Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

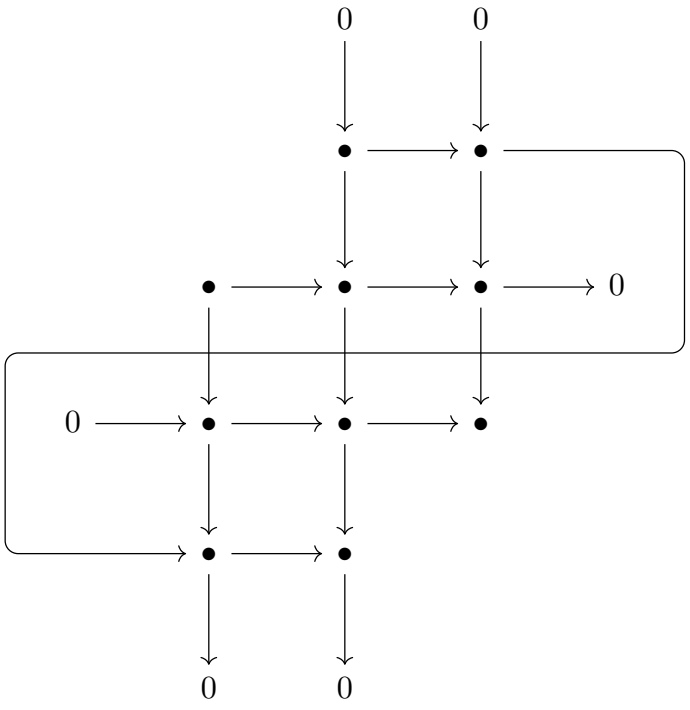
Lehrstuhl Prof. Dr.-Ing. Snelting

Diagram Chasing in Interactive Theorem Proving

Bachelorarbeit von

Markus Himmel

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Sebastian Ullrich
Abgabedatum:	17. April 2020

2010 *Mathematics Subject Classification.* 03B35, 18E10, 68T15

Zusammenfassung

Wir erweitern die mathematische Bibliothek des interaktiven Theorembeweisers Lean um Definitionen und erste Resultate über abelsche Kategorien. Insbesondere formalisieren wir die Theorie der Pseudoelemente, welche die Beweistechnik der Diagrammjagd in einem allgemeinen kategoriellen Kontext zugänglich macht. Wir nutzen die in Lean verfügbaren Methoden der Metaprogrammierung zur Entwicklung einer interaktiven Taktik, die Exaktheitsaussagen in abelschen Kategorien durch Diagrammjagd mit Pseudoelementen halbautomatisch beweisen kann. Mithilfe dieser Automatisierung geben wir den ersten formal verifizierten Beweis des Schlangenlemmas, einem wichtigen Werkzeug der homologischen Algebra, das aufgrund seiner Rolle in der Konstruktion der langen exakten Sequenz in Kohomologie in vielen Bereichen der reinen Mathematik und darüber hinaus Anwendung findet.

We extend the mathematical library of the Lean interactive theorem prover with basic definitions and results about abelian categories. Within this theory, we formalize the notion of pseudoelement, which generalizes the notion of element in an abelian group to a general arrow-theoretic setting and gives access to the proof technique of diagram chasing. Using the metaprogramming framework of the Lean theorem prover, we develop tactics that semiautomatically synthesize proofs of statements about commutative diagrams in abelian categories using pseudoelements. Using this, we give the first formally verified proof of the snake lemma, an important tool in homological algebra with ample applications in pure mathematics and beyond.

Contents

Introduction

Studying complex mathematical objects directly can be immensely difficult due to their rich and intricate structure. For this reason, it is often beneficial to associate to a complex object a simpler object that can be more readily studied while still allowing for something to be learned about the original object.

One kind of simple object we can use is given by the (very general) concept of *cohomology*: First, we choose a collection $(C^i)_{i\in\mathbb{Z}}$ of algebraic objects such as abelian groups that in some way encode the structure of the object we are interested in. We also choose homomorphisms $\delta^i : C^i \to C^{i+1}$ such that $\delta^{i+1} \circ \delta^i = 0$. The i-th cohomology group is now defined as $H^i \coloneqq \operatorname{Ker} \delta^i / \operatorname{Im} \delta^{i-1}$.

This general recipe can be used to study a wide range of mathematical objects: We just have to find good definitions for C^i and δ^i.

Cohomology was originally defined in the 1940s as an algebraic invariant of topological spaces. Since then, this notion has spread to a wide variety of mathematical disciplines and is now ubiquitous in many areas of modern pure mathematics. Cohomology has been a key tool in many recent advances in mathematics such as Wiles' resolution of Fermat's conjecture [28] or the development of modern algebraic geometry by Grothendieck and his collaborators.

The mathematical discipline called *homological algebra* studies the properties of the construction mentioned above, without referencing the concrete definitions of C^i and δ^i. In this way, homological algebra provides tools that are useful for working with all kinds of cohomology theories that appear in mathematics.

Some of the most fundamental tools of homological algebra are the so-called *diagram lemmas*, which allow us to derive properties of morphisms from properties of morphisms we already know, provided they somehow relate to the morphism we are interested in. The proof technique commonly employed for these lemmas is called *diagram chasing*. Here is what Paolo Aluffi has to say about the proof of one of the diagram lemmas, called the *snake lemma*:

> Proving the snake lemma is something that should not be done in public, and it is notoriously useless to write down the details of the verification for others to read: the details are all essentially obvious, but they lead quickly to a notational quagmire. Such proofs are collectively known as the sport of *diagram chase*, best executed by pointing several fingers at different parts of a diagram on a blackboard, while enunciating the elements one is manipulating and stating their fate. [1, p. 180]

This seems like an almost perfect match for some kind of automated proof. Indeed, a common claim about proofs by diagram chase is that at any point, there is really only one possible next step in the proof. If that is true, it should even be possible for a computer to perform a complete search over the possible proofs and find the desired proof.

The natural setting to study this question in is interactive theorem proving. In an interactive theorem prover one can state definitions and theorems and then give

proofs of these theorems by providing hints to the system on how to construct a proof of the claim in the system's underlying logic. These systems are general-purpose and allow for the formal development of entire mathematical theories. Since these lemmas are really tools rather than interesting statements in their own right, this is clearly a desirable feature: We will be able to build the tools of homological algebra and then also apply them where they are needed.

In this thesis, we present a formalization of the technique of diagram chasing in a general mathematical setting. To be able to state the diagram lemmas at the correct level of generality, we have developed the theory of abelian categories and have formalized the tools that are necessary to perform proofs by diagram chasing in general abelian categories. Furthermore, we have implemented proof automation to perform these proofs semi-automatically.

The thesis is structured as follows. In Chapter 1, we give an overview over the mathematical prerequisites that are needed in this thesis. We also describe Lean, the interactive theorem prover we have used. Chapters 2 and 3 contain our results. Chapter 2 describes our formal theory of abelian categories and in Chapter 3 we describe the tactic for semi-automatic diagram chasing that we have developed as well as some of the results we were able to show using it.

This thesis only contains a subset of the results we have formalized. Many results, and almost all proofs, have been omitted for reasons of brevity. For the full formalization, we refer to the source code. This thesis corresponds to revision `125b4b1d` of the source code repository, which may be accessed at

`https://github.com/TwoFX/lean-homological-algebra/tree/125b4b1dabdf7df88f517954ace6a6aa9f7054c8`.

It is based on revision `0567b7fa` of the Lean mathematical library.

CHAPTER 1

Background

1. Mathematical preliminaries

We will assume results present in a typical introductory course on linear algebra. In particular, it will be useful to know what the kernel of a linear map between vector spaces is. For a refresher, we recommand any textbook on (linear) algebra, for example the textbook by Aluffi [1], which also contains much of the mathematics touched upon in the rest of this thesis.

1.1. Diagrams of modules.

DEFINITION. Let R be a ring. An (additive) abelian group M together with a map $\bullet\colon R \times M \to M$ is called a *left R-module* if it satisfies the following properties:

i) $\forall m \in M\colon 1 \bullet m = m$,
ii) $\forall r, s \in R, m \in M\colon rs \bullet m = r \bullet (s \bullet m)$,
iii) $\forall r, s \in R, m \in M\colon (r + s) \bullet m = r \bullet m + s \bullet m$,
iv) $\forall r \in R, m, n, \in M\colon r \bullet (m + n) = r \bullet m + r \bullet n$.

The operation $\bullet$ is called *scalar multiplication*. If M and N are R-modules, a map $f\colon M \to N$ is called R-linear if

$$\forall r, s \in R, m, n \in M\colon f(r \bullet m + s \bullet n) = r \bullet f(m) + s \bullet f(n).$$

We denote the set of R-linear maps from M to N by $\mathrm{Hom}(M, N)$.

REMARK. The definition of a module is exactly equal to the definition of a vector space, except that we do not require R to be a field. It turns out that the theory of modules is in many way much more intricate than the study of vector spaces. One key reason is that not every module has a basis.

We shall use terms known from linear algebra, such as kernel and image of linear maps. A submodule is defined in the same way as a subspace. A quotient module is defined in the same way as a quotient space.

REMARK. Every additive abelian group G becomes a $\mathbb{Z}$-module in a unique way: Indeed, assume that G has been equipped with a scalar multiplication making it into a $\mathbb{Z}$-module and let $g \in G$. We have $1 \bullet g = g$. Furthermore, $0 \bullet g = (0+0) \bullet g = 0 \bullet g + 0 \bullet g$, which implies that $0 \bullet g = 0$. Finally, we compute $0 = 0 \bullet g = (1-1) \bullet g = 1 \bullet g + (-1) \bullet g$, which implies $(-1) \bullet g = -(1 \bullet g) = -g$. These identities together with the third axiom determine the module structure completely: It is given by $n \bullet g = g + \cdots + g$, where there are n summands. It is easy to see that a $\mathbb{Z}$-module structure can always be defined in this way.

Using this observation, module theory can be viewed as a generalization of the theory of abelian groups. Indeed, many interesting results about abelian groups generalize nicely to modules. For example, the classification of finitely generated abelian groups (cf. [1, Corollary IV.6.5]) is a special case of the classification of finitely generated modules over a principal ideal domain.

DEFINITION. Let R be a ring and $\mathcal{M}$ be a set of R-modules. A *commutative diagram* of R-modules is a directed multigraph $G = (V, E)$ together with a function

$M\colon V \to \mathcal{M}$ and a function $f\colon E \to \bigcup_{M,N\in\mathcal{M}} \operatorname{Hom}(M,N)$ satisfying the following properties:

i) If $e \in E$ is an edge from u to v, then $f(e) \in \operatorname{Hom}(M(v), M(v))$,
ii) If $(e_1, e_2, \dots, e_n)$ and $(\hat{e}_1, \dots, \hat{e}_m)$ are paths in G from u to v, then

$$f(e_n) \circ f(e_{n-1}) \circ \cdots \circ f(e_1) = f(\hat{e}_m) \circ \cdots \circ f(\hat{e}_1).$$

EXAMPLE. Using the language of commutative diagrams, the statement "Let A, B, C, D be R-modules and $f\colon A \to B$, $g\colon C \to D$, $\alpha\colon A \to C$, $\beta\colon B \to D$ be R-linear maps satisfying $\beta \circ f = g \circ \alpha$." can be rephrased as "Consider the following commutative diagram of R-modules:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\beta} \\ C & \xrightarrow{g} & D \end{array}$$

".

DEFINITION. Let R be a ring, and let

$$M_0 \xrightarrow{f_0} M_1 \xrightarrow{f_1} M_2 \xrightarrow{f_2} \dots \xrightarrow{f_{n-1}} M_n$$

be a sequence of modules and R-linear maps. We say that the sequence is *exact at* M_i if $\operatorname{Ker} f_i = \operatorname{Im} f_{i-1}$. In this case, we may also call (f_{i-1}, f_i) an *exact pair*. We say that the sequence M is exact if it is exact at $M_1, \dots, M_{n-1}$. This also generalizes to infinite sequences.

A *short exact sequence* is an exact sequence of the form

$$0 \longrightarrow A \xrightarrow{\iota} B \xrightarrow{\pi} C \longrightarrow 0,$$

where 0 is the trivial module $0 \coloneqq \{0\}$. The reader is encouraged to think about why the maps might be called ι and π.

A *long exact sequence* is an infinite sequence $(M_i)_{i\in\mathbb{Z}}$, $(f_i\colon M_i \to M_{i+1})_{i\in\mathbb{Z}}$, which is exact everywhere.

EXAMPLE. It should by now be obvious what is meant when we say that the following commutative diagram has *exact rows*.

$$\begin{array}{ccccccccc} 0 & \longrightarrow & A & \xrightarrow{\iota} & B & \xrightarrow{\pi} & C & \longrightarrow & 0 \\ & & \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\beta} & & \downarrow{\scriptstyle\gamma} & & \\ 0 & \longrightarrow & A' & \xrightarrow{\iota'} & B' & \xrightarrow{\pi'} & C' & \longrightarrow & 0 \end{array}$$

1.2. A very short primer on algebraic topology. This section is an attempt to provide some form of motivation as to why the rather abstract algebraic objects considered in this thesis might be interesting, and how the seemingly arbitrary statements of the diagram lemmas appear in applications. We will not need any information from this section in the remainder of this thesis, so a reader who is not interested in applications may skip this section. The definitions and results mentioned here can be found in any introductory textbook on algebraic topology, for example the book by May [17]. We will attempt to convey a general picture and will not give precise definitions or proofs.

Algebraic topology attempts to *classify spaces up to deformation*. By "spaces" we mean topological spaces, which provide a general framework in which the concept of "continuity" can be studied. The fundamental question is "Given two spaces X

and Y, are they the same?", where by "the same" we mean *homotopy equivalent*, which roughly means that we can continuously transform one space into the other by "squishing", but not by "tearing".

Our method of tackling this question is by assigning to every space algebraic objects which are invariant under homotopy equivalence. If two spaces have different invariants, then they are not homotopy equivalent.

Among these algebraic invariants are the *(reduced singular) homology modules* $H_n(X;R)$ and the *(reduced singular) cohomology modules* $H^n(X;R)$ associated to a space X, one for each $n \in \mathbb{N}_0$. The n-th (co)homology module somehow encodes the n-dimensional structure of the space X. For example, if $n > 0$ and $S^n = \{x \in \mathbb{R}^{n+1} \mid \|x\| = 1\}$ is the n-sphere, then we have $H_i(S^n;\mathbb{Z}) = \mathbb{Z}$ for $i = n$ and $H_i(S^n;\mathbb{Z}) = 0$ otherwise.

The definition of the homology modules requires some machinery, which makes it difficult to directly compute the (co)homology of complicated spaces. The solution is to compute the homology of simple spaces (such as the space consisting of a single point, or spheres) and prove theorems that allow us to compute the homology of spaces which are assembled from simpler spaces whose homology we already know. For example, the torus T^2 can be written as the cartesian product $T^2 = S^1 \times S^1$ of two circles. So if we know the homology of the circle, and we know how to compute the homology of cartesian products, then we also know the homology of the torus. Similar considerations hold for cohomology.

An example of one of these tools is the so-called *Meyer-Vietoris sequence in reduced homology*. Let X be a topological space, and let A and B be subspaces of X such that the union of the interiors of A and B is X (for example, if X is the sphere, then we could choose A as a little more than the northern hemisphere of the sphere, and B as a little more than the southern hemisphere of the sphere). The Meyer-Vietoris sequence is a long exact sequence of the form

$$\begin{aligned}\ldots \to H_{n+1}(A \cap B) \to H_{n+1}(A) \oplus H_{n+1}(B) \to H_{n+1}(X) \to H_n(A \cap B) \to \\ H_n(A) \oplus H_n(B) \to H_n(X) \to H_{n-1}(A \cap B) \to \ldots \to H_0(X) \to 0 \to 0 \to \ldots\end{aligned}$$

In many cases, this lets us compute the homology of X if we already know the homology of A and B. For example, if we were able to choose A and B such that $A \cap B$ satisfies $H_n(A \cap B) = 0$ for all $n \geq 0$, then exactness implies that $H_n(X) \cong H_n(A) \oplus H_n(B)$ for all $n \geq 0$.

