use the notation $+$ when dealing with any structure inheriting a `has_add` instance. Often, a structure can inherit an instance via multiple different paths, called diamond inheritance. This is fine when the inherited instances are definitionally equal. For instance, the semiring instance on a commutative ring coming from the inherited ring instance is definitionally equal to the one coming from the inherited commutative semiring instance, so it does not matter which path typeclass inference uses. But in practice, different inheritance paths will not always lead to definitionally equal instances, and this is problematic. Moreover, whilst concise notation is an advantage of typeclasses, it can make it harder to see when Lean's behaviour is not what we want. In practice we often want to consider multiple different $k[G]$-actions on the same object. Here 🔗 is an example of two different instances clashing; the action of $k[k^\times]$ on itself naturally extending the action of $k^\times$ on coefficients is not the same as the action of $k[k^\times]$ on itself by multiplication. But even two equal instances can conflict. Lean cannot unify instances that are only propositionally equal: the point of typeclass inference is to reduce the need to supply arguments explicitly, so we cannot provide a proof to the inference system of such an equality.

If we have a non-definitionally equal diamond and still wish to exploit typeclass inference, a possible solution is type aliases: a nickname for a type.

```
variables (R : Type*) [ring R]
def copy := R -- ‘copy’ is a nickname for ‘R’
```

If we declare an instance on the type alias, it will not pollute the underlying type. For example, we can try defining 0 in `copy R` to be the 1 of the underlying ring:

```
instance : has_zero (copy R) := ⟨(1 : R)⟩
example : (0 : copy R) = 1 := rfl -- succeeds
example : (0 : R) = 1 := rfl -- fails
```

Conversely, Lean will not apply instances on the underlying type to the type alias, unless asked to:

```
instance : ring (copy R) := infer_instance -- fails
instance : ring (copy R) := by unfold copy; apply_instance -- succeeds
```

We can also use type aliases to organise API. For any type X and any type k with a 0, `finsupp X k` is the type of finitely supported functions $X \to k$. When G is a monoid and k is a commutative ring, `finsupp G k` is the k-algebra $k[G]$, with multiplication induced by that of G – but instead of creating this instance, `mathlib` defines a type alias `monoid_algebra k G` for `finsupp G k`, on which the k-algebra instance is defined. Hence results and instances relying on multiplication in G can be organised into the `monoid_algebra k G` API.

Thus, if we have a $k[G]$-action on a type M which could create diamonds if we declared it as an instance, we can instead make a type alias for M, and limit the scope of the instance.

In `mathlib`'s category theory library, we do something essentially equivalent to using type aliases constantly, but working in Lean's category `Module (monoid_algebra k G)` (i.e. $k[G]$-Mod) for our purposes is still tricky. Outside this library, an R-module is 3 different variables: a term M of type `Type*`, an additive commutative group instance `[add_comm_group M]`, and an R-module instance `[module R M]`. But when we work category-theoretically, an R-module is one variable: a term M of type `Module R`, which is a structure with 3 fields:

```
structure Module :=
(carrier : Type v)
[is_add_comm_group : add_comm_group carrier]
[is_module : module R carrier]
```

meaning the R-module structure on a term `M : Module R` is built into its type and is unambiguous. This achieves the same thing as declaring an alias for M, and then only defining the specific R-module structure we want on the alias.

But for us, even this will not suffice. If we package a G-module M with a compatible k-module structure as an object in $k[G]$-Mod, we still want to be able to talk about the underlying k-module structure, and the natural k-module structure on terms of type `Module (monoid_algebra k G)` is not definitionally equal to the k-module structure we started with, as proved here 🔗.

Instead, we use a further alternative. We bundle our actions of k and G as **k-linear representations of G**: a k-module M equipped with a monoid homomorphism `M.`ρ of type $G \to \mathrm{End}_k(M)$, where $\mathrm{End}_k(M)$ is the ring of k-linear maps from M to itself. When developing representation theory, `mathlib` contributors were unsure whether to define representations this way, or as objects with a separate k-action and G-action, or as $k[G]$-modules, and Antoine Labelle and Eric Wieser vouch for the first definition here 🔗 on April 19th, 2022. It is similar to the definition chosen in Coq's Mathematical Components library 🔗. This way we only deal with one k-module instance, and the G-action on M is unambiguous. However, since the action has become an explicit homomorphism `M.`ρ we cannot use typeclass inference or the notation $\cdot$. This does not even increase the number of arguments we need to give Lean, though; functions that would otherwise require the arguments k, G, M now only require M, since `M.`ρ contains the information of k, G and M in its type. We call the category `Rep k G`:

```
structure Rep (k G : Type u) [comm_ring k] [group G] :=
(V : Module k)
(ρ : G →* End (Module k))
```

which we will subsequently denote G-Rep_k. Since this category is equivalent to $k[G]$-Mod, it has "enough" projective objects for us to talk about the derived functor Ext. We state everything in terms of representations.

3 Formalising the standard resolution

Now we are ready to discuss the content of the project. We started by constructing the standard projective resolution of the trivial k-module k, which we will denote P from now on. For each n, its nth object is the k-module $k[G^{n+1}]$ equipped with the representation induced by the diagonal action of G on G^{n+1}. The differentials, meanwhile, are easy to define, but we have to prove that $d_n \circ d_{n+1} = 0$ for all n. There is a sense in which this was already in `mathlib`; if we can build our resolution abstractly enough to use the `mathlib` proof, we will avoid some code duplication. We summarise what this entails.

3.1 Simplicial objects

Our resolution will come from something called a simplicial object, which we now define.

▶ **Definition 9.** *The* ***simplex category*** *is the category whose objects are the totally ordered sets $[n] := \{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$, and whose morphisms are the order-preserving functions.*

In `mathlib` we just represent the objects as individual natural numbers; `simplex_category` is a type alias for $\mathbb{N}$. The category is generated by the maps $\delta_n(i), \sigma_n(i)$, where $\delta_n(i)$ is the unique order preserving injection $[n] \to [n+1]$ which misses i, and $\sigma_n(i)$ is the unique order preserving surjection $[n+1] \to [n]$ which hits i twice.

▶ **Definition 10.** *A* ***simplicial object*** *in $\mathcal{C}$ is a contravariant functor from the simplex category Δ to $\mathcal{C}$.*

If we apply a simplicial object X to $\delta_n(i)$, we get a map $X([n+1]) \to X([n])$, which we call the face maps of X. When $\mathcal{C}$ is abelian we can then define the **alternating face map complex** associated to X [11, Def 2.6]: a chain complex whose nth object is $X([n])$ and whose differential is given by the alternating sum of the face maps

$$\sum_{i=0}^{n+1} (-1)^i \cdot X(\delta_n(i))$$

which looks like the differential in our resolution, and also like the boundary maps of a topological simplicial complex. We then have the proof that this squares to zero formalised by Joël Riou, which uses the fact that $\delta_{n+1}(i) \circ \delta_n(j+1) = \delta_{n+1}(j) \circ \delta_n(i)$. But why are we using the term "face"?

This is because any simplicial set X (i.e. simplicial object in Set; we can also view any of the simplicial objects we will be concerned with as simplicial sets) has a "geometric realisation" $|X|$: there is a functor from simplicial sets to the category of compactly-generated Hausdorff topological spaces. Essentially, we replace the elements of each $X([n])$ with standard topological n-simplices Δ^n, and how they glue together depends on how X acts on morphisms [5, Section I.2]. This is where the topological interpretation of group cohomology comes from. We have

$$H^n(BG, \mathbb{Z}) \cong H^n(G, \mathbb{Z}),$$

[13, Thm 6.10.5], where the lefthand side is the topological cohomology of BG, which is the classifying space of G: the fundamental group of BG is G and its higher homotopy groups are trivial. The classifying space BG is the quotient of a contractible space EG by an action of G, and EG is determined by the structure of G; it is the universal cover of BG.