Another important tool for calculating invariants of spaces are *induced maps*. Constructions like (co)homology are often functorial: If $f\colon X \to Y$ is a continuous map of topological spaces, then we get a well-behaved map $H_n(f)\colon H_n(X) \to H_n(Y)$ for every n.

To demonstrate why this is useful, we sketch a proof of Brouwer's fixed point theorem in dimension 2. Let

$$D^2 = \{x \in \mathbb{R}^2 \mid \|x\| \leq 1\}$$

be the 2-dimensional disk. We can squish the disk to a point, so the disk and the point are homotopy equivalent. Using this, we hope the reader thrusts us that it is easy to compute directly using the definition that $H_1(D^2) = 0$. We have noted earlier that $H_1(S^1) = \mathbb{Z}$.

THEOREM. Let $h\colon D^2 \to D^2$ be a continuous map. Then there is some $x \in D^2$ such that $h(x) = x$.

PROOF. Assume the contrary. Let $x \in D^2$. Consider the ray starting at $h(x)$ going through x. Let $r(x)$ be the point in S^1 where this ray leaves D^2 (cf. Figure 1). It is easy to see that this gives a continuous map $r\colon D^2 \to S^1$ which satisfies $r(x) = x$

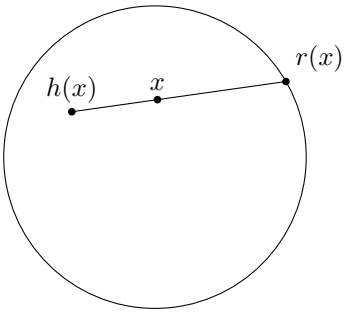


FIGURE 1. Illustration of the definition of r in the proof of the Brouwer fixed point theorem.

for every $x \in S^1$. Let $i\colon S^1 \to D^2$ be the inclusion. Then $r \circ i$ is the identity on S^1 as we just noted. By functoriality of $H_1(-;\mathbb{Z})$, this means that the composite

$$H_1(S^1;\mathbb{Z}) \xrightarrow{H_1(i;\mathbb{Z})} H_1(D^2;\mathbb{Z}) \xrightarrow{H_1(r;\mathbb{Z})} H_1(S^1;\mathbb{Z})$$

is the identity on $H_1(S^1;\mathbb{Z})$, but this is of course absurd: Since $H_1(D^2;\mathbb{Z}) = 0$, the composite is actually the zero map. □

Amazingly, tools like the Meyer-Vietoris sequence interact nicely with induced morphisms: Let (X, A, B) and (Y, A', B') be triples of spaces that fulfill the requirements of Meyer-Vietoris. If $f\colon X \to Y$ is a continuous map such that $f(A) \subseteq A'$ and $f(B) \subseteq B'$, then we get an infinite commutative diagram with exact rows

$$\begin{array}{ccccccccc}
\dots & \longrightarrow & H_n(A) \oplus H_n(B) & \longrightarrow & H_n(X) & \longrightarrow & H_{n-1}(A \cap B) & \longrightarrow & \dots \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
\dots & \longrightarrow & H_n(A') \oplus H_n(B') & \longrightarrow & H_n(Y) & \longrightarrow & H_{n-1}(A' \cap B') & \longrightarrow & \dots,
\end{array}$$

where the vertical maps are all induced by f. A common situation that arises when computing homology using such a long exact sequence is that we already know that f induces isomorphisms $H_n(f)\colon H_n(A) \to H_n(A')$, $H_n(f)\colon H_n(B) \to H_n(B')$ and $H_n(f)\colon H_n(A \cap B) \to H_n(A' \cap B')$. In other words, the subspaces that X and Y are assembled from have the same homology. A natural question now is: Does this already imply that X and Y have the same homology? Indeed, this is the case. Perhaps surprisingly, the reason for this is purely algebraic: It directly follows from the five lemma, which is one of the diagram lemmas we will discuss now.

1.3. Diagram chasing in the category of modules.

LEMMA (Five lemma). Consider the following commutative diagram of R-modules with exact rows:

$$\begin{array}{ccccccccc}
A & \xrightarrow{f} & B & \xrightarrow{g} & C & \xrightarrow{h} & D & \xrightarrow{i} & E \\
\downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\beta} & & \downarrow{\scriptstyle\gamma} & & \downarrow{\scriptstyle\delta} & & \downarrow{\scriptstyle\varepsilon} \\
A' & \xrightarrow{f'} & B' & \xrightarrow{g'} & C' & \xrightarrow{h'} & D' & \xrightarrow{i'} & E'
\end{array}$$

If α, β, δ and ε are isomorphisms, then so is γ.

The five lemma is a corollary of the two four lemmas.

LEMMA (Four lemma, version 1). Consider the following commutative diagram of R-modules with exact rows:

$$\begin{array}{ccccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C & \xrightarrow{h} & D \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \delta} \\ A' & \xrightarrow{f'} & B' & \xrightarrow{g'} & C' & \xrightarrow{h'} & D' \end{array}$$

If α is surjective and β and δ are injective, then γ is injective.

This is one of the classic results where the reader really should come up with the proof themselves. Since the four lemma will serve as a running example for the rest of this document, we will give a proof anyway. However, we will defer this until Section 3.2, so that we can give a formally verified proof.

Here is the second four lemma:

LEMMA. If α and γ are surjective and δ is injective, then β is surjective.

The proof of the second four lemma is similar in style to the proof of the first four lemma, and once again, the reader is encouraged to fill it in themselves[1]. We will not give a proof here.

It turns out that these diagrammatic situations appear in a wide range of contexts, and many very important appliations of this theory do not live in the world of modules. For this reason, we will develop our theory in a more general context: Abelian categories. We give a quick summary of the categorical notions that will be important to us.

1.4. The bare necessities of category theory. The material in this section can be found in any textbook on basic category theory, such as the first volume of Borceux [4].

1.4.1. *Categories.*

DEFINITION. A *category* C consists of

- a collection of *objects* $\mathrm{Ob}(\mathsf{C})$,
- for every pair of objects X, Y a collection of *morphisms* $\mathrm{Hom}(X, Y)$, and
- for every triple of objects X, Y, Z a binary operation

$$\gg\colon \mathrm{Hom}(X,Y) \times \mathrm{Hom}(Y,Z) \to \mathrm{Hom}(X,Z)$$

such that

- $\gg$ is associative, and
- for every object X there is some morphism $\mathbf{1}_X \in \mathrm{Hom}(X,X)$ such that for every object Y and morphisms $f \in \mathrm{Hom}(X,Y)$, $g \in \mathrm{Hom}(Y,X)$ we have $\mathbf{1}_X \gg f = f$ and $g \gg \mathbf{1}_X = g$.

We will write $X \in \mathsf{C}$ for $X \in \mathrm{Ob}(\mathsf{C})$ and $f\colon X \to Y$ for $f \in \mathrm{Hom}(X,Y)$.

REMARK. Most authors follow a different notation and write $g \circ f$ instead of $f \gg g$. We adopt the rather unusual notation $f \gg g$, which is sometimes called "diagrammatic order", in order to be consistent with the formal library of category theory we will build upon (see Section 3).

DEFINITION. Let $f\colon X \to Y$ be a morphism in a category C.

a) We call f a *monomorphism* if for all $g, h\colon Z \to X$ we have the implication $g \gg f = h \gg f \implies g = h$. In diagrams, we will sometimes write a monomorphism as $f\colon X \hookrightarrow Y$. It is useful (but not always accurate) to think of monomorphisms as a generalization of injective functions.

[1]Here is a hint: It will be beneficial to use the additive structure of modules.

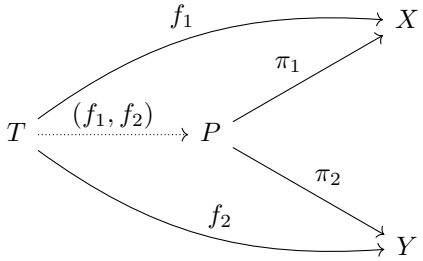
b) We call f an *epimorphism* if for all $g, h\colon Y \to Z$ we have the implication $f \gg g = f \gg h \implies g = h$. In diagrams, we will sometimes write an epimorphism as $f\colon X \twoheadrightarrow Y$. It is useful (but not always accurate) to think of epimorphisms as a generalization of surjective functions.
c) We call f an *isomorphism* if there is some $g\colon Y \to X$ such that $g \gg f = \mathbf{1}_Y$ and $f \gg g = \mathbf{1}_X$. We write $f^{-1} \coloneqq g$.

REMARK. f^{-1} is well-defined, since if g' has the same properties as g, then $g = \mathbf{1}_Y \gg g = g' \gg f \gg g = g' \gg \mathbf{1}_X = g'$.

It is obvious that an isomorphism is both a monomorphism and an epimorphism. The converse is false in general, but it is true for abelian categories, which we will define later.

1.4.2. *Universal properties.* A central notion in algebra and category theory is that of a universal property. The concept of universal property is best illustrated by example.

DEFINITION. Let C be a category and let $X, Y \in \mathsf{C}$. We call an object $P \in \mathsf{C}$ together with morphisms $\pi_1\colon P \to X$ and $\pi_2\colon P \to Y$ a *product of X and Y* if for every $T \in \mathsf{C}$ and $f_1\colon T \to X$, $f_2\colon T \to Y$ there is a unique morphism $(f_1, f_2)\colon T \to P$ making the following diagram commute:



EXAMPLE. If X and Y are sets, then the cartesian product $X \times Y$ is a categorical product of X and Y.

LEMMA. If (P, π_1, π_2) and (P', π_1', π_2') are products of X and Y, then there is an isomorphism $i : P \to P'$.

PROOF. By the universal property of P, there is a unique morphism $i\colon P' \to P$ such that $i \gg \pi_1 = \pi_1'$ and $i \gg \pi_2 = \pi_2'$. Similarly, by the universal property of P', we find a unique $j\colon P \to P'$ such that $j \gg \pi_1' = \pi_1$ and $j \gg \pi_2' = \pi_2$. Now we may calculate $j \gg i \gg \pi_1 = j \gg \pi_1' = \pi_1$ and analogously $j \gg i \gg \pi_2 = \pi_2$.

By the universal property of P, there is a unique morphism $k\colon P \to P$ such that $k \gg \pi_1 = \pi_1$ and $k \gg \pi_2 = \pi_2$. We have just seen that $j \gg i$ has this property, so by uniqueness $j \gg i = k$. On the other hand, $\mathbf{1}_P$ also has this property, so by uniqueness, $\mathbf{1}_P = k$, and we conclude $j \gg i = \mathbf{1}_P$.

By the same argument, $i \gg j = \mathbf{1}_{P'}$. Therefore, i is an isomorphism as required. □

REMARK. The previous definition and proof are part of a larger story: *Limits*. Defining limits requires a bit of machinery, but since we will make heavy use of limits in Chapter 2, we give the definition here. For a much more detailed and leisurely exposition of this material, we refer to Chapter 2 of Borceux [4].

DEFINITION. Let J and C be categories.

a) A *functor* $F\colon \mathsf{J} \to \mathsf{C}$ assigns to every object $J \in \mathsf{J}$ an object $FJ \in \mathsf{C}$ and to every morphism $j\colon J \to J'$ a morphism $Fj\colon FJ \to FJ'$ such that $F\mathbf{1}_J = \mathbf{1}_{FJ}$ for every $J \in \mathsf{J}$ and $F(j \gg j') = Fj \gg Fj'$ for every pair

of composable morphisms j and j'. A functor $F\colon \mathsf{J} \to \mathsf{C}$ is also called a *diagram in* C *of shape* J.

b) If $F\colon \mathsf{J} \to \mathsf{C}$ is a diagram in C, then a *cone* c *on* F is an object $X \in \mathsf{C}$ together with morphisms $c_J\colon X \to FJ$ for every $J \in \mathsf{J}$ such that for all $j\colon J \to J'$ we have $c_J \gg Fj = c_{J'}$. The object X is called the *vertex* of c.

c) If $F\colon \mathsf{J} \to \mathsf{C}$ is a diagram in C and c is a cone on F with vertex X, then we call c a *limit cone* if for every cone d on F with vertex X' there is a unique morphism $t\colon X' \to X$ such that $t \gg c_J = d_J$ for every $J \in \mathsf{J}$.

d) If $F\colon \mathsf{J} \to \mathsf{C}$ is a diagram in C, then a *cocone* of F is an object $X \in \mathsf{C}$ together with morphisms $c_J\colon FJ \to X$ for every $J \in \mathsf{J}$, such that for all $j\colon J \to J'$ we have $Fj \gg c_{J'} = c_J$. The object X is called the vertex of c.

e) If $F\colon \mathsf{J} \to \mathsf{C}$ is a diagram in C and c is a cocone on F with vertex X, then we call c a *colimit cocone* if for every cocone d on F with vertex X there is a unique morphism $t\colon X \to X'$ such that $c_J \gg t = d_J$ for every $J \in \mathsf{J}$.

Remark. Not every diagram admits a limit cone. If a limit cone exists, then it is unique in the following sense: If c and d are two limit cones with vertices X and X' over $F\colon \mathsf{J} \to \mathsf{F}$, then there is a unique isomorphism $t\colon X \to X'$ satisfying $t \gg c_J = d_J$ for every $J \in \mathsf{J}$. The proof of this statement works exactly like the proof of uniqueness of products that we gave earlier.

Notation. We will use small diagrams for writing the index category J, where bullets denote objects and arrows denote morphisms. For example, the notation $\mathsf{J} = \bullet \to \bullet \leftarrow \bullet$ means that that J has three objects X, Y and Z, and two non-identity morphisms, $f\colon X \to Y$, and $g\colon Z \to Y$.

Definition.

a) A limit of shape J, where J is the empty category, is called an *initial object*. A colimit of shape J, where J is the empty category, is called a *terminal object*.

b) A limit of shape $\bullet\ \bullet$ is called a *product*. The product of two objects X and Y is written $X \times Y$ or $X \amalg Y$, or, more correctly, $(\prod_{J\in\mathsf{J}} FJ, (\pi_J)_{J\in\mathsf{J}})$, since the limit really consists of the vertex *and* the morphisms comprising the cone. There is an easy generalization to products larger families of objects.

In the case of binary products, if f and g are morphisms with a common domain Z and codomain $X \times Y$, we will write (f, g) for the unique morphism $(f, g)\colon Z \to X \times Y$ satisfying $f = (f, g) \gg \pi_X$ and $g = (f, g) \gg {\pi_Y}$[2].

c) A colimit of shape $\bullet\ \bullet$ is called a *coproduct*. The coproduct of two objects X and Y is written $X \amalg Y$.

d) A limit of shape $\bullet \rightrightarrows \bullet$ is called an *equalizer*.

e) A colimit of shape $\bullet \rightrightarrows \bullet$ is called a *coequalizer*.

f) A limit of shape $\bullet \to \bullet \leftarrow \bullet$ is called a *pullback*.

g) A colimit of shape $\bullet \leftarrow \bullet \to \bullet$ is called a *pushout*.

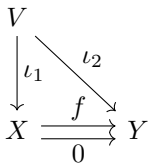
Definition. An object $0 \in \mathsf{C}$ of a category C is called a *zero object* if it is both initial and terminal.

Definition. We say that a category C *has zero morphisms* if for all $P, Q \in \mathsf{C}$ the collection $\mathrm{Hom}(P, Q)$ is a pointed set containing a special morphism $0\colon P \to Q$ such that $0 \gg f = 0$ and $g \gg 0 = 0$ for all composable f and g.

[2]Note the abuse of notation: In the subscript of π, we have used objects of C instead of objects of J.

DEFINITION. If C is a category with zero morphisms, then an equalizer of $f\colon X \to Y$ and $0\colon X \to Y$ is called a *kernel* of f. A coequalizer of f and 0 is called a *cokernel* of f.

IMPORTANT REMARK. Recall that due to the definition of a limit, a kernel of a morphism $f\colon X \to Y$ is really a cone, which consists of a vertex $V \in C$ as well as two morphisms $\iota_1\colon V \to X$ and $\iota_2\colon V \to Y$ making the following diagram commute.



Note that $\iota_2 = \iota_1 \gg 0 = 0$. We write $\ker f \coloneqq \iota_1$ and $\operatorname{Ker} f \coloneqq V$. Note that this notation is dangerous: $\operatorname{Ker} f$ and $\ker f$ are only defined up to unique preferred isomorphism. Many authors will regularly abuse this fact and write $g = \ker f$ and $h = \ker f$ despite the fact that $g \neq h$.