3.2 Constructing the resolution using EG

Using the comparisons described above between certain topological spaces, simplicial objects and chain complexes, we can ultimately derive our projective resolution (3) from EG, as suggested by Joël Riou here, June 3rd 2022. The author formalised his suggestion; this approach to the standard resolution ultimately involved more lines of code, but the resulting formalisation was more suited to `mathlib` in its abstraction, motivated the creation of more API for objects like the Čech nerve, and taught the author material. We sketch the maths involved. Most of the code is here; anything else is here or here. We also provide links to any key result formalised by someone else (i.e. Joël Riou).

As an overview of the strategy, we will define a simplicial object EG in the category G-Set (types with an action of G which respects multiplication in G). As a simplicial set, its geometric realization is the universal cover of BG. We can later "linearise": compose EG

with the free k-module functor from G-Set to G-Rep$_k$. Then, since G-Rep$_k$ is an abelian category, we can take the alternating face map complex associated to the resulting simplicial k-linear G-representation, which will be the standard resolution we seek. As hoped, defining the resolution using this EG gives us a proof that composition of the differentials equals zero for free. Moreover, we will show that our resolution is homotopy equivalent to $k[0]$ (the chain complex with k at 0 and 0 everywhere else) – this will define a quasi-isomorphism – and again, the simplicial approach gives us a more general proof of this than if we were to prove it directly. The homotopy equivalence's topological analogue is the contractibility of $|EG|$.

With this overview in mind, we explain the process in more detail. Given an appropriate morphism f in a category, we can define a certain simplicial object $C(f)$ called a Čech nerve. We first show that for a G-Set X, the Čech nerve of the unique morphism $X \to \top$ to the terminal object (in G-Set, this is the type with 1 term) sends $[n]$ to X^{n+1}. Now, considering G as a G-set, acting on itself by left multiplication, EG is the Čech nerve of $G \to \top$.

Now, recall that we are not only eventually defining a complex P, but also a morphism to $k[0]$ which we want to show is a homotopy equivalence. Note that since $k[0]$ is only nontrivial in degree 0, such a morphism is determined by a map $f_0 : P_0 \to k$ and a proof that $f_0 \circ d_0 = 0$, where d_0 is the last differential in P. Call the data of a chain complex and a morphism to a complex concentrated in degree 0 an "augmented chain complex"; analogously, an augmented simplicial object is a simplicial object X plus a morphism from $X([0])$ to some object Y satisfying a similar property. In a "nice" enough category, there is a natural augmentation of any simplicial object through which all other augmentations factor, which in the case of EG is given by the map $G \to \top$.

We have said that P being homotopy equivalent to $k[0]$ corresponds to $|EG|$ being contractible in the topological world; the analogue of contractibility for a simplicial object is that its natural augmentation has an **extra degeneracy**. Given a simplicial object X augmented by $f_0 : X([0]) \to Y$, an extra degeneracy is a family of maps $s : Y \to X([0])$ and $s_n : X([n]) \to X([n+1])$ for $n \geq 0$ satisfying certain properties, listed in [5, p. 200].

Now, it is a fact that the natural augmentation of the Čech nerve of a split epimorphism has an extra degeneracy 🔗, as formalised by Joël Riou. For our map of interest, $G \to \top$, to be a split epimorphism, there must be a morphism $\tau : \top \to G$ such that $\tau \circ \epsilon = \mathrm{id}$. But no such map of G-sets exists; unless G is trivial, any function $\top \to G$ does not respect the action of G. So we will have to compose EG with the forgetful functor to Set (it simply forgets the G-action), thus giving us an extra degeneracy for EG as an augmented simplicial *set*. But this is still sufficient for our purposes.

Indeed, when we forget the G-action, the resulting simplicial set is still a Čech nerve, so has an extra degeneracy. Then, when we compose with the free k-module functor, the resulting simplicial k-module is no longer a Čech nerve. Thus discarding the k-action initially was necessary for the proof strategy, and not just for the sake of generality. But it still has an extra degeneracy, as these are preserved by any functor.