In this thesis, we adopt a different convention. For us, $\ker f$ will always refer to a fixed morphism, which we also refer to as "the" kernel of f. If we have some other morphism g that has the universal property of the kernel of f, then we will say that g is "a" kernel of f.

2. The Lean interactive theorem prover

Mathematics can be formalized in a number of available systems which differ in aspects such as the underlying theory, the style in which proofs are written down, the amount of automation that is available, the amount of mathematics and computer science that have been formalized and comfort-of-life aspects such as available tooling and syntax.

In this thesis, we use version 3.8.0 of the community-maintained fork of the Lean theorem prover [19, 20]. Lean is a relatively new theorem prover developed by Leonardo de Moura at Microsoft Research.

The underlying theory of Lean is a dependent type theory called the calculus of inductive constructions[3] [20, 23] with proof irrelevance. Lean has native support for quotient types.

Lean makes heavy use of the so-called Curry-Howard isomorphism. Roughly speaking, we identify a mathematical statement with the collection of its proofs. In particular, a false statement is identified with the empty set and, in our proof-irrelevant theory, a true statement is identified with the one-element set. Under this identification, we should think of a proof of an implication $P \implies Q$ as a function that takes a proof of P and produces a proof of Q. For example, here is a (not very idiomatic) proof of the statement $P \wedge Q \implies Q \wedge P$ in Lean.

```
lemma and_comm {P Q : Prop} (proof_of_P_and_Q : P ∧ Q) : Q ∧ P :=
match proof_of_P_and_Q with
| ⟨proof_of_P, proof_of_Q⟩ := ⟨proof_of_Q, proof_of_P⟩
end
```

Here is how this code should be read: We declare a *function* called `and_comm`, which takes three arguments[4], the statements P and Q and a proof of $P \wedge Q$, and produces a proof of $Q \wedge P$. It does this by deconstructing the proof of $P \wedge Q$, which is really just a tuple of a proof of P and a proof of Q into its components and

[3] Notably, unlike in some other interactive theorem provers, the underlying theory is *not* some variant of Zermelo-Fraenkel set theory.

[4] Note that we use type theory, and type theorists use the symbol : for membership rather than $\in$.

rearranging them into a tuple of a proof of Q and a proof of P, that is, a proof of $Q \wedge P$.

In this way, Lean is really just a functional programming language with a rather powerful type system. One could even think about actually executing proofs, but we will not dwell on that idea. Instead, we will go in the opposite direction and assume a very powerful version of the axiom of choice.

```
axiom choice {α : Sort u} : nonempty α → α
```

`choice` takes a proof that a type is nonempty and produces an element of said type. This destroys all dreams of doing computation (our theory ceases to be constructive), but it allows us to make arguments such as making a case distinction over an arbitrary statement and other types of arguments which mathematicians will generally take for granted.

2.1. Interactive proofs. Constructing proofs by using Lean like a programming language can become rather tedious when proofs get larger, so Lean has a mode for constructing proofs semi-automatically, called *tactic mode*. A proof in tactic mode is a sequence of steps similar to how a mathematician might formulate a proof. When writing a tactic-mode proof, the current state of the proof is displayed to the user, so a user can write the proof step by step and use the feedback provided by Lean to figure out what to do next in order to progress in the proof. This mode is what makes Lean an *interactive* theorem prover.

As an example for such an interactive proof, consider the following problem from an early problem sheet of an introductory course on algebra:

Problem. Let M be a semigroup (that is, a set M with an associative binary operation $\cdot$). Assume that there is an element $e \in M$ satisfying the following conditions:

1. $\forall m \in M \colon e \cdot m = m$
2. $\forall m \in M\ \exists m' \in M \colon m' \cdot m = e$

Prove the following statements.

1. $\forall m \in M \colon m \cdot e = m$
2. $\forall m \in M\ \exists m' \in M \colon m \cdot m' = e = m' \cdot m$

The hypotheses of the example can be expressed in Lean in the following way (we suggest the reader ignores the different types of parentheses around the arguments for now):

```
variables {M : Type*} [semigroup M]
variables {e : M} (left_neutral : ∀ m, e * m = m)
variables {inv : M → M} (left_inv : ∀ m, (inv m) * m = e)
```

Specifically, we declare a semigroup M, and an element $e \in M$. Furthermore, we let `left_neutral` refer to a proof of $\forall m \in M \colon e \cdot m = m$. We also assert the existence of a function (of sets) `inv` from M to M, and a proof called `let_inv` of the statement $\forall m \in M, \mathrm{inv}\, m \cdot m = e$. We direct the reader's attention to the function application syntax used by Lean: To apply the argument `m` to the function `inv`, we write `inv m` instead of `inv(m)`. Similarly, if we had a function `f` to which we wish to apply two parameters `a` and `b`, then we would write `f a b`.

Let us first state and prove the second claim.

```
theorem right_inv : ∀ m, m * (inv m) = e :=
```

Proof. We enter an interactice proof by saying `begin`.

```
begin
```

If we move the cursor after the `begin`, Lean will notice that we are inside an interactive proof and show us the current state of the proof. In our case, it looks like this:

```
1 goal
M : Type u_1,
e : M,
left_neutral : ∀ (m : M), e * m = m,
inv : M → M,
left_inv : ∀ (m : M), inv m * m = e
⊢ ∀ (m : M), m * inv m = e
```

The state of the proof consists of a list of so-called *goals*. A goal consists of a list of hypothesis and a statement we have to prove using the hypotheses. Now, we can use tactics to manipulate one or more goals.

The first tactic we will use is called `intro`. It is the Lean equivalent of saying "Let m be..." in a mathematical proof. If we write

```
intro m,
```

then our state becomes

```
1 goal
M : Type u_1,
e : M,
left_neutral : ∀ (m : M), e * m = m,
inv : M → M,
left_inv : ∀ (m : M), inv m * m = e,
m : M
⊢ m * inv m = e
```

As we can see, the ∀ (m : M) has been eliminated. Instead, m is now part of the hypotheses, and it remains to show $m \cdot \text{inv}\, m = e$ for this specific m.

Now how do we prove the statement? Most tactics let us manipulate the goal, allowing us to work backwards from the goal. If we have managed to change the goal to something trivial like $a = a$, then we have finished the proof.

In many cases, it is also helpful to work forwards from the hypotheses. This can be done by manipulating hypotheses directly or by proving new hypotheses. In our case, it will be helpful to know that $m \cdot \text{inv}\, m = m \cdot \text{inv}\, m \cdot m \cdot \text{inv}\, m$. If we say

```
have double_m_inv : m * (inv m) = m * (inv m) * m * (inv m),
```

then the goal state will change to

```
2 goals
M : Type u_1,
e : M,
left_neutral : ∀ (m : M), e * m = m,
inv : M → M,
left_inv : ∀ (m : M), inv m * m = e,
m : M
⊢ m * inv m = m * inv m * m * inv m

M : Type u_1,
e : M,
left_neutral : ∀ (m : M), e * m = m,
inv : M → M,
left_inv : ∀ (m : M), inv m * m = e,
m : M,
double_m_inv : m * inv m = m * inv m * m * inv m
⊢ m * inv m = e
```

In other words, by saying `have name : type` we introduce a new goal where we have to prove `type`, and in our original goal we get a new hypothesis `name : type`. In this way, saying `have` is like saying "Let us first show that...".

For reasons of brevity, let us skip the proof of the first goal[5] and look at the proof of the second goal. Since we will be working backwards from the goal from now, we will only show the proof obligation part of the goal state, i.e., the line including $\vdash$.

The first thing we say is

```
rw ←left_inv (m * (inv m))
```

This tells lean to search for a place where it can apply the equality

$$\operatorname{inv}(m \cdot \operatorname{inv} m) \cdot m \cdot \operatorname{inv} m = e$$

in the backwards direction. Since there is only one occurrence of e in the goal, the goal becomes

```
⊢ m * inv m = inv (m * inv m) * (m * inv m)
```

Next, we want to use the identity `double_m_inv` that we proved earlier on the very right of the goal. Since the term $m \cdot \operatorname{inv} m$ occurs multiple times in the goal, we use some special syntax to apply it only in the correct position.

```
conv { for (m * inv m) [3] { rw double_m_inv } }
```

This, plus some reassociating that we omit here, changes the goal into

```
m * inv m = inv (m * inv m) * (m * inv m) * m * inv m
```

Now we are almost done. We tell Lean to apply the left inverse property

```
rw left_inv,
```

and are left with the goal

```
⊢ m * inv m = e * m * inv m
```

But now all that remains is to use that e is left neutral:

```
rw left_neutral,
```

After this, the goal would become

```
⊢ m * inv m = m * inv m
```

but Lean notices that this is tautological, so it closes the goal and says

```
goals accomplished
```

and we can close the proof by saying

```
end
```

For reference, here is the full proof:

```
begin
  intro m,
  have double_m_inv : m * (inv m) = m * (inv m) * m * (inv m),
  { conv_lhs { rw [←left_neutral (inv m), ←left_inv m,
     ←mul_assoc, ←mul_assoc], }, },
  rw ←left_inv (m * (inv m)),
  conv { for (m * inv m) [3] { rw double_m_inv } },
  simp only [←mul_assoc],
  conv_rhs { congr, congr, rw mul_assoc, },
  rw left_inv,
  rw left_neutral,
end  □
```

[5]Here is the pen-and-paper proof: $m \cdot \operatorname{inv} m = m \cdot (e \cdot \operatorname{inv} m) = m \cdot \operatorname{inv} m \cdot m \cdot \operatorname{inv} m$.

The first claim of the exercise is an easy corollary of the second claim, and we encourage the reader to try it out themselves.

The examples given here barely scratch the surface of Lean syntax. For a more complete introduction, we refer to the official introduction to Lean [2].

3. The Lean mathematical library

What sets Lean apart from many other theorem provers is that it has been used to formalize modern mathematics[6]. Indeed, the last few years have seen formalizations in Lean of Hensel's Lemma [15], schemes [11], and even the definition of a perfectoid space [7]. These results are of significance because they demonstrate that interactive theorem provers are capable of coping with the complex definitions appearing in modern mathematics. This is orthogonal to other major breakthroughs in interactive theorem provers like the formal proof of the Feit-Thompson theorem [13], which deal with long proofs about very simple objects rather than very complex definitions.

There is a community-maintained repository of formalized mathematical theories called mathlib [8]. It contains a large number of sections, including basic mathematical objects, analysis, algebra, measure theory, theoretical computer science and much more. Of particular interest to us are the sections containing linear algebra and category theory.

3.1. Limits. We will give a quick overview over a section of the Lean category theory that we will make heavy use of: the limits library.

A cone over a diagram $F\colon \mathsf{J} \to \mathsf{C}$ is defined as a natural transformation from a constant functor to F. We shall not be concerned with what this means precisely, except for the fact that it is equivalent to the definition we gave above while allowing to reuse some existing machinery.

```
structure cone (F : J ⥤ C) :=
(X : C)
(π : (const J).obj X ⟶ F)
```

If `s : cone F`, then we can access the vertex of `s` by saying `s.X`. The morphisms s_J appearing in our definition of a cone are accessed using `s.π.app J`.

The definition of a limit cone coincides with our definition:

```
structure is_limit (t : cone F) :=
(lift  : Π (s : cone F), s.X ⟶ t.X)
(fac  : ∀ (s : cone F) (j : J), lift s ≫ t.π.app j = s.π.app j)
(uniq : ∀ (s : cone F) (m : s.X ⟶ t.X) (w : ∀ j : J, m ≫ t.π.app j = s.π.app j),
  m = lift s)
```

Note that there are two common use cases for limits: Sometimes we want to assert that some specific cone is a limit cone, and sometimes we simply need access to some limit cone for a certain diagram (i.e., we only care about the existence of a limit cone). The first use case is covered by the previous definition. For the second use case, there is a class that asserts that some limit exists:

```
class has_limit (F : J ⥤ C) :=
(cone : cone F)
(is_limit : is_limit cone)
```

[6]Of course, "modern mathematics" is an ill-defined notion and there is an ongoing debate about which terms to use for "mainstream mathematicians" and the work they do. For us, "modern mathematics" will mean things like algebraic geometry, algebraic number theory, topology, category theory, functional analysis, measure theory and so on, but not things like higher topos theory, mathematical logic or theoretical computer science.

If we write `limit F`, then type class resolution will search for a limit cone c of F and return its vertex. If we write `limit.π F J`, then type class resolution will return c_J for the same c.

In addition, the limits library defines the various shapes of diagrams mentioned above. Most importantly for us, `parallel_pair f g` is the diagram

$$X \underset{g}{\overset{f}{\rightrightarrows}} Y,$$

and `kernel f 0` means `limit (parallel_pair f 0)`, or, using the notation introduced above, $\operatorname{Ker} f$. Similarly, `kernel.ι f` is the same as $\ker f$. Dually, `cokernel f` is $\operatorname{Coker} f$ and `cokernel.π f` is $\operatorname{coker} f$. In this way, Lean adopts the same convention as us: `kernel.ι f` refers to a specific morphism. If we want to express that g is a kernel of f, then we have to say `is_limit (kernel_fork.of_ι g _)`.

3.2. A formal proof of the four lemma. As promised, we will give a formal proof of the four lemma for modules. Recall that the four lemma discusses the following commutative diagram of R-modules with exact rows:

$$\begin{array}{ccccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C & \xrightarrow{h} & D \\ \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\beta} & & \downarrow{\scriptstyle\gamma} & & \downarrow{\scriptstyle\delta} \\ A' & \xrightarrow{f'} & B' & \xrightarrow{g'} & C' & \xrightarrow{h'} & D' \end{array}$$

Let us first formalize the data contained in the commutative diagram.

```
variables {R : Type*} [ring R]
variables {A : Type*} {B : Type*} {C : Type*} {D : Type*}
variables [add_comm_group A] [add_comm_group B]
variables [add_comm_group C] [add_comm_group D]
variables [module R A] [module R B] [module R C] [module R D]
variables {A' : Type*} {B' : Type*} {C' : Type*} {D' : Type*}
variables [add_comm_group A'] [add_comm_group B']
variables [add_comm_group C'] [add_comm_group D']
variables [module R A'] [module R B'] [module R C'] [module R D']
variables {f : A →ₗ[R] B} {g : B →ₗ[R] C} {h : C →ₗ[R] D}
variables {f' : A' →ₗ[R] B'} {g' : B' →ₗ[R] C'} {h' : C' →ₗ[R] D'}
variables {α : A →ₗ[R] A'} {β : B →ₗ[R] B'} {γ : C →ₗ[R] C'} {δ : D →ₗ[R] D'}
variables (fg : f.range = g.ker) (gh : g.range = h.ker)
variables (fg' : f'.range = g'.ker) (gh' : g'.range = h'.ker)
variables (comm₁ : f' ∘ α = β ∘ f) (comm₂ : g' ∘ β = γ ∘ g) (comm₃ : h' ∘ γ = δ ∘ h)
```

As we can see, to introduce a module A, we have to include three statements: `A : Type*`, which should be thought of as the underlying set of A, `add_comm_group A`, which asserts that A has been equipped with an abelian group structure, and `module R A`, which asserts that A has been equipped with an R-module structure. The square brackets indicate that an argument should be determined using type class inference, effectively telling Lean to automatically search for the structure. As an example, if we want to apply a lemma about abelian groups, this lets us apply this lemma with argument $\mathbb{Z}$ (the set[7]!), and Lean will automatically figure out that we meant $\mathbb{Z}$ together with its usual abelian group structure. This saves a lot of typing, at the expense of performance and potential headaches if we (on purpose or, even worse, by accident) defined multiple abelian group structures on the same set[8].

Now we can give a formal statement and proof of the four lemma.

[7] actually, the type

[8] $\mathbb{Z}$ is actually one of the worst offenders in this regard. $\mathbb{Z}$ is a $\mathbb{Z}$-module, since every ring is a module over itself, but $\mathbb{Z}$ is also a $\mathbb{Z}$-module, since it is an abelian group, and every abelian group

LEMMA (Four lemma, version 1). If α is surjective and β and δ are injective, then γ is injective.

```
lemma four (hα : α.range = ⊤) (hβ : β.ker = ⊥) (hδ : δ.ker = ⊥) : γ.ker = ⊥ :=
```

PROOF. Let $c \in C$ such that $\gamma(c) = 0$. We have to show that $c = 0$.

```
ker_eq_bot'.2 $ assume (c : C) (hc : γ c = 0), show c = 0, from
```

Let us first observe that c is in the kernel of h.

```
 have c ∈ ker h, from
```

Since δ is injective, it suffices to show that $\delta(h(c)) = 0$,

```
  suffices (δ ∘ h) c = 0, from mem_ker.2 $ ker_eq_bot'.1 hδ (h c) this,
```

which is true by commutativity.

```
  calc δ (h c) = h' (γ c) : congr_fun comm₃.symm c
       ... = h' 0     : congr_arg h' hc
       ... = 0        : map_zero h',
```

By exactness, this gives us a preimage b of c under g.

```
 exists.elim (gh.symm ▸ this : c ∈ range g) $ assume (b : B) ⟨_, (hb : g b = c)⟩,
```

Now we claim that $g'(\beta(b)) = 0$,

```
  have β b ∈ ker g', from mem_ker.2 $
```

which is true by commutativity.

```
   calc g' (β b) = γ (g b) : congr_fun comm₂ b
        ... = γ c     : congr_arg γ hb
        ... = 0       : hc,
```

Again by exactness, we have a preimage a' of $\beta(b)$ under f'...

```
  exists.elim (fg'.symm ▸ this : β b ∈ range f') $ assume (a' : A') ⟨_, (ha' : f' a' = β
    b)⟩,
```

...which in turn, by surjectivity of α, has a preimage a.

```
   exists.elim (range_eq_top.1 hα a') $ assume (a : A) (ha : α a = a'),
```

We claim that $f(a) = b$.

```
   have f a = b, from
```

By injectivity of β, we might as well check that $\beta(f(a)) = \beta(b)$,

```
     suffices β (f a) = β b, from ker_eq_bot.1 hβ this,
```

which is true by commutativity.

```
     calc β (f a) = f' (α a) : congr_fun comm₁.symm a
          ... = f' a'    : congr_arg f' ha
          ... = β b      : ha',
```

But now we are done, since going two steps in an exact sequence is zero.

```
   calc c = g b     : hb.symm
     ... = g (f a) : congr_arg g this.symm
     ... = 0       : mem_ker.1 $ fg ▸ submodule.mem_map_of_mem trivial   □
```

---

is a $\mathbb{Z}$-module, as we have shown in Section 1.1. Of course, these two $\mathbb{Z}$-module structures are equal, but they are not definitionally equal, which is enough to cause problems.

CHAPTER 2

Pseudoelements in abelian categories

The formal proof of the four lemma was carried out in the category of R-modules for some ring R. It is possible to state and prove this result in the much more general context of abelian categories. A formalization of this development will be presented in this chapter.

Most of the mathematics in this chapter, in particular the proofs we have formalized, can be found in volume 2 of Borceux [5, Chapter 1]. An exception is Section 2.4, where the "technical result" was contributed by us, and the proof of the main result (which is omitted) is taken from Aluffi [1, Lemma IX.2.3]. In addition, the mathematical content of Sections 3 and 4 is original (but not very deep).

The formalization work described in this chapter is original, unless otherwise indicated.

In the entire chapter, let `C` be a category.

```
variables {C : Type u} [C : category.{v} C]
```

1. Preadditive categories and biproducts

1.1. Preadditive categories. A *preadditive category* `C` is a category such that for every two objects $X, Y \in$ `C` the set $\mathrm{Hom}(X, Y)$ has the structure of an abelian group. Furthermore, we require the composition map

$$\gg\colon \mathrm{Hom}(X, Y) \times \mathrm{Hom}(Y, Z) \to \mathrm{Hom}(X, Z)$$

to be linear in both variables (cf. [5, Def. 1.2.1]).

This is the "pedestrian" version of the definition. It is also possible to define preadditive categories as categories enriched over the category of abelian groups (cf. [5, Def. 6.2.1]). Developing enriched categories instead of preadditive categories is much more general, but it has significant overhead compared to the direct definition, with no real benefit if the only aim is to define abelian categories. Morrison and Buzzard are currently developing monoidal and enriched categories for the Lean mathematical library[1], and we expect that their work will work as a drop-in replacement for the naive implementation presented here once it is ready.

For now, here is the direct definition:

```
class preadditive :=
(hom_group : Π P Q : C, add_comm_group (P ⟶ Q))
(distrib_left' : Π P Q R : C, ∀ (f f' : P ⟶ Q) (g : Q ⟶ R),
  (f + f') ≫ g = f ≫ g + f' ≫ g . obviously)
(distrib_right' : Π P Q R : C, ∀ (f : P ⟶ Q) (g g' : Q ⟶ R),
  f ≫ (g + g') = f ≫ g + f ≫ g' . obviously)
```

Note the `obviously` in the declaration. This is a so-called `auto_param`. When creating an instance of `preadditive`, it is possible to leave out the proof of a field marked with an `auto_param`. In this case, Lean will automatically attempt to apply the tactic `obviously` to the goal given by the omitted parameter. If the tactic

[1] see, for example, the discussion at `https://leanprover-community.github.io/archive/116395maths/72916longexactsequenceofcohomology.html`.

successfully dispatches the goal, the structure is created with the generated proof. Otherwise, an error is generated. The tactic `obviously` is currently a synonym for `tidy`. The tactic `tidy` attempts to execute a variety of conservative tactics—such as invoking the simplifier or eliminating universal quantifiers in the goal—in sequence until it makes no further progress [8]. In practice, it is particularly good at proving trivial facts such as easy instances of naturality or proving facts involving finite categories by breaking up a goal into all its cases and then dispatching all cases using `simp` or `refl`.

In this particular case, `obviously` is useful for deriving the distributive properties for categories that are built out of other categories. For example, if C is a preadditive category, then the category of chain complexes over C is again preadditive, and `obviously` will be able to prove the distributive properties. However, for concrete categories, the automation will likely struggle to dispatch the goal, since the goal will not be in the correct form for the relevant `simp` lemmas to fire.

Another important consideration is runtime. It is sometimes undesirable to rely on this sort of heavy automation even if it works, since it can take many seconds to succeed, where a manual proof is elaborated in a few milliseconds. This can quickly add up in larger files, so in some cases in the category theory library proofs are carried out manually purely for performance reasons.

Since an `auto_param` is part of the type of the structure field, using the projections that are automatically generated by Lean is undesirable, as there is an extra step involved to strip the auto param declaration to recover the actual type of the field. For this reason, mathlib defines a command called `restate_axiom` which generates lemmas which are identical to the automatically generated projections, but with the `auto_param` stripped off.

```
restate_axiom preadditive.distrib_left'
restate_axiom preadditive.distrib_right'
```

Now that we have defined preadditive categories, we are interested in basic properties such as

$$(f - f') \gg g = f \gg g - f' \gg g. \tag{1}$$

Of course, we could prove these properties by hand, but it is much more efficient to utilize the fact that by fixing one argument of composition and varying the other we obtain homomorphisms of abelian groups:

```
def hom_right {P Q : C} (R : C) (f : P ⟶ Q) : (Q ⟶ R) →+ (P ⟶ R) :=
mk' (λ g, f ≫ g) $ preadditive.distrib_right C P Q R f

def hom_left (P : C) {Q R : C} (g : Q ⟶ R) : (P ⟶ Q) →+ (P ⟶ R) :=
mk' (λ f, f ≫ g) $ λ f f', preadditive.distrib_left C P Q R f f' g
```

Now we can express the desired properties using these homomorphisms. For example, the proof of property (1) is now significantly shorter than its statement:

```
@[simp] lemma sub_distrib_left {P Q R : C} (f f' : P ⟶ Q) (g : Q ⟶ R) :
  (f - f') ≫ g = f ≫ g - f' ≫ g :=
map_sub (hom_left _ _) _ _
```

This is because we have access to `map_sub`, which states that if $\varphi\colon G \to H$ is a homomorphism of abelian groups, then for all $g, g' \in G$ it is $\varphi(g - g') = \varphi(g) - \varphi(g')$.

Finally, we note that a preadditive category with kernels already has all equalizers. Mathematically, this is obvious: Of course the kernel of $f - g$ is an equalizer of f and g. The formal proof of this directly translates the universal property of the equalizer of $f - g$ and 0 into the universal property of the equalizer of f and g.

1.2. Biproducts. Let C be a preadditive category and $X, Y \in \mathsf{C}$. A *biproduct* of X and Y is an object P together with morphisms $\pi_1 \colon P \to X$, $\pi_2 \colon P \to Y$, $\iota_1 \colon X \to P$, $\iota_2 \colon Y \to P$ such that

- $\iota_1 \gg \pi_1 = \mathbf{1}_X$,
- $\iota_2 \gg \pi_2 = \mathbf{1}_Y$,
- $\iota_1 \gg \pi_2 = 0$,
- $\iota_2 \gg \pi_1 = 0$ and
- $\pi_1 \gg \iota_1 + \pi_2 \gg \iota_2 = \mathbf{1}_P$.

To make this definition smooth to use in Lean, we follow a two-step approach in the formalization. First, we define a structure that captures the fact that an object is a biproduct of some given objects.

```
class is_biproduct (X Y P : C) :=
(fst : P ⟶ X)
(snd : P ⟶ Y)
(inl : X ⟶ P)
(inr : Y ⟶ P)
(inl_fst' : inl ≫ fst = 𝟙 X . obviously)
(inr_snd' : inr ≫ snd = 𝟙 Y . obviously)
(inr_fst' : inr ≫ fst = 0 . obviously)
(inl_snd' : inl ≫ snd = 0 . obviously)
(total' : fst ≫ inl + snd ≫ inr = 𝟙 P . obviously)
```

After some omitted invocations of `restate_axiom` and `attribute` [`simp`], we can then define a class that expresses the fact that a biproduct of two objects exists.

```
class has_biproduct (X Y : C) :=
(P : C)
(is_biproduct : is_biproduct.{v} X Y P)

attribute [instance] has_biproduct.is_biproduct
```

Using this definition, code can require that the biproduct exists by assuming an instance of `has_biproduct X Y` and then access the biproduct using `has_biproduct.P X Y`. Since this is a bit verbose and this formulation has a tendency to run into universe resolution failures, we provide shorthands:

```
variables (X Y)
abbreviation biproduct := has_biproduct.P.{v} X Y

notation X ` ⊞ `:20 Y:20 := biproduct X Y
```

Unfortunately, the symbol $\oplus$, which is the usual notation for biproducts, is already in use by the Lean mathematical library, so we resort to using $\boxplus$ instead.

We verify that the biproduct has the universal properties of both the product and coproduct. In particular, this gives us access to a very important property of the biproduct: Two morphisms $f, g \colon T \to X \boxplus Y$ are equal if and only if $f \gg \pi_i = g \gg \pi_i$ for $i = 1, 2$.

```
lemma biproduct.ext_lift {T : C} (f g : T ⟶ X ⊞ Y)
  (h₁ : f ≫ biproduct.fst = g ≫ biproduct.fst)
  (h₂ : f ≫ biproduct.snd = g ≫ biproduct.snd) : f = g :=
is_limit.hom_ext (biproduct.cone_is_limit X Y) $
  λ j, by { cases j, exact h₁, exact h₂ }
```

We also prove the (equally important) dual statement.

Finally, an important question is how to construct biproducts. The answer is simple: In a preadditive category, a product (and a coproduct, for that matter, but we do not need that) is already a biproduct:

```
def biproduct.of_prod (X Y : C) [has_limit.{v} (pair X Y)] : has_biproduct.{v} X Y :=
{ P := X × Y,
  is_biproduct :=
  { fst := prod.fst,
    snd := prod.snd,
    inl := prod.lift (𝟙 X) 0,
   inr := prod.lift 0 (𝟙 Y) } }
```

Here we see a nice example of the automation Lean provides: All of the five properties that a biproduct must satisfy are checked by Lean automatically. Thanks to `obviously`, we do not even have to mention them in the instantiation.

2. Abelian categories

2.1. The definition. A preadditive category C is called abelian if it has a zero object, binary products and coproducts, all kernels and cokernels and if every monomorphism is the kernel of some morphism and every epimorphism is the cokernel of some morphism.

```
class abelian extends preadditive.{v} C :=
(has_zero_object : has_zero_object.{v} C)
(has_binary_products : has_binary_products.{v} C)
(has_binary_coproducts : has_binary_coproducts.{v} C)
(has_kernels : has_kernels.{v} C)
(has_cokernels : has_cokernels.{v} C)
(mono_is_kernel : Π {X Y : C} (f : X ⟶ Y) [mono f], normal_mono.{v} f)
(epi_is_cokernel : Π {X Y : C} (f : X ⟶ Y) [epi f], normal_epi.{v} f)
```

There are numerous equivalent definitions of abelian category. We chose to use the present definition because it is reasonably easy to verify for concrete categories while still allowing quick proofs of the statements we are interested in. In particular, it must be noted that, if we remove "preadditive" from the definition of an abelian category, nothing changes: We can construct a preadditive structure on C purely from the limit-colimit structure dictated by the other properties, and it is even possible to show that this preadditive structure is unique up to isomorphism[2]. This result, while amazing, is essentially irrelevant in practice, because all interesting examples of abelian categories come with a natural preadditive structure.

2.2. Image factorization. One of the defining features of abelian categories is the existence of image factorizations. In its full form, the theorem about image factorizations in abelian categories states that every morphism $f\colon P \to Q$ factors as

$$P \xrightarrow{p} \twoheadrightarrow \operatorname{Coker}\ker f \xrightarrow{h} \operatorname{Ker}\operatorname{coker} f \xhookrightarrow{i} Q,$$

where p is the canonical morphism into the cokernel, which is an epimorphism, h is a unique isomorphism and i is the canonical morphism from the kernel, which is a monomorphism. The proof of this statement consists of several steps (cf. [5, Proposition 1.5.5]):

1. Using the universal property of the cokernel, show that f factors as
$$P \xrightarrow{p} \twoheadrightarrow \operatorname{Coker}\ker f \xrightarrow{i'} Q.$$
2. Show that i' is a monomorphism.
3. Using the universal property of the kernel, show that f factors as
$$P \xrightarrow{p'} \operatorname{Ker}\operatorname{coker} f \xhookrightarrow{i} Q.$$

[2]This statement must be made more precise to make sense, but we will not go into that here. We refer the interested reader to Proposition 1.2.7 and Section 1.6 of Borceux [5].

4. Show that p' is an epimorphism.
5. Use a general theorem about *strong epi-mono factorizations* to derive the existence of a unique isomorphism h making the diagram

$$\begin{array}{ccccc} P & \xrightarrow{p} & \operatorname{Coker}\ker f & \xrightarrow{i'} & Q \\ \downarrow{\scriptstyle \mathbf{1}_P} & & \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle \mathbf{1}_Q} \\ P & \xrightarrow{p'} & \operatorname{Ker}\operatorname{coker} f & \xrightarrow{i} & Q \end{array}$$

commute.

We have formalized the entire proof, including the definitions and relevant theorems about strong epimorphisms and strong epi-mono factorizations[3]. Steps one and two are very similar to steps three and four. We will present the formal proof of steps three and four here. Let $f\colon P \to Q$. The morphism $i\colon \operatorname{Ker}\operatorname{coker} f \to Q$ is just the canonical map of the kernel of the cokernel. In Lean, it is called `kernel.ι (cokernel.π f)`. The map $p\colon P \to \operatorname{Ker}\operatorname{coker} f$ now comes from the universal property of the kernel. For a morphism u to factor through the kernel of v, we must show that $u \gg v = 0$. But in our case this amounts to showing that $f \gg \operatorname{coker} f = 0$, which is true by definition of the cokernel, and so we get our morphism p.

```
def factor_thru_image : P ⟶ kernel (cokernel.π f) :=
kernel.lift (cokernel.π f) f $ cokernel.condition f
```

The fact that $p \gg i = f$ is now true by construction:

```
lemma image.fac : factor_thru_image f ≫ kernel.ι (cokernel.π f) = f :=
by erw limit.lift_π; refl
```

We are already done with the third step. It remains to show that p (that is, `factor_thru_image f`) is an epimorphism.

```
instance : epi (factor_thru_image f) :=
```

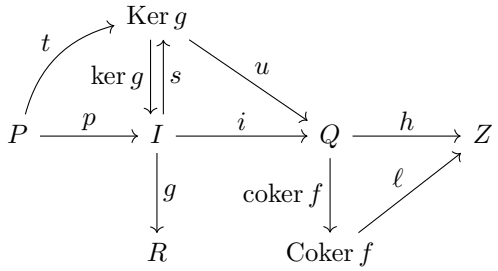
Proof. Let us first introduce the shorthands we have already been using to Lean to improve readability.

```
let I := kernel (cokernel.π f), p := factor_thru_image f,
  i := kernel.ι (cokernel.π f) in
```

There is an easy theorem about preadditive categories that states that p is an epimorphism if and only if for every composable g satisfying $p \gg g = 0$ we have $g = 0$. We will use this characterization in our proof.

```
(cancel_zero_iff_epi _).2 $ λ R (g : I ⟶ R) (hpg : p ≫ g = 0),
begin
```

We refer to the following diagram.