Now that we are in an abelian category, we can take the alternating face map complex. The result is our standard resolution P as a complex of k-modules – we have forgotten the representation structure. Given an augmented simplicial object with an extra degeneracy, the natural augmentation of the resulting alternating face map complex is a homotopy equivalence, as formalised here 🔗, by Joël Riou. Applying this gives us a homotopy equivalence, and hence a quasi-isomorphism, of complexes of k-modules between P and $k[0]$. We need to upgrade this to a quasi-isomorphism of complexes of representations. But this amounts to showing our map of k-modules $P_0 \to k$ comes from a map of representations, and then checking properties determined on the level of sets – hence since they hold on the level of k-modules, due to our quasi-isomorphism, they also hold in G-Rep$_k$, and we are fine.

With all this in place, we can define the chain complex `group_cohomology.resolution` and its quasi-isomorphism. We define the complex to be the alternating face map complex of EG composed with the "linearisation functor" from G-Set to G-Rep$_k$, which is induced by the free k-module functor on Set.

```
def group_cohomology.resolution :=
(algebraic_topology.alternating_face_map_complex (Rep k G)).obj
  (classifying_space_universal_cover G ⋙ (Rep.linearization k G).1.1)
```

Given this definition, the objects in the complex are definitionally isomorphic to $k[G^{n+1}]$, and `simp` proves that the differential agrees with (3).

We note that up to here we have only required G to be a monoid.

3.3 Freeness of $k[G^{n+1}]$

The main remaining task is to show the objects in the resolution are projective, and for this we shall need G to be a group. Since they are not only projective, but in fact free, we show this instead. This is the only place we will use the category of $k[G]$-modules, for its free object API. We do this by first constructing the isomorphism

$$k[G] \otimes_k k[G^n] \cong k[G^{n+1}] \tag{4}$$

as representations, where the representations on $k[G^{n+1}]$ and $k[G]$ are induced by left multiplication of G, whilst $k[G^n]$ has the trivial representation. Then, passing to the $k[G]$-module category, we can send the natural k-basis of $k[G^n]$ to a $k[G]$-basis of $k[G] \otimes_k k[G^n]$, and transport this across the isomorphism. The author constructed (4) twice; first at the very start of the project, and secondly whilst writing this paper. We will review each formalisation, and compare them. The crux of the original formalisation is here 🔗, with the rest here 🔗 and here 🔗. The new formalisation is here 🔗.

Originally, we defined a map $G^n \to G^{n+1}$ which sends

$$(g_1, \dots, g_n) \mapsto (1, g_1, g_1 g_2, \dots, g_1 \dots g_n),$$

and extended this to a k-linear map $k[G] \otimes_k k[G^n] \to k[G^{n+1}]$ that sends $g \otimes (g_1, \dots, g_n)$ to $g \cdot (1, g_1, g_1 g_2, \dots, g_1 \dots g_n)$, in `of_tensor_aux`.

The type of a morphism in G-Rep$_k$ is a structure with two fields: a k-module morphism, and a proof it is compatible with the representations. If we put `of_tensor_aux` in the first field and then try to prove the statement in the second field, we get timeouts when using common tactics like `dsimp` (performs some definitional reduction, typically making the goal easier to read) and `simp`. Thus we prove the required compatibility result in a separate lemma. This difficulty surprised the author, as the objects involved seemed relatively low level. However, we are marrying some category-theoretic material (the definition of G-Rep$_k$ morphisms) and some non-category-theoretic material (everything else) – a task which has seemed to cause basic tactics to time out at other points in the project too. Meanwhile, we can state the separate lemma without category-theoretic terms, so we can prove it with our usual tactics.

Similarly, we define the inverse map `to_tensor`, which sends

$$(g_0, g_1, \dots, g_n) \mapsto g_0 \otimes (g_0^{-1} g_1, \dots, g_0^{-1} g_n),$$

and again factor out the proof of compatibility. We must also prove the two maps are left and right inverse to one another, facts we cannot leave to automation but which are not too troublesome to prove. The only other awkwardness in this formalisation was organisation:

having to name the underlying k-linear maps separately (suffixed with `aux`) and deciding when to add API for the auxiliary k-linear maps or for the G-Rep_k morphisms `of_tensor`, `to_tensor`.

In the refactor, we instead define an isomorphism of G-sets $G \times G^n \cong G^{n+1}$, with G acting by left multiplication on G^{n+1} and G but trivially on G^n. We then apply the linearisation functor G-Set $\to$ G-Rep_k. But G-Set and G-Rep_k are monoidal categories – they have a binary operation $\otimes$ on objects satisfying certain properties. In G-Set, $\otimes$ is induced by $\times$ on the underlying sets, and in G-Rep_k, $\otimes$ is induced by the usual tensor product $\otimes_k$. The linearisation functor is monoidal, meaning it commutes with $\otimes$, so we end up with $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$ as before. This approach uses more abstract tools already in `mathlib` than the first, and means we no longer have to prove that the resulting maps define morphisms in G-Rep_k. Moreover, the construction itself is more general: we define it in the simpler category G-Set, and instead of using the somewhat messy functions of the previous formalisation, we assemble it inductively from some more general building blocks. For example, for a G-set X, we define $G \times X \cong G \times X$ where G acts on the first X by the G-action `X.`ρ but on the second X trivially: the map sends $(g, x) \mapsto (g,\, \rho(g^{-1})(x))$.

However, the work left for us to do has changed: now we need to prove that the resulting isomorphism agrees with those messy maps from before. The proofs are not painless: we are proving something non-category-theoretic about objects constructed with heavy dependence on the category theory library, and as in the first version, this means avoiding `dsimp`. However, `simp` was useful when used appropriately. Additionally, these lemmas are not being factored out of some structure field or proved about some auxiliary function, so we avoid the ugly duplication of the first approach. Finally, the refactor improves performance. In the original formalisation, the representation morphism $k[G^{n+1}] \to k[G] \otimes_k k[G^n]$ takes 27 seconds to compile on the author's machine. Similarly, Lean is slow to elaborate the lemmas describing how the isormorphism acts on simple elements, despite the proofs being one line (simply using the corresponding lemmas about the underlying k-linear maps). Naïvely applying said corresponding lemmas takes about a minute to compile (on the author's machine). However, we can speed up the lemmas (though not the morphism's definition) by prefixing their proofs with `by apply`. Writing `by apply foo` achieves the same thing as writing `(foo : _)`; it makes Lean elaborate `foo` without an expected type. Otherwise, when compiling the old lemmas, it seems Lean spends too much time struggling with unification.

In the new formalisation, meanwhile, the isomorphism compiles in 5 seconds, and the lemmas describing its action on simple elements take less than 5 seconds, despite the proofs being more involved.

Regardless of its construction, given this isomorphism, we can now pass to the category of $k[G]$-modules, to transport a $k[G]$-basis of $k[G] \otimes_k k[G^n]$ across to $k[G^{n+1}]$. This requires some care, though; the category equivalence sends a G-Rep_k M to a type alias `M.`ρ`.as_module`, equipped with the $k[G]$-module instance defined $\sum n_i g_i \cdot v := \sum n_i \cdot \rho(g_i)(v)$. But taking `as_module` of the lefthand side of the isomorphism gives a $k[G]$-module structure which is only propositionally equal to the one we want in order to use the relevant $k[G]$-module API. Instead, since there is a k-module isomorphism underlying (4), we use this to define the *functions* in our $k[G]$-module isomorphism, and then prove that this commutes with the $k[G]$-action we actually want. This allows us to define the $k[G]$-basis as desired.

We have a few loose ends (a collection 🔗 of various category-theoretic details) to tie up before we can assemble our results into a term of type `ProjectiveResolution k`. These concern how certain functors interact with projectiveness and quasi-isomorphisms, and were not much trouble to formalise. Bringing together everything we have done so far allows us to define `group_cohomology.ProjectiveResolution` as hoped.

```
def group_cohomology.ProjectiveResolution :
  ProjectiveResolution (Rep.trivial k G k) :=
(ε_to_single₀ k G).to_single₀_ProjectiveResolution (X_projective k G)
```