[3]see `https://github.com/leanprover-community/mathlib/commit/0567b7faba5783ac4c272c3b6265a4513f69c47d`.

We define $u \coloneqq \ker g \gg i$. Since i and $\ker g$ are both monomorphisms, so is u. Since C is abelian, u is the kernel of some morphism h.

```
let u := kernel.ι g ≫ i,
haveI : mono u := mono_comp _ _
have hu := abelian.mono_is_kernel u,
let h := hu.g,
```

Since $p \gg g = 0$, p factors through $\ker g$ via some morphism $t\colon P \to \operatorname{Ker} g$.

```
obtain ⟨t, ht⟩ := kernel.lift' g p hpg,
```

Now we can calculate that $f \gg h = 0$.

```
have fh : f ≫ h = 0, calc
  f ≫ h = (p ≫ i) ≫ h : (image.fac f).symm ▹ rfl
    ... = ((t ≫ kernel.ι g) ≫ i) ≫ h : ht ▹ rfl
    ... = t ≫ u ≫ h : by simp only [category.assoc]; conv_lhs { congr, skip, rw ← 
    category.assoc }
    ... = t ≫ 0 : hu.w ▹ rfl
    ... = 0 : has_zero_morphisms.comp_zero _ _,
```

But this means that h factors through $\operatorname{coker} f$ via some $\ell\colon \operatorname{Coker} f \to Z$,

```
obtain ⟨l, hl⟩ := cokernel.desc' f h fh,
```

and we can use this to show that $i \gg h = 0$.

```
have hih : i ≫ h = 0, calc
  i ≫ h = i ≫ cokernel.π f ≫ l : hl ▹ rfl
    ... = 0 ≫ l : by rw [←category.assoc, kernel.condition]
    ... = 0 : has_zero_morphisms.zero_comp _ _ _,
```

We recall that u was the kernel of h. This means that i factors through u via some morphism $s\colon I \to \operatorname{Ker} g$.

```
obtain ⟨s, hs⟩ := normal_mono.lift' u i hih,
```

But then $s \gg \ker g \gg i = s \gg u = i$, but since i is a monomorphism, $s \gg \ker g = \mathbf{1}_I$. In particular, this means that $\ker g$ is an epimorphism.

```
have hs' : (s ≫ kernel.ι g) ≫ i = 𝟙 I ≫ i,
  by rw [category.assoc, hs, category.id_comp],
have : epi (kernel.ι g) := epi_of_epi_fac ((cancel_mono _).1 hs'),
```

But then $\ker g \gg g = 0 = \ker g \gg 0$, and since $\ker g$ is an epimorphism, $g = 0$ as required.

```
  exact (cancel_zero_iff_epi _).1 this _ _ (kernel.condition g)
end  □
```

This was an example of a typical proof using abelian categories: Keep invoking universal properties of kernels and cokernels until the result becomes apparent.

2.3. An epimorphism is the cokernel of its kernel. There is a theorem that mathematicians like to succinctly state as follows: "In an abelian category, an epimorphism is the cokernel of its kernel," see for example Proposition 1.5.7 in Borceux [5] or Lemma IX.1.8 in Aluffi [1]. One might be tempted to attempt to formalize this statement in the following way.

```
lemma epi_is_cokernel_of_kernel [epi f] : f = cokernel.π (kernel.ι f)
```

The problem with this statement is that it is not always true. The reason is that limits are only unique up to isomorphism, and so we may have to compose the right hand side of the equation with an isomorphism for the lemma to become correct.

This is not an inaccuracy in the mathematical formulation. Limits are not only unique up to isomorphism, but they are unique up to unique cone isomorphism, or,

put differently, unique up to a unique preferred isomorphism that has certain nice properties, as laid out in Section 1.4.2 of the first chapter. So when a mathematician talks about *the* kernel, there is no confusion about which kernel they mean: It is always possible to translate statements about one kernel into statements about another kernel using the preferred isomorphism.

However, to formalize a statement like this, we have to be a bit more precise. An immediate fix would be to assert that there is some isomorphism i that can be composed with $\operatorname{coker} \ker f$ so that the statement becomes true. This would not be a useful way to state the lemma for two reasons.

First, what we care about is usually not the precise morphism, but its universal property: If $\ker f \gg g = 0$ for some g, we want g to uniquely factor through f. The statement of this lemma should make this property easy to use.

Second, there is another unique-up-to-isomorphism problem hiding in the statement: The morphism $\ker f$ is *also* only unique up to preferred isomorphism! So maybe we do not care about a situation where $\ker f \gg g = 0$, but $h \gg g = 0$, where h is some morphism with the universal property of the kernel of f. In this way, the lemma as originally stated was actually too weak.

Putting these two factors together, we arrive at the following way to state the lemma.

```
def epi_is_cokernel_of_kernel [epi f] (s : fork f 0) (h : is_limit s) :
  is_colimit (cokernel_cofork.of_π f (kernel_fork.condition s))
```

The input is any morphism `fork.ι s` satisfying the universal property of the kernel of f, and the output is the statement "f has the universal property of the cokernel of `fork.ι s`."

2.4. The pullback of an epimorphism. Consider the diagram

$$\begin{array}{ccc} P & \xrightarrow{j} & X \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle f} \\ Y & \xrightarrow{g} & Z, \end{array}$$

where (P, i, j) are obtained by taking the pullback of f and g. If f is a monomorphism, then i is also a monomorphism. This is true without any assumptions on the category C (except, of course, that the pullback of f and g exists). There is a simple proof using the existensionality property of limits that we have contributed to the Lean category theory library[4].

If f is instead an epimorphism, then i will not be an epimorphism in general, but the result becomes true again if we require C to be abelian. This fact will be central to our definition of pseudoelements in Section 5.

Before we can state the proof, we require a slightly technical result about the diagram in Figure 1, where $p \coloneqq j \gg \iota_1 + i \gg \iota_2$ and $d \coloneqq \pi_1 \gg f - \pi_2 \gg g$. In the proof that i is an epimorphism if f is we will need to know that d is a cokernel of p. To show that, we use the fact that an epimorphism is the cokernel of its kernel (if f is an epimorphism, then so is d), so it suffices to show that p is a kernel of d.

Using the definitions of p and d as well as the properties of biproducts and the fact that the diagram is a pullback it is straightforward to verify that $p \gg d = 0$.

```
def pullback_to_biproduct_fork : fork (biproduct.desc f (-g)) 0 :=
kernel_fork.of_ι (pullback_to_biproduct f g) $
  by rw [biproduct.lift_desc, neg_right, pullback.condition, add_right_neg]
```

[4]see `https://github.com/leanprover-community/mathlib/commit/e8ad2e385d1546535a3badea6dedaf7b85c116a9`.

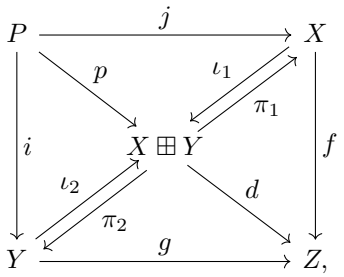


FIGURE 1. Commutative diagram illustrating the interaction between pullbacks and biproducts.

Next, we have to prove that every morphism $t : T \to X \boxplus Y$ satisfying $t \gg d = 0$ factors uniquely through P. The proof is standard: We translate the universal property of being a kernel of d into the universal property of being a pullback of f and g, which allows us to construct the lift. To show uniqueness, we use the existensionality lemma for pullbacks.

When formalizing this proof, an interesting issue arises. From the definition of an abelian category, the only limits we know exist are the initial object, kernels and binary products. We also have the dual colimits: A terminal object, cokernels and binary coproducts. We have already seen that this implies that an abelian category also has all equalizers and coequalizers. It can be shown that a category with equalizers, binary products and a terminal object has all finite limits. However, this fact is not fully available in mathlib. The missing piece is constructing finite products from binary products and a terminal objects. The idea of the construction is very simple, but implementing it in Lean is more intricate than it looks. The result has been formalized in Lean by Nawrocki as part of the Topos project [18], but during development of the theory of abelian categories we decided against using this result. Instead, we directly constructed pullbacks from equalizers and binary products using a very simple method and contributed the construction to mathlib[5].

When we switched to using this construction, the proof that p is a kernel of d ceased to compile. The reason was due to implementation details of the construction leaking into the proof. Lean provides the `ext` tactic that automatically searches for applicable existensionality lemmas and applies them as long as it finds more applicable lemmas. When the complicated construction of pullbacks in abelian categories is used, then Lean correctly finds and applies the existensionality lemma for pullbacks when calling `ext`. However, when the direct construction, which constructs a pullback as an equalizer of a product, is used, then Lean notices[6] that it can apply the existensionality lemma for equalizers. After applying that, it even notices that it can apply the existensionality lemma for products. But after applying these lemmas, the existensionality lemma for pullbacks is not applicable any more, and the rest of the proof fails.

There are two ways to fix this: Instead of using `ext`, we could instead `apply pullback.ext` directly. The other option, and the one we elected to use, is to tell Lean

[5]see `https://github.com/leanprover-community/mathlib/commit/c240fcb38cfdbe15b790f8f62dae391e194277bd`.

[6]The fact that Lean is able to look into the construction of the pullback is rather remarkable. Starting from our instantiation of `has_pullbacks`, we construct the pullback of a diagram that is naturally isomorphic to the diagram we are interested in, and transport the limit property over to the correct diagram via this natural isomorphism. This makes implementation easier, but adds multiple layers of indirection for Lean to sift through. Lean, however, seems to have no problem with that.

not to unfold the construction of the pullback by locally marking the corresponding definition as irreducible:

```
local attribute [irreducible]
    has_limit_cospan_of_has_limit_pair_of_has_limit_parallel_pair
```

From an engineering perspective, there is a strong argument to be made for constructions like this to be irreducible by default so that there is no way for these implementation details to leak (or, even worse, to actually be used in a proof). However, the trend in the Lean mathematical library is rather the opposite: Mathematicians love to rely on implementation details, and formal proofs that attempt to be independent of implementation details can be significantly harder to write, so a great deal of effort is spent in making sure constructions have nice and usable definitional properties. From this perspective, our requirements in this proof are the exception instead of the rule.

Finally, we can state our desired property of pullbacks.

```
instance epi_pullback_of_epi_f [epi f] : epi (pullback.snd : pullback f g ⟶ Y)
```

The proof is similar in style to the portion of the proof of image factorization we presented in detail, so we will omit it here, except for mentioning that at this point we need the full power of the statement that epimorphisms are cokernels of their kernels as discussed in Section 2.3.

3. Exact sequences

As we have already seen, exactness plays a major role in homological algebra, so we are naturally interested in defining it in the general context of abelian categories.

Recall that in the category of R-modules, we called the sequence

$$P \xrightarrow{f} Q \xrightarrow{g} R$$

of linear maps exact if the kernel of g coincides with the image of f. This definition cannot be directly transferred to general abelian categories, again due to the fact that kernels are not unique. There are several ways to fix this problem. Borceux [5] defines exactness of f and g to mean that $\ker \operatorname{coker} f$ has the universal property of $\ker g$. Another way is to assert that the canonical map $\operatorname{Ker} \operatorname{coker} f \to \operatorname{Ker} g$ is an isomorphism. This slightly stronger sounding statement is in fact equivalent to the one before. We have chosen to adopt yet another equivalent definition, used by Aluffi [1]:

```
def exact {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) : Prop :=
f ≫ g = 0 ∧ kernel.ι g ≫ cokernel.π f = 0
```

We chose this definition because it is very easy to write down and because it is very easy to verify both in abstract situations, but importantly also for concrete abelian categories. At the same time, it is easy to show that it is equivalent to the definitions mentioned before. Note however that we can freely move between equivalent definitions, so the exact definition does not matter[7].

We prove that our definition implies the first definition above. For this, we first have to show that $\ker \operatorname{coker} f \gg g = 0$, which is true because the map $P \to \operatorname{Ker} \operatorname{coker} g$ is an epimorphism as shown in Section 2.1.

```
def exact_fork {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) (e : exact f g) :
  kernel_fork g :=
kernel_fork.of_ι (kernel.ι (cokernel.π f)) $
  (preadditive.cancel_zero_iff_epi (factor_thru_image f)).1
    (by apply_instance) _ (kernel.ι (cokernel.π f) ≫ g) $
```

[7] That is, as long as we assume classical logic. Otherwise, it will make a difference whether our definition is a `Prop`, like the definition we chose, or carries data like the other definitions.

```
  by rw [←category.assoc, image.fac f, e.1]
```

Then we can verify that $\ker \operatorname{coker} f$ has the universal property of $\ker g$.

```
def exact_ker {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) (e : exact f g) :
 is_limit $ exact_fork f g e
```

We also give the well-known characterization of mono and epi in terms of exact sequences: A morphism f is a monomorphism if and only if $(0, f)$ is an exact sequence, where 0 is any zero morphism.

```
lemma exact_zero_of_mono (P : C) {Q R : C} (f : Q ⟶ R) [mono f] :
 exact (0 : P ⟶ Q) f

lemma mono_of_exact_zero (P : C) {Q R : C} (f : Q ⟶ R) (h : exact (0 : P ⟶ Q) f) :
 mono f
```

Finally, we show that if (f, g) is an exact sequence and f is a monomorphism, then f has the universal property of $\ker g$.

```
def kernel_of_mono_exact {P Q R : C} (f : P ⟶ Q) [mono f] (g : Q ⟶ R)
 (h : exact f g) : is_limit $ kernel_fork.of_ι f h.1
```

This is a direct consequence of the fact that $\ker \operatorname{coker} f$ has the universal property of $\ker g$, because if f is mono, then so is the map $P \to \operatorname{Ker} \operatorname{coker} f$ from image factorization. But since that map is also epi, it is an isomorphism, so the maps f and $\ker \operatorname{coker} g$ differ only by an isomorphism[8].

The reason why the last result is so interesting is because it naturally occurs in short exact sequences, which abound in homological algebra. We will see an application of this result in Section 6.

4. Another look at the category of modules

In order to be able to use the theory we are developing, it is necessary to connect it to the existing mathematical infrastructure in the Lean mathematical library. Specifically, we have to show that the categories we are interested in are in fact abelian and that our categorical characterization of exactness matches with the concrete notion of exactness.

Motivated by the examples we have seen in the introduction, we have performed this verification work for the category of R-modules[9].

Unlike in the preceding and the following sections, in this section, we will use the symbols $\operatorname{Ker} f$ and $\ker f$ for *the* kernel of a linear map $f\colon M \to N$ of modules, and we will use the symbols $\operatorname{Coker} f$ and $\operatorname{coker} f$ for *the* projection $\operatorname{coker} f\colon N \to M/\operatorname{Im} f$.

4.1. First results. Recall that for a category to be abelian, we need preadditivity, a zero object, kernels and cokernels, binary products and coproducts, and the fact that every monomorphism is a kernel and every epimorphism is a cokernel.

Preadditivity is very easy to verify. It is also clear what the zero object is: The trivial module. Lean provides a standard type with one member called `punit` for which countless instances have been defined. Among these is a module structure, so we can use `punit` as our zero object.

---

[8] Or, put differently, a monomorphism is the kernel of its cokernel. We already know that!