4 Defining group cohomology

We can immediately give one definition of group cohomology. The isomorphism `functor.left_derived_obj_iso` shows that applying $\mathrm{Hom}(-, M)$ to our resolution and taking cohomology calculates $\mathrm{Ext}_{G\text{-}\mathrm{Rep}_k}(k, M)$. However, before we can finish the definition, Lean times out:

```
def group_cohomology.Ext_iso (M : Rep k G) (n : ℕ) :
  ((Ext k (Rep k G) n).obj ...).obj M ≅ ... := sorry
```

(where we omit opaque code, and the tactic `sorry` allows us to leave a declaration unfinished without (typically) giving an error). This is strange; when we replace the `sorry` with the correct isomorphism, Lean no longer times out. Meanwhile, replacing `def` with `lemma` also stops the timeout. This is what is known as the `def/lemma` issue: Lean will try and work out whether a definition is computable, even if we mark it as noncomputable, as we have done here. It is this computability check which is timing out. On the other hand, since Lean is proof irrelevant, it does not check lemmas are computable, so temporarily making the definition a lemma fixes the issue. There is now a less ad-hoc solution to this problem, due to Gabriel Ebner: prefixing a definition with `noncomputable!` will force it to be noncomputable before we have filled in the `sorry`; see [3, p. 16] for details.

But the isomorphism we really want is between cohomology of the complex in (1) and the Ext groups. First, we need an isomorphism of the objects in each complex; recall that this will come from (2). We have already defined the other isomorphism needed, $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$, when constructing a basis. Meanwhile, (2) relies on an adjunction of functors. There is some module API which would be useful for defining the adjunction, but (despite the author's, at this point, misguided efforts), this is still an inappropriate place to be working in $k[G]$-Mod, for the reasons discussed earlier 🔗. We should instead generalise, and work with the notion of a monoidal closed category.

We omit the mathematical details involved; the code can be found here 🔗. Concisely, by defining a monoidal closed instance on G-Rep_k, we get the desired adjunction – but then we must prove it behaves as it should. This means showing that when we evaluate on elements, the abstract, category-theoretic maps in our adjunction agree with the maps in the tensor-hom adjunction for k-modules, which are simpler and not defined with category theory. As in the refactor of the isomorphism $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$, we are relating a structure many layers deep in the category theory library with much simpler objects. In similar situations prior, the author had accepted slow compilation times as an inevitable part of life, without learning the "art" of using the category theory library. By this point such an approach no longer worked: `dsimp` and `simp` would often time out, or were otherwise unusably slow, and the lemmas could not be stated without use of category theory, unlike when originally defining the morphisms in (4). Moreover, the goals were too complicated to sanely close without any automation. As far as the author could tell, the user must restrict their range of proof techniques and appeal only to syntactic equalities: this means adding `rfl`-lemmas (lemmas which are true by definition, i.e. whose proof is `rfl`) to the library and rewriting these, outsourcing the work of `dsimp` to `simp` and to the user themselves. On the occasions `dsimp` does work, it would help to be able to ask which definitional equalities it applied, so the user can replace its use (since it was slowing down proofs so badly) with the corresponding `rfl`-lemmas. The tactic `squeeze_simp` tells us this regarding `simp`, and is very useful; the corresponding tactic `squeeze_dsimp` almost never works. Nonetheless, the desired results were provable and didactic for developing an instinct on how to work with category theory in Lean.

Given this, we define the complex of inhomogeneous cochains in the simplest way – by essentially translating (1) directly into Lean (with k instead of $\mathbb{Z}$). However, we also prove that each differential $\mathrm{Fun}(G^n, M) \to \mathrm{Fun}(G^{n+1}, M)$ agrees with

$$\mathrm{Fun}(G^n, M) \xrightarrow{\sim} \mathrm{Hom}_{G\text{-Rep}_k}(k[G^{n+1}], M) \xrightarrow{-\circ d_n} \mathrm{Hom}_{G\text{-Rep}_k}(k[G^{n+2}], M) \xrightarrow{\sim} \mathrm{Fun}(G^{n+1}, M)$$

where d_n is the differential in the standard resolution. This gives us for free the proof that the composition of two differentials is zero, which to do directly is somewhat onerous, as seen in work of Shenyang Wu 🔗.

With this done, there is one more obstacle to defining the isomorphism between "concrete group cohomology" and the Ext groups. On one side we have cohomology of a complex with objects in k-Mod, and on the other we have homology of a chain complex with objects in the opposite category $k\text{-Mod}^{op}$ – an instance of something we do not think about in real life but which takes a non-trivial amount of code 🔗 to formalise. However, the process was straightforward, and allows us to define group cohomology, here 🔗.

```
def group_cohomology [group G] (A : Rep k G) (n : ℕ) :=
(inhomogeneous_cochains A).homology n
```

5 Conclusion and future work

In real life, keeping exposition of a mathematical concept self-contained is a good thing, and group cohomology is quite amenable to this. But the trajectory of our project demonstrates just how irrelevant this quality is as an aim when contributing to `mathlib`. Indeed, the need to prioritise generality, to keep the growth of a library sustainable, means recognising and exploiting as many connections between different mathematical objects as possible, in search of concepts' common principles and structural "ancestors".

But the quest for abstraction has to stop somewhere. All of this material has been merged with `mathlib`: we conclude it is possible to honour the maxims of `mathlib` design to a considerable extent, and still connect the resulting convoluted, abstract definition of group cohomology with the down to earth definition used for computation. The power afforded by results in the category theory library can, in practice, interact with `mathlib`'s lower-level objects.

However, we have illustrated this statement's caveats. Currently, it seems to the author that there is an "art" to using the category theory library, meaning it can be frustrating to work with for the naïve user. Of course, exactly the same could be said of Lean in general; a learning curve is unavoidable. But as more people apply category theory to simpler structures in the library, documentation of this "art" will increase.

Alternatively, the new version of Lean, Lean 4, promises many advances in performance, and this could make it easier to use automation with category theory. This project was done in Lean 3, which has been the most current version for most of Lean's history. But this is changing – `mathlib` is currently being ported from Lean 3 to Lean 4: a huge undertaking. When our project eventually transitions to Lean 4, we suspect it may look very different, with the obstacles outlined in this paper perhaps diminished.

In the meantime, there is considerably more to be added before `mathlib` has the facts about group cohomology taught in a typical introductory course. Most of these are proved using the concrete formulation, and the author has a repository containing work in this direction. Because there is not much structure involved, the high-powered tools of category theory are irrelevant, making the results simple to formalise. Thus, although this code has not been tidied up with `mathlib` in mind, the author is fairly confident it will not go through as many reformulations as the rest of the material so far.

We have written an API for cohomology in degree $n \leq 2$, and will open a pull request for this soon. We have also shown that given a group homomorphism $f : G \to H$, a G-representation A, an H-representation B and a k-linear map $\phi : B \to A$ such that $\phi(\rho_B(f(g))(x)) = \rho_A(g)(\phi(x))$ for all $g \in G, x \in B$, then we get an induced k-linear map of cohomology groups $\mathrm{H}^n(H, B) \to \mathrm{H}^n(G, A)$ for all n. We used this to formalise the "inflation-restriction" exact sequence, and have also formalised Hilbert's theorem 90, along with the fact that $\mathrm{H}^1(G, A) \cong \mathrm{Hom}(G, A)$ when the G-action on A is trivial, and about half of the work in using H^2 to classify group extensions. The code is fast enough and readable; the only disappointment is that in real life we can view a group's operation as either multiplicative or additive when convenient, and this is messier in Lean. A representation A is an additive group with an action of a multiplicative group, so to describe the set of group morphisms $\mathrm{Hom}(G, A)$ we must write `G →* multiplicative A`, for example.

Similarly, we have done some non-`mathlib`-style work on Galois cohomology. Many nice group cohomology facts assume G is finite; Galois groups are profinite, meaning they are limits of families of finite groups. Proving that the group cohomology of a profinite group is a limit of the cohomology of the constituent finite groups lets us extend results to Galois groups, and this is used everywhere in algebraic number theory. To formalise this, the author has proved some of the requisite topological group facts; similar to the concrete group cohomology API, the code was enjoyable to write. Instead of complicated definitions, it involves complicated proofs, which can be preferable in Lean.