[9] Ideally, we would like to say that this also immediately implies that the category of abelian groups is also abelian—after all, the category of $\mathbb{Z}$-modules is "equal" to the category of abelian groups. The Lean mathematical library contains a proof of the fact that Ab and $\mathsf{Mod}_{\mathbb{Z}}$ are equivalent, but the machinery to transfer properties along equivalences does not exist yet. In particular, the Lean mathematical library develops the theory of modules and the theory of abelian groups independently, and for the transferred instance to have definitonally nice properties, it would be necessary to somehow connect the two different notions of kernels and quotients for abelian groups and modules.

For kernels, cokernels, products and coproducts, we want to reuse the existing theory developed in the linear algebra section of the Lean mathematical library. This works essentially as expected, but there is some friction due to the fact that this work appears at the intersection of two "languages": The language of modules and the language of categories. Even though the expressions `f ≫ g` and `g.comp f` (this is composition of linear maps) are definitionally equal (as they should be), if the wrong form of the expression is present in the goal, the simplifier may fail to apply a lemma. This is especially painful for arguments where lemmas from category-world and lemmas from algebra-world must be used in alternating fashion. In these proofs, we have made much use of the `erw` tactic, which—sometimes—manages to apply lemmas even if the goal is stated in the wrong form. This is much less verbose than making heavy use of the `change` tactic to bring the goal in the correct form. We also use the following helper definition, which promotes a linear map to a morphism in the category of modules:

```
def up (g : A →ₗ[R] B) : (of R A) ⟶ (of R B) := g
```

It is used in cases where Lean expects a morphism in the category of R-modules. Sometimes, supplying a linear map for this morphism works, but in many cases, it produces an error. The definition `up` can be used to fix the error. One might conjecture that writing `(g : (of R A) ⟶ (of R B))`, i.e., inserting a type annotation, should work just as well as `up g`, but we have found several examples of cases where the explicit type specification does not help.

4.2. Monomorphisms and epimorphisms. A prerequisite to showing that every monomorphism is a kernel is knowing that in the category of R-modules, monomorphisms are precisely injective linear maps and epimorphisms are precisely surjective linear maps.

To see why a monomorphism $f\colon M \to N$ of R-modules is injective, consider the diagram

$$\operatorname{Ker} f \underset{0}{\overset{\ker f}{\rightrightarrows}} M \xrightarrow{\ f\ } N.$$

Recall that in this section $\operatorname{Ker} f$ is the kernel of f as a submodule of M and $\ker f$ is the inclusion of $\operatorname{Ker} f$ into M. Since $\ker f \gg f = 0 = 0 \gg f$ and since f is a monomorphism, we have $\ker f = 0$ (i.e., the inclusion of the kernel of f into M is the zero map). But this implies $\operatorname{Ker} f = \operatorname{Im} \ker f = \operatorname{Im} 0 = 0$ (i.e., the kernel of f is the trivial module), and we are done since injective linear maps are precisely the linear maps with trivial kernel.

```
lemma ker_eq_bot_of_mono [mono f] : f.ker = ⊥
```

For the other direction, we simply have to notice that the usual definition of injectivity of functions tells us that $f(u(x)) = f(v(x))$ implies $u(x) = v(x)$. But by functional extensionality, this means that f is a monomorphism.

```
lemma mono_of_ker_eq_bot (hf : f.ker = ⊥) : mono f
```

Dual arguments and characterizations hold for f an epimorphism.

4.3. Monomorphisms are normal. Let $f\colon M \to N$ be a monomorphism of R-modules, i.e., by the preceding section, an injective linear map. We have to show that there is some morphism g such that f has the universal property of the kernel of g. There is an obvious choice for g: We choose g to be the quotient map $N \to N/\operatorname{Im} f$. Since we are in the category of modules, it is non-trivial to verify the universal property directly, because the only ways of constructing maps that are available to us are inclusions of subtypes and lifting via quotients, but with just an

injective linear map available, we have to get a bit creative. Consider the following diagram.

$$\begin{array}{ccccc} M & & \xrightarrow{f} & & N \xrightarrow{g} N/\operatorname{Im} f \\ \downarrow{\scriptstyle p} & & & & \uparrow{\scriptstyle i} \\ M/\operatorname{Ker} f & & \xrightarrow{h} & & \operatorname{Ker} g \end{array}$$

Here, p is is the projection, and i is the inclusion. Our goal is to construct h such that the diagram commutes and p and h are isomorphisms. Since we have instantiated our categorical kernels in the category of modules to be the inclusion of the kernel and i is just that, Lean already knows that i has the universal property of the kernel of g. Then we can use a categorical result that allows us to transport the universal property of i along the isomorphism $p \gg h$ to f.

It is obvious that p is an isomorphism: $\operatorname{Ker} f$ is the trivial submodule! How do we construct h? Mathematically, this is easy: The first isomorphism theorem for modules gives an isomorphism $M/\operatorname{Ker} f \cong \operatorname{Im} f$, but by construction, $\operatorname{Ker} g = \operatorname{Im} f$, and we are done.

If we try to formalize this intuition in Lean, we run into a problem: The equality $\operatorname{Ker} g = \operatorname{Im} f$ is not definitional. In practice, this means that the first isomorphism theorem, which is part of the Lean mathematical library, gives a map $M/\operatorname{Ker} f \to \operatorname{Im} f$, but what what we actually want is a map $M/\operatorname{Ker} f \to \operatorname{Ker} \operatorname{coker} f$, where by $\operatorname{coker} f$ we mean the projection onto the quotient $N/\operatorname{Im} f$. While we can use Lean's rewriting mechanism to change $\operatorname{Im} f$ into $\operatorname{Ker} \operatorname{coker} f$ or vice versa, this will make it impossible to verify that the diagram is commutative due to how the rewriting engine manipulates terms.

In the category theory library, there is a mechanism called `eq_to_iso` which is designed to deal with this kind of problem. If X and Y are two objects in a category, `eq_to_iso` takes a proof of $X = Y$ and outputs an isomorphism $X \cong Y$ which is just the identity on X. In some cases, this mechanism can be used to deal with problems involving non-definitional equality. However, in our case it is not applicable, once again because it prevents checking commutativity of the diagram.

The path we have chosen to solve this issue is to restate and reprove the isomorphism theorem in a form that yields a map $M/\operatorname{Ker} f \to \operatorname{Ker} \operatorname{coker} f$ directly. Very few modifications to the proof are necessary to achieve this goal. Remarkably, after doing this, commutativity of the diagram is pointwise definitional (i.e., the proof is `ext`, `refl`). This is a result of Lean's native support for quotient types.

Using this, we can complete the proof that every monomorphism has the universal property of the kernel of some morphism. The proof that every epimorphism is the cokernel of some morphism works dually. This concludes the proof that the category of modules is abelian.

4.4. Exact sequences. Finally, it remains to show that the notion of exact sequence we have defined in abelian categories coincides with the notion of exact sequence for modules that we have seen in the introduction.

We rely on two lemmas about modules[10].

```
lemma range_le_ker_iff {f : M₁ →ₗ[R] M₂} {g : M₂ →ₗ[R] M₃} :
  range f ≤ ker g ↔ g.comp f = 0

lemma ker_le_range_iff {f : M₁ →ₗ[R] M₂} {g : M₂ →ₗ[R] M₃} :
  ker g ≤ range f ↔ f.range.mkq.comp g.ker.subtype = 0
```

[10]see `https://github.com/leanprover-community/mathlib/commit/b50443020d530379735 7d82f121810a9917e269b`.

The first lemma states that $\operatorname{Im} f \subseteq \operatorname{Ker} g$ if and only if $g \circ f = 0$, while the second states that $\operatorname{Ker} g \subseteq \operatorname{Im} f$ if and only if the composition $\operatorname{Ker} g \to M_2 \to M_2/\operatorname{Im} f$ vanishes.

Both lemmas are mathematically obvious and allow very quick proofs using the tools provided by the Lean mathematical library for manipulating submodules. Now, our characterization of exactness becomes a mere matter of restating the above lemmas in a slightly different way, owed to the fact that the categorical kernels and cokernels we have defined for the category of modules are definitionally equal to the maps that appear in the second lemma of the previous listing.

```
lemma exact_is_exact : exact f g ↔ f.range = g.ker :=
⟨λ h, le_antisymm (range_le_ker_iff.2 h.1) (ker_le_range_iff.2 h.2),
 λ h, ⟨range_le_ker_iff.1 $ le_of_eq h, ker_le_range_iff.1 $ le_of_eq h.symm⟩⟩
```

5. Pseudoelements

Recall that the proof of the four lemma in the category of modules heavily relies on studying how elements of the modules are manipulated by linear maps. Using only what we have seen so far, this proof does not naturally generalize to abelian categories: In general, objects in abelian categories are not going to be sets with some additional structure as in the category of modules or the category of abelian groups, and hence there is no notion of "element".

Instead, a priori, we have to use arrow-theoretic proofs, like the proofs seen in Section 2. These proofs can be difficult both to write and to comprehend (consider, for example, the arrow-theoretic proof of the four lemma given by Kashiwara [14, Lemma 8.3.13]), so we are interested in a simpler way of proving theorems about abelian categories.

From a mathemathical perspective, there is a simple answer to this problem. It is called the Freyd-Mitchell embedding theorem [12]. We will not give the precise statement here, but in effect, it says the following: When proving a statement about a diagram in an abelian category C, without loss of generality we may assume that C is the category of R-modules for some (not necessarily commutative) ring R. So when we proved the four lemma for modules, we really already proved it for general abelian categories.

The downside to this theorem is that its proof is rather involved and giving a formal proof would be a somewhat monumental effort, so at the current time, this approach is impractical.

Pseudoelements are a method of formally assigning a set of "elements" to each object of an abelian category so that morphisms can be applied to them in a way that behaves similarly to elements in the category of abelian groups. In particular, there are very natural characterizations of properties such as being monic or epic and exactness in terms of pseudoelements.

In this way, the use of pseudoelements provides an alternative to the Freyd-Mitchell embedding theorem, but unlike the embedding theorem, the properties of pseudoelements are easy to prove, even formally. The downside is that there are some desirable properties of elements in the category of abelian groups that are not true for pseudoelements. For example, it is in general not true that two morphisms are equal if they act in the same way on pseudoelements (i.e., there is no `funext` for morphisms in abelian categories). This also means that we cannot construct morphisms in abelian categories by dictating their action on pseudoelements. Another disadvantage is that the pseudoelements of an object do not form an abelian group, so we cannot add or subtract them. There is a very limited construction on pseudoelements that in some ways behaves like subtraction, but it lacks some very important properties that we are used to from the category of abelian groups.

We will now describe our formalization of the therory of pseudoelements. There are various slightly different ways to construct pseudoelements [3, 5, 16]. Our formalization follows Borceux [5].

5.1. Defining pseudoelements. The pseudoelements of an object $P \in \mathsf{C}$ will be equivalence classes of arrows with codomain P. We will use Lean's quotient types, so we first collect the morphisms with codomain P in a type. This is easy using Lean's sigma types.

```
def with_codomain (P : C) := Σ Q, Q ⟶ P
```

Next, we have to define when two arrows are equivalent. We call two arrows $f : X \to P$ and $g : Y \to P$ *pseudo-equal* if there is some object R and epimorphisms $p : R \to X$ and $q : R \to Y$ such that $p \gg f = q \gg g$.

```
def pseudo_equal (P : C) (f g : with_codomain P) : Prop :=
∃ (R : C) (p : R ⟶ f.1) (q : R ⟶ g.1) [epi p] [epi q], p ≫ f.2 = q ≫ g.2
```

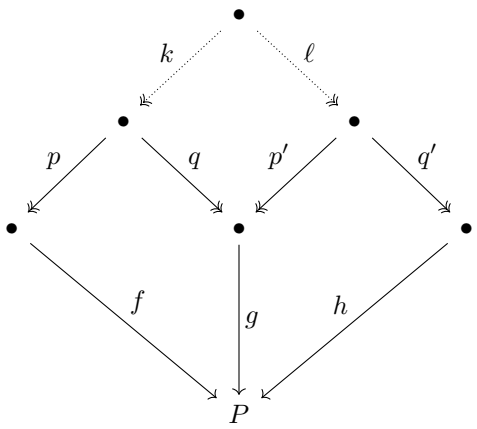
Note the use of `f.1` and `f.2` in the definition. Members of the type `with_codomain P` are really tuples of an object in C and a morphism from that object to P.

Let us show that pseudo-equality is an equivalence relation. Reflexivity and symmetry are apparent from the definition.

```
lemma pseudo_equal_refl {P : C} : reflexive (pseudo_equal P) :=
λ f, ⟨f.1, 𝟙 f.1, 𝟙 f.1, by apply_instance, by apply_instance, by simp⟩

lemma pseudo_equal_symm {P : C} : symmetric (pseudo_equal P) :=
λ f g ⟨R, p, q, ep, eq, comm⟩, ⟨R, q, p, eq, ep, comm.symm⟩
```

Transitivity is not much harder. Assume that f, g, h are arrows with codomain P and that f and g as well as g and h are pseudo-equal. This means that we have the following commutative diagram (so far, without the dotted arrows).



If we now take the pullback of q and p' we obtain the dotted arrows k and ℓ making the diagram commute. But by our final result in Section 2.4, these arrows are epimorphisms. Then so are $k \gg p$ and $l \gg q'$, and it is easily verified that $k \gg p \gg f = \ell \gg q' \gg h$, so f and h are pseudo-equal as required.

```
lemma pseudo_equal_trans {P : C} : transitive (pseudo_equal P) :=
λ f g h ⟨R, p, q, ep, eq, comm⟩ ⟨R', p', q', ep', eq', comm'⟩,
begin
  refine ⟨pullback q p', pullback.fst ≫ p, pullback.snd ≫ q', _, _, _⟩,
  { resetI, exact epi_comp _ _ },
  { resetI, exact epi_comp _ _ }
  { rw [category.assoc, comm, ←category.assoc, pullback.condition,
     category.assoc, comm', category.assoc] }
```

```
end
```

Now we are ready to define the pseudoelements of P as the quotient of `with_codomain P` by being pseudo-equal.

```
lemma pseudo_equal_equiv {P : C} : equivalence (pseudo_equal P) :=
⟨pseudo_equal_refl, pseudo_equal_symm, pseudo_equal_trans⟩

def pseudoelements.setoid (P : C) : setoid (with_codomain P) :=
{ r := pseudo_equal P,
  iseqv := pseudo_equal_equiv }

local attribute [instance] pseudoelements.setoid

def pseudoelements (P : C) : Type (max u v) := quotient (pseudoelements.setoid P)
```

Next, we use Lean's coercion mechanism to allow us to write `P` instead of `pseudoelements P` in places where Lean is able to infer what we mean.

```
def object_to_sort : has_coe_to_sort C :=
{ S := Type (max u v),
  coe := λ P, pseudoelements P }
```

This, together with a similar mechanism for morphisms, will make proofs using pseudoelements look almost identical to proofs using actual elements in a concrete category.

5.2. The action of a morphism on pseudoelements. A morphism $f\colon P \to Q$ transforms an arrow $a\colon \bullet \to P$ with codomain P into an arrow $a \gg f\colon \bullet \to Q$ with codomain Q by composition.

```
def app {P Q : C} (f : P ⟶ Q) (a : with_codomain P) : with_codomain Q :=
⟨a.1, a.2 ≫ f⟩
```

Composing with the projection to the quotient yields a mapping `with_codomain P → pseudoelements Q`. Our goal is to lift this mapping to a function `pseudoelements P → pseudoelements Q`. The rules of quotient types tell us that such a lift exists if two pseudo-equal arrows map to the same pseudoelement. But this is a simple consequence of the associativity rule for composition.

```
lemma pseudo_apply_aux {P Q : C} (f : P ⟶ Q) (a b : with_codomain P) :
  a ≈ b → [app f a] = [app f b] :=
λ ⟨R, p, q, ep, eq, comm⟩, quotient.sound ⟨R, p, q, ep, eq,
  show p ≫ a.2 ≫ f = q ≫ b.2 ≫ f,
    by rw [←category.assoc, comm, category.assoc]⟩
```

Using this, we can define our lift.

```
def pseudo_apply {P Q : C} (f : P ⟶ Q) : P → Q :=
quotient.lift (λ (g : with_codomain P), [app f g]) (pseudo_apply_aux f)
```

Note the different types of arrows in the previous listing. `P ⟶ Q` is the type of morphisms in the category `C`. On the other hand, the `P → Q` is the type of functions between the types `P` and `Q`. But P and Q are not types. Instead, they are objects in `C`, so Lean looks for ways to interpret P and Q as types, and it finds the coercion we set up earlier. Therefore `P → Q` is actually shorthand for `pseudoelements P → pseudoelements Q`.

Now we tell Lean that it can regard a morphism $f\colon P \to Q$ as a function from the pseudoelements of P to the pseudoelements of Q. This allows us to use Lean's function application syntax for morphisms and pseudoelements.

```
def hom_to_fun {P Q : C} : has_coe_to_fun (P ⟶ Q) := ⟨_, pseudo_apply⟩
local attribute [instance] hom_to_fun
```

The first property of morphisms and pseudoelements that we will show is that function composition is the same as composition of morphisms with regard to the action on pseudoelements.

```
theorem comp_apply {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) (a : P) :
  (f ≫ g) a = g (f a) :=
quotient.induction_on a $ λ x, quotient.sound $ by unfold app; rw category.assoc
```

The proof demonstrates a typical approach to proving statements about quotients. Elements of a quotient type are opaque objects, since support for quotients is provided by the Lean kernel. It is possible to prove properties of quotients by using `quotient.induction_on`, which turns goals of the form Pa, where P is some statement and a is a member of a quotient type into goals of the form $P[a]$, where a is now a member of the base type of the quotient and $[a]$ is its equivalence class inside the quotient. In other words, `quotient.induction_on` is a convenient way to argue using surjectivity of the quotient map. It uses type inference to automatically transform the goal. If the goal is then of the form $[a] = [b]$, we can use `quotient.sound` to transform the goal into $a \sim b$, where $\sim$ is the equivalence relation that we quotiented out. Now we have removed all mention of quotient types from our goal and can write our proof.

5.3. The zero pseudoelement. We already mentioned at the beginning of this section that there is no abelian group structure on the pseudoelements of P. However, there is a distinguished pseudoelement for every object P: It is an equivalence class that contains precisely all the zero morphisms with codomain P. The existence of this pseudoelement is asserted by the following lemma, which states that for some fixed zero arrow with codomain P, an arrow f is pseudo-equal to this arrow if and only if f is also a zero arrow.

```
lemma pseudo_zero_aux {P : C} (Q : C) (f : with_codomain P) :
  f ≈ (0 : Q ⟶ P) ↔ f.2 = 0 :=
⟨λ ⟨R, p, q, ep, eq, comm⟩, (preadditive.cancel_zero_iff_epi p).1 ep _ f.snd $
    by erw [comm, has_zero_morphisms.comp_zero],
  λ hf, ⟨biproduct f.1 Q, biproduct.fst, biproduct.snd, by apply_instance,
    by apply_instance,
    by erw [hf, has_zero_morphisms.comp_zero, has_zero_morphisms.comp_zero]⟩⟩
```

It is sensible to call this equivalance class the *zero pseudoelement*. We tell Lean to associate the symbol 0 with this pseudoelement.

```
def pseudo_zero {P : C} : P := [(0 : P ⟶ P)]
instance {P : C} : has_zero P := ⟨pseudo_zero⟩
```

The zero pseudoelement behaves like one would expect: A morphism maps the zero pseudoelement to the zero pseudoelement and a zero morphism sends every pseudoelement to the zero pseudoelement.

```
theorem apply_zero {P Q : C} (f : P ⟶ Q) : f 0 = 0 :=
by erw [pseudo_apply_bar, has_zero_morphisms.zero_comp]; exact zero_eq_zero

theorem zero_apply {P Q : C} (a : P) : (0 : P ⟶ Q) a = 0 :=
quotient.induction_on a $ λ a',
  by erw [pseudo_apply_bar, has_zero_morphisms.comp_zero]; exact zero_eq_zero
```

5.4. Characterizations in terms of pseudoelements. So far, we have developed a theory of pseudoelements, but we have not demonstrated why this theory is useful at all. In this section, we give characterizations of interesting properties of morphisms in an abelian category in terms of pseudoelements. In the next section, we will see how these characterizations can be used to prove theorems.

The first characterization is a special case of the existensionality principle: A morphism is the zero morphism if (and, as we already know, only if) it maps every pseudoelement to the zero pseudoelement.

```
theorem zero_morphism_ext {P Q : C} (f : P ⟶ Q) : (∀ a, f a = 0) → f = 0 :=
λ h, by { rw ←category.id_comp _ f,
  apply (pseudo_zero_iff ((1 P ≫ f) : with_codomain Q)).1,
  exact h (1 P) }
```

Next, we show that the known charactization of injectivity for modules, namely that a linear map is injective if and only if its kernel is trivial, extends to pseudoelements:

```
theorem pseudo_injective_of_mono {P Q : C} (f : P ⟶ Q) :
  mono f → function.injective f

lemma zero_of_map_zero_of_pseudo_injective {P Q : C} (f : P ⟶ Q) :
  function.injective f → (∀ a, f a = 0 → a = 0)

theorem mono_of_zero_of_map_zero {P Q : C} (f : P ⟶ Q) :
  (∀ a, f a = 0 → a = 0) → mono f
```

Similarly, an epimorphism is precisely a morphism that is surjective on pseudoelements:

```
theorem pseudo_surjective_of_epi {P Q : C} (f : P ⟶ Q) :
  epi f → function.surjective f

theorem epi_of_pseudo_surjective {P Q : C} (f : P ⟶ Q) :
  function.surjective f → epi f
```

Generalizing this, we provide characterizations of exactness that match the situation known from abelian groups.

```
theorem pseudo_exact_of_exact {P Q R : C} {f : P ⟶ Q} {g : Q ⟶ R} :
  exact f g → (∀ a, g (f a) = 0) ∧ (∀ b, g b = 0 → ∃ a, f a = b)

theorem exact_of_pseudo_exact {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) :
  (∀ a, g (f a) = 0) ∧ (∀ b, g b = 0 → ∃ a, f a = b) → exact f g
```

Next, we construct a very weak form of subtraction of pseudoelements: If f is a morphism and x, y are pseudoelements satisfying $f(x) = f(y)$, then we find some pseudoelement z such that $f(z) = 0$ and for every other morphism g such that $g(y) = 0$ we have $g(x) = g(z)$.

```
theorem sub_of_eq_image {P Q : C} (f : P ⟶ Q) (x y : P) : f x = f y →
  ∃ z, f z = 0 ∧ ∀ (R : C) (g : P ⟶ R), (g : P ⟶ R) y = 0 → g z = g x
```

Finally, we provide a computational tool that is not directly inspired from proofs in the category of abelian groups, but is nonetheless useful. It states that the categorical pullback behaves with regards to pseudoelements like it would with regards to elements of abelian groups.

```
theorem pseudo_pullback {P Q R : C} {f : P ⟶ R} {g : Q ⟶ R} {p : P} {q : Q} :
  f p = g q → ∃ (s : pullback f g), pullback.fst s = p ∧ pullback.snd s = q
```

Borceux [5, Proposition 1.9.5] additionally asserts that the pseudoelement s is unique. However, his proof of uniqueness is very brief and we were not able to turn his sketch into a full pen-and-paper proof, so we have not formalized this claim.

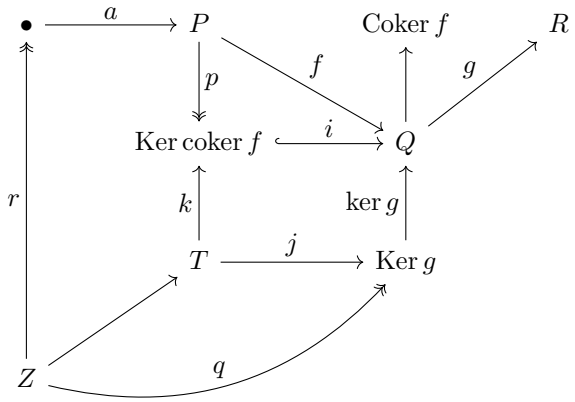
The proofs of these characterizations range from very easy to somewhat involved. As an example, we present the proof of one of the more difficult characterizations: If a pair of morphisms is exact on pseudoelements, then it is exact.

```
theorem exact_of_pseudo_exact {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) :
  (∀ a, g (f a) = 0) ∧ (∀ b, g b = 0 → ∃ a, f a = b) → exact f g :=
```

PROOF. We have to show that $f \gg g = 0$ and $\ker g \gg \operatorname{coker} f = 0$. The first claim follows directly from the hypothesis using the existensionality lemma for the zero morphism, and it remains to show the second claim.

```
λ ⟨h₁, h₂⟩, ⟨zero_morphism_ext _ $ λ a, by rw [comp_apply, h₁ a],
begin
```

We refer to the following diagram.



An easy calculation shows that if we apply g to the pseudoelement induced by $\ker g$, we get the zero pseudoelement.

```
have : g (kernel.ι g) = 0 := comp_zero _ _ (kernel.condition _),
```

This means that we get a preimage $a\colon \bullet \to P$ satisfying $f[a] = [\ker g]$, which means that there is some Z and epimorphisms r and q such that $r \gg a \gg f = q \gg \ker g$.

```
obtain ⟨a', ha⟩ := h₂ _ this,
obtain ⟨a, ha'⟩ := quotient.exists_rep a',
rw ←ha' at ha,
obtain ⟨Z, r, q, er, eq, comm⟩ := quotient.exact ha,
```

Let T be the pullback of i and $\ker g$, where $f = p \gg i$ is the image factorization of f. Then $r \gg a \gg p$ and q factor through the pullback.

```
obtain ⟨z, hz₁, hz₂⟩ := pullback.lift' (kernel.ι (cokernel.π f)) (kernel.ι g)
  (r ≫ a.2 ≫ factor_thru_image f) q (by simp only [category.assoc, image.fac];
    exact comm),
let j : pullback (kernel.ι (cokernel.π f)) (kernel.ι g) ⟶ kernel g :=
    pullback.snd,
```

Now j is a monomorphism, because i is, and an epimorphism, because q is. But in an abelian category, a monomorphism that is an epimorphism is an isomorphism.

```
have pe : epi j, by resetI; exact epi_of_epi_fac hz₂,
have pm : mono j := by apply_instance,
haveI : is_iso j := @mono_epi_iso _ _ _ _ _ _ pm pe,
```

But then we can write $\ker g \gg \operatorname{coker} f$ as $j^{-1} \gg k \gg i \gg \operatorname{coker} f$, with $i = \ker \operatorname{coker} f$, so $\ker g \gg \operatorname{coker} f = 0$ as required.

```
rw (iso.eq_inv_comp (as_iso j)).2 pullback.condition.symm,
simp only [category.assoc, kernel.condition, has_zero_morphisms.comp_zero]
end⟩  □
```

6. Diagram chasing in abelian categories

6.1. The four lemma revisited. Using the theory developed in Section 5, we can now state and prove the four lemma in a general abelian category V.

```
variables {A B C D A' B' C' D' : V}
variables {f : A ⟶ B} {g : B ⟶ C} {h : C ⟶ D}
variables {f' : A' ⟶ B'} {g' : B' ⟶ C'} {h' : C' ⟶ D'}
variables {α : A ⟶ A'} {β : B ⟶ B'} {γ : C ⟶ C'} {δ : D ⟶ D'}
variables (fg : exact f g) (gh : exact g h) (fg' : exact f' g') (gh' : exact g' h')
variables (comm₁ : α ≫ f' = f ≫ β) (comm₂ : β ≫ g' = g ≫ γ)
variables (comm₃ : γ ≫ h' = h ≫ δ)

lemma four (hα : epi α) (hβ : mono β) (hδ : mono δ) : mono γ :=
```

Proof. Since the structure of this proof is identical to the structure of the proof given in Section 3.2 of the first chapter, we will omit the commentary and only show the formal proof.

```
mono_of_zero_of_map_zero _ $ assume (c : C) (hc : γ c = 0), show c = 0, from

have h c = 0, from
  suffices δ (h c) = 0, from zero_of_map_zero _ (pseudo_injective_of_mono _) _
    this,
  calc δ (h c) = h' (γ c) : by rw [←comp_apply, ←comm₃, comp_apply]
        ... = h' 0     : by rw hc
        ... = 0        : apply_zero _,

exists.elim ((pseudo_exact_of_exact gh).2 _ this) $ assume (b : B) (hb : g b = c),
  have g' (β b) = 0, from
    calc g' (β b) = γ (g b) : by rw [←comp_apply, comm₂, comp_apply]
          ... = γ c     : by rw hb
          ... = 0       : hc,

  exists.elim ((pseudo_exact_of_exact fg').2 _ this) $ assume (a' : A') (ha' : f' a' =
    β b),
    exists.elim (pseudo_surjective_of_epi α a') $ assume (a : A) (ha : α a = a'),

    have f a = b, from
      suffices β (f a) = β b, from pseudo_injective_of_mono _ this,
      calc β (f a) = f' (α a) : by rw [←comp_apply, ←comm₁, comp_apply]
            ... = f' a'    : by rw ha
            ... = β b      : ha',

    calc c = g b     : hb.symm
      ... = g (f a) : by rw this
      ... = 0       : (pseudo_exact_of_exact fg).1 _   □
```

This proof is nearly identical to the proof in the category of modules given in Section 3.2 of the first chapter. The only difference is that we have substituted lemmas about modules (for example `ker_eq_bot`) with lemmas about pseudoelements (for example `mono_of_zero_of_map_zero`). See also Figure 2 for a comparison of the goal states at the end of the proofs, further illustating how our use of Lean's features makes working with pseudoelements look and feel just like working with actual elements in a concrete category.

6.2. Constructing morphisms. We have already discussed in the introduction of Section 5 that the notion of pseudoelement is too weak to define a morphism

```
A B C D A′ B′ C′ D′ : Type u,
f : A →ₗ[R] B,
g : B →ₗ[R] C,
h : C →ₗ[R] D,
f′ : A′ →ₗ[R] B′,
g′ : B′ →ₗ[R] C′,
h′ : C′ →ₗ[R] D′,
α : A →ₗ[R] A′,
β : B →ₗ[R] B′,
γ : C →ₗ[R] C′,
δ : D →ₗ[R] D′,
fg : range f = ker g,
gh : range g = ker h,
fg′ : range f′ = ker g′,
gh′ : range g′ = ker h′,
comm₁ : ⇑f′ ∘ ⇑α = ⇑β ∘ ⇑f,
comm₂ : ⇑g′ ∘ ⇑β = ⇑γ ∘ ⇑g,
comm₃ : ⇑h′ ∘ ⇑γ = ⇑δ ∘ ⇑h,
hα : range α = ⊤,
hβ : ker β = ⊥,
hδ : ker δ = ⊥,
c : C,
hc : ⇑γ c = 0,
this : c ∈ ker h,
b : B,
hb : ⇑g b = c,
this : ⇑β b ∈ ker g′,
a′ : A′,
ha′ : ⇑f′ a′ = ⇑β b,
a : A,
ha : ⇑α a = a′,
this : ⇑f a = b
⊢ c = 0
```

(A) In the category of R-modules

```
A B C D A′ B′ C′ D′ : V,
f : A ⟶ B,
g : B ⟶ C,
h : C ⟶ D,
f′ : A′ ⟶ B′,
g′ : B′ ⟶ C′,
h′ : C′ ⟶ D′,
α : A ⟶ A′,
β : B ⟶ B′,
γ : C ⟶ C′,
δ : D ⟶ D′,
fg : exact f g,
gh : exact g h,
fg′ : exact f′ g′,
gh′ : exact g′ h′,
comm₁ : α ≫ f′ = f ≫ β,
comm₂ : β ≫ g′ = g ≫ γ,
comm₃ : γ ≫ h′ = h ≫ δ,
hα : epi α,
hβ : mono β,
hδ : mono δ,
c : ↥C,
hc : ⇑γ c = 0,
this : ⇑h c = 0,
b : ↥B,
hb : ⇑g b = c,
this : ⇑g′ (⇑β b) = 0,
a′ : ↥A′,
ha′ : ⇑f′ a′ = ⇑β b,
a : ↥A,
ha : ⇑α a = a′,
this : ⇑f a = b
⊢ c = 0
```

(B) In a general abelian category V

FIGURE 2. Comparison of goal states at the end of the proof of the four lemma at different levels of generality. By using coercions and quotient types, the proof using pseudoelements (right) looks very similar to the proof using real elements (left), but it gives a much more general result.

in an abelian category by dictating its action on pseudoelements. This is problematic insofar as some of the diagram lemmas assert the existence of some morphism $f\colon X \to Y$ with certain properties. The proof of such a diagram lemma in the category of R-modules will generally proceed in four steps:

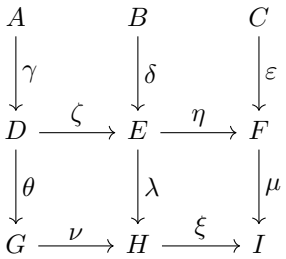
1. Let $x \in X$. Chase this x along the diagram to construct some $y \in Y$ and define $f(x) \coloneqq y$.
2. If the construction of y involved some choice (for example when using an exactness property), show that y is independent of that choice.
3. Show that this elementwise definition of f is linear.
4. Show that f has the desired properties by chasing elements.

In a general abelian category, the first three steps are not available. Instead, the only real way to construct new morphisms is by using universal properties. So a modified proof strategy for general abelian categories is this:

1. Construct f by repeated applications of universal properties.
2. Show that f has the desired properties by chasing pseudoelements.

An example of this approach is the *kernels lemma*.

LEMMA. Consider the following commutative diagram with exact rows and columns:



Assume that δ, ε and ν are monomorphisms. Then there are unique morphisms $\alpha\colon A \to B$ and $\beta\colon B \to C$ such that the diagram commutes. Furthermore, (α, β) is exact.

REMARK. The name "kernels lemma" stems from the fact that by the results of Section 3, δ is a kernel of λ and ε is a kernel of μ.

PROOF. Let us first construct the morphisms α and β. We have

$$\gamma \gg \zeta \gg \lambda = \gamma \gg \theta \gg \nu = 0 \gg \nu = 0,$$

so $\gamma \gg \zeta$ factors through δ, as δ is a kernel of λ.

```
def fill_left [mono δ] : { x : A ⟶ B // x ≫ δ = γ ≫ ζ } :=
limit_kernel_fork.lift' _ (kernel_of_mono_exact _ _ δ1) (γ ≫ ζ) $
  by rw [category.assoc, comm₁, ←category.assoc, γθ.1, zero_comp]
```

We can define β in exactly the same way.

```
def fill_right [mono ε] : { x : B ⟶ C // x ≫ ε = δ ≫ η } :=
limit_kernel_fork.lift' _ (kernel_of_mono_exact _ _ εμ) (δ ≫ η) $
  by rw [category.assoc, comm₂, ←category.assoc, δ1.1, zero_comp]
```

The diagram now commutes by construction and uniqueness follows from uniqueness of lifts.

```
variables {α : A ⟶ B} {β : B ⟶ C}
variables (comm₃ : α ≫ δ = γ ≫ ζ) (comm₄ : β ≫ ε = δ ≫ η)
include comm₃ comm₄

lemma fill_left_unique [mono δ] : α = fill_left comm₁ comm₂ γθ δ1 εμ ζη νξ
lemma fill_right_unique [mono ε] : β = fill_right comm₁ comm₂ γθ δ1 εμ ζη νξ
```

It remains to show the exactness result. We show it for α and β instead of for `fill_left` and `fill_right` because in applications, we may already have some α and β and we only want the exactness result, so phrasing it this way saves us an application of the uniqueness lemma.

```
lemma kernels [mono δ] [mono ε] [mono ν] : exact α β
```

The reader who has familiarized themselves with the results of Section 5.4 will have no difficulties filling in the proof, which is a bit lengthy but completely routine. We will omit it here, and return to the proof once we can automate it[11]. □

[11]In the meantime, the reader may find a non-automated proof at `https://github.com/TwoFX/lean-homological-algebra/blob/125b4b1dabdf7df88f517954ace6a6aa9f7054c8/src/diagram_lemmas.lean#L199`.

CHAPTER 3

An interactive tactic for diagram chasing

The goal of this chapter will be to develop a tactic that can synthesize diagram chasing proofs based on some form of hint regarding how to perform the chase. It should take little more time to formalize a proof by diagram chase than to perform it to a live audience, and certainly much less time than to write down all the arguments in detail.

In the first section of this chapter, we describe our algorithm for diagram chasing from a high level perspective. Next, we will describe the implementation of our algorithm as a tactic using the Lean metaprogramming framework [10]. Finally, we present examples of proofs created using our tactic.

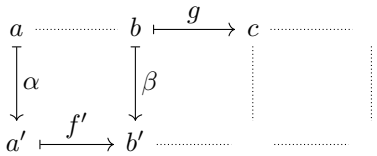
1. An algorithm for diagram chasing

We will first sketch the general approach to diagram chasing that our tactic should follow. To do so, we must determine what the core information encapsulated in a diagram chasing argument is. The answer is given by Aluffi [1, p. 180]: A diagram chasing argument is determined by the *path* through the diagram that the chase takes.

As an example, recall the statement of the four lemma.

$$\begin{array}{ccccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C & \xrightarrow{h} & D \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \delta} \\ A' & \xrightarrow{f'} & B' & \xrightarrow{g'} & C' & \xrightarrow{h'} & D' \end{array}$$

The proof amounts to showing that given $c \in C$ such that $\gamma(c) = 0$, we already have $c = 0$. Here is a graphical representation of the proof:



The image describes how to construct the various (pseudo)elements appearing in the proof. Given this recipe, it only remains to show that these constructions are possible and that the construction finishes the proof. In the case of the four lemma, the latter amounts to showing that $g(f(a)) = c$, from which $c = 0$ by exactness.

There are two types of constructions. In the first type, which we call *pushforward*, we have an element x of the domain of some morphism $f\colon X \to Y$ and wish to construct an element y of the codomain such that $f(x) = y$. For example, this is the face for the construction of b' from b in the four lemma. This is very simple: We simply set $y \coloneqq f(x)$.

In the second type, which we call *pullback*, we have an element $y \in Y$ of the codomain of some morphism $f\colon X \to Y$ and want to construct an element x of the domain such that $f(x) = y$. There are two ways to do so: We can either use that

f is an epimorphism or that f is part of some exact pair (f, g). However, we have shown in Section 3 that being an epimorphism is equivalent to being part of an exact pair $(f, 0)$, so it will suffice to consider pulling back along exact sequences.

Let

$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

be an exact sequence and let $y \in Y$. The characterization of exactness from Section 5.4 tells us that a preimage $x \in X$ satisfying $f(x) = y$ exists if and only if $g(y) = 0$. Therefore, our algorithm must have some way of proving this type of equality.

1.1. Proving equalities involving pseudoelements. We need to consider the following problem: We are given hypotheses of the form

$$f_1 \gg \cdots \gg f_n = g_1 \gg \cdots \gg g_m$$

for morphisms f_i and g_i, which we will call *commutativity lemmas* as well as hypotheses of the form

$$(f_1 \gg \cdots \gg f_n)(a) = (g_1 \gg \cdots \gg g_m)(b)$$

for morphisms f_i and g_i and pseudoelements a and b, which we will call *element lemmas*. Our goal is to find a proof for a validity of an equality of the second type.

If all of our lemmas are of the second type, this is a problem in the theory of equality with uninterpreted function symbols, which is equivalent to the congruence closure problem, for which efficient algorithms are known [21], even if, like in our case, the decision procedure needs to emit a proof [22]. In fact, an extension of the concruence closure algorithm to the type theory in Lean has been developed by Selsam and de Moura [24] and is available in Lean via the `cc` tactic.

However, we have decided against using congruence closure in our algorithm. An immediate obstacle is that we have to support lemmas of the first type. This is not a big problem: In our application there is only a finite and easy to explicitly enumerate set of possible arguments of each type, so every lemma of the first type can be efficiently transformed in a finite set of equalities of the second type.

A much larger problem is the performance of the `cc` tactic in practice. Despite being one of the few Lean tactics that are implemented in C++ rather than in Lean itself, in our tests the runtime of `cc` ranged in the seconds even for very simple tasks, which makes it unsuitable for being used in the "inner loop" of some larger automation.

For this reason, our algorithm uses a simple approach based on depth first search that searches the entire graph of possible lemma applications for a path from the left hand side to the right hand side. Note that this approach will, in general, not even terminate when there are lemmas such as $a = f(a)$. However, the lemmas that occur in the proofs of the diagram lemmas are well-behaved and thus do not exhibit this looping behavior[1].

2. Infrastructure for diagram chasing

2.1. Structures for common objects. Before we can implement the algorithm discussed in Section 1, it will be advantageous to provide some infrastructure to facilitate the implementation. This has two main benefits: Working with an

[1]Of course, using a breadth first search instead of a depth first search solves this problem and the algorithm will always terminate and find a proof, provided that a proof exists. The runtime of the BFS-based approach will not be polynomial, but we strongly suspect that due to the small input sizes a BFS-based approach would outperform the `cc` tactic in practice. We have not tested this approach because the DFS-based approach described above worked well in all applications that we tested.

abstraction of the raw expressions will make manipulating them easier and will enable precomputing expensive operations in a comfortable fashion.

First, we define a structure that represents a morphism in an abelian category.

```
meta structure morphism :=
(ex : expr)
(domain : expr)
(codomain : expr)
(app : expr)
```

Here, `ex` is the expression that is represented by the morphism object, and `app` is the expression `coe_fn ex`, i.e., the function on pseudoelements induced by the morphism. We carry this expression as part of the structure since it is needed in multiple places and somewhat expensive to create.

Another important building block are expressions of the form $f_1(f_2(\dots(f_n(a))))$. They are represented by a list of morphisms in the order indicated by the numbering and an expression for the element a.

```
meta def morphism_chain := list morphism

meta structure diagram_term :=
(ms : morphism_chain)
(elem : expr)
```

Using these definitions, it is easy to define the two types of lemmas mentioned in the description of our algorithm.

```
meta structure commutativity_lemma :=
(lhs rhs : morphism_chain)
(ex : expr)

meta structure element_lemma :=
(lhs rhs : diagram_term)
(ex : expr)
```

Finally, we provide a structure for an exact sequence.

```
meta structure exactness_lemma :=
(lhs rhs : morphism_chain)
(ex : expr)
```

When used in combination, these structures will allow us to operate semantically on the objects we care about in our proof instead of having to work directly with the underlying expressions. While Lean's support for pattern matching on expressions is very good, adding a layer of abstraction will make recursing on the structure of these expressions more convenient and less error-prone.

2.2. Working with the structures. For each of the structures defined in the previous section, we provide a function that takes an arbitrary expression and attempts to interpret that expression as an instance of that structure and returns an instance upon success. As an example, here is the parsing function for a morphism:

```
meta def as_morphism (e : expr) : tactic (option morphism) :=
do
 `(%%l ⟶ %%r) ← infer_type e | return none,
 app ← mk_app `coe_fn [e],
 return $ some ⟨e, l, r, app⟩
```

Note that `diagram_term` does not come with an `ex` field like the other structures. The reason for this is that we frequently need to manipulate these terms without needing the expression, for example in incorrect branches of the depth-first search or when doing recursion on a diagram term. Since building the expression for a diagram term is quite expensive, we only do so when it is actually needed.

```
meta def as_expr : diagram_term → expr
| ⟨[], e⟩ := e
| ⟨t::ts, e⟩ := expr.app t.app $ as_expr ⟨ts, e⟩
```

2.3. Collecting lemmas. Before we can execute our tactic, we have to scan the environment for morphisms and lemmas and create instances of our structures which the tactic can use. Lean provides the local context of a tactic proof via the `local_context` tactic. Using our parsing functions, we can automatically extract all relevant expressions in the local context and create instances for them.

```
meta def get_morphisms : tactic (list morphism) :=
local_context >>= list.mfiltermap as_morphism
```

This works well for small, self-contained proofs. However, we would like to be able to split longer proofs into multiple definitions, lemmas and theorems, and these declarations would not appear in the local context. Hence, we have to provide a way for the user to mark definitions and lemmas as relevant for diagram chasing.

Lean provides a mechanism for this: user-defined attributes. We declare an attribute `chase`, and the user will be able to mark definitions and lemmas by prepending `@[chase]` to the declaration.

```
meta def chase_attribute : user_attribute := {
  name := `chase,
  descr := "A definition or lemma that can be used in a diagram chase."
}
```

Since a definition or lemma depends on the variables of the theorem we are proving (for example the diagram we are chasing in), the user will need to supply a parameter that contains all the information necessary to instantiate all parameters of the lemmas. Given this parameter, called `f` in the following listing, we can retrieve all tagged declarations.

```
meta def get_lemmas_from_attribute (f : expr) : tactic (list expr) :=
attribute.get_instances `chase >>= (list.mfiltermap $ λ n,
  (some <$> (do e ← resolve_name n, to_expr ``(%%e %%f) tt ff)) <|>
    return none)
```

2.4. Extending the tactic monad. We collect all information about the local context in a separate structure called `chase_data`.

```
meta structure chase_data :=
(morphisms : list morphism)
(comm_lemmas : list commutativity_lemma)
(elem_lemmas : list element_lemma)
(exact_lemmas : list exactness_lemma)
```

We provide a tactic that scans the local context and, optionally, the lemmas marked by our attribute and returns an object of type `chase_data`.

```
meta def mk_chase_data (e : option expr) : tactic chase_data
```

Passing an argument of type `chase_data` to every function that uses it would quickly get tedious, especially since we sometimes need to modify the argument, for example when a new hypothesis is generated about an element that was constructed as part of a diagram chase. Instead, we use a construction similar to that used by `smt_tactic` in Lean core [10]: We use the state monad transformer to create a new tactic type called `chase_tactic` that automatically carries the `chase_data` object.

```
@[reducible]
meta def chase_tactic :=
state_t chase_data tactic
```

Every normal `tactic` becomes a `chase_tactic` simply by ignoring the parameter.

```
meta instance {α} : has_coe (tactic α) (chase_tactic α) :=
⟨monad_lift⟩
```

Given a `chase_data` object, we can run a `chase_tactic` like a `tactic`.

```
meta def run_chase_tactic_with_data {α} (t : chase_tactic α) (d : chase_data) :
  tactic α :=
do (res, _) ← t.run d, return res

meta def run_chase_tactic {α} (e : option expr) (t : chase_tactic α) : tactic α :=
mk_chase_data e >>= run_chase_tactic_with_data t
```

A `chase_tactic` can retrieve and assign the `chase_data` using `get` and `put`. In addition, we provide a tactic that adds an element lemma as well as its symmetric version to the context.

```
meta def add_elem_lemma (l : element_lemma) : chase_tactic unit :=
do
  ⟨ms, cs, es, el⟩ ← get,
  ls ← l.symm,
  put ⟨ms, cs, (ls::l::es), el⟩
```

3. A tactic for proving equalities in commutative diagrams

The goal of this section is to develop a tactic `commutativity` that can solve goals of the form

$$(f_1 \gg \cdots \gg f_n)(a) = (g_1 \gg \cdots \gg g_m)(b).$$

Using the infrastructure developed in the previous section, very little remains to do. We simply implement a depth first search that explores all possible lemma applications. The set of possible applications of lemmas can be enumerated naively. If we have found a path, we build the term out of the applied lemma as well as the basic lemmas `congr_fun` and `congr_arg`, which state that $f = g$ implies $f(a) = g(a)$ and that $a = b$ implies $f(a) = f(b)$, respectively.

We deviate from this algorithm in one aspect, namely the handling of zero pseudoelements and zero morphisms. Making the unmodified algorithm work with zero pseudoelements and zero morphisms would require adding a large number of hypotheses of the form $f(0) = 0$ and $0(a) = 0$. Instead of adding all of these hypotheses, we add special handling for this case.

Assume that the goal is $t_1 = t_2$, where t_1 and t_2 are diagram terms. If the DFS discovers a proof of $t_1 = h_1(\dots h_n(a))$, where either $h_i = 0$ for some i or $a = 0$, then it will try to find a proof of $t_2 = 0$ using DFS and, if that succeeds, directly construct a proof of $t_1 = 0$ using the lemmas `apply_zero` and `zero_apply`. It is easy to see that this procedure finds a proof of $t_1 = t_2$ if and only if the original algorithm finds a proof of $t_1 = t_2$ (note however, that in general, the two algorithms will find different proof terms), but this method avoids generating a large number of auxiliary hypotheses.

4. Constructing new pseudoelements

The final missing piece of the tactic is the implementation of the pushforward and pullback procedures and the "end user interface" of the tactic.

Pushing