A different story is the remaining abstract material, like universal delta functors and spectral sequences. Spectral sequences [9, Chapter 20, Section 9] will require considerable work to define at all, let alone in a `mathlib`-compatible way. These are necessary to compare cohomology as we vary the group G via the Lyndon-Hochschild-Serre spectral sequence [6, p. 8] – an extension of the "inflation-restriction" exact sequence. We need delta functors, meanwhile, to finish setting up Galois cohomology [12, Tag 0DVG]. They are also needed to show that group cohomology agrees with Ext as functors, and not just in their action on objects [9, Chapter 20, Section 8]. Happily, though, the requisite delta functor material is done, in the Liquid Tensor Experiment 🔗; it has just not been readied for `mathlib`, and given its abstract nature this process may be non-trivial.

All of the abstract material, and essentially any group cohomological results concerning $\mathrm{H}^n(G, M)$ for general n, rely on long exact sequences. These, too, are defined in the Liquid Tensor Experiment, and should be usable in our work after proving a couple of easy, concrete lemmas. But even preparing these for `mathlib` raises challenges – when using them in real life we rely on drawing diagrams, and the clarity this affords is lost when translated into Lean. This file 🔗 gives some demonstration of what "diagram chasing" can look like in Lean 3. However, Wojciech Nawrocki is working on a widget 🔗 for Lean 4 which displays commutative diagrams in the goal state; maybe this will help us chase diagrams in the future.

References

1 Anne Baanen. Use and Abuse of Instance Parameters in the Lean Mathematical Library. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2022.4`.

2 John W. S. Cassels and Albrecht Fröhlich. *Algebraic Number Theory*. London Mathematical Society, London, 2010.

3 María Inés de Frutos-Fernández. Formalizing the Ring of Adèles of a Global Field. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237, pages 14:1–14:18, 2022. URL: `https://drops.dagstuhl.de/opus/volltexte/2022/16723/pdf/LIPIcs-ITP-2022-14.pdf`.

4 César Domínguez and Julio Rubio. Computing in coq with infinite algebraic data structures. In *Proceedings of the 10th ASIC and 9th MKM International Conference, and 17th Calculemus Conference on Intelligent Computer Mathematics*, volume 6167, April 2010. `doi:10.1007/978-3-642-14128-7_18`.

5 Paul G. Goerss and John F. Jardine. *Simplicial Homotopy Theory*. Springer Science & Business Media, 2009. `doi:10.1007/978-3-0346-0189-4`.

6 Gerhard Hochschild and Jean-Pierre Serre. Cohomology of group extensions. *Transactions of the American Mathematical Society*, 74(1):110–134, 1953. `doi:10.1090/S0002-9947-1953-0052438-8`.

7 Shin-ichi Katayama. Diophantine Equations and Hilbert's Theorem 90. *Journal of mathematics, the University of Tokushima*, 48:35–40, 2014. URL: `https://cir.nii.ac.jp/crid/1574231877578024320`.

8 Ernst E. Kummer. Über eine besondere Art, aus complexen Einheiten gebildeter Ausdrücke. *Journal für die reine und angewandte Mathematik*, 1855(50):212–232, 1855. `doi:10.1515/crll.1855.50.212`.

9 Serge Lang. *Algebra*, volume 211. Springer Science & Business Media, 2012. `doi:10.1007/978-1-4613-0041-0`.

10 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373824`.

11 nLab authors. Moore complex. `https://ncatlab.org/nlab/show/Moore+complex`, February 2023. Revision 59.

12 The Stacks project authors. The stacks project. `https://stacks.math.columbia.edu`, 2023.

13 Charles A. Weibel. *An Introduction to Homological Algebra*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1994. `doi:10.1017/CBO9781139644136`.

14 Charles A. Weibel. History of Homological Algebra. In Ioan M. James, editor, *History of Topology*, chapter 28, pages 797–836. North Holland, 1999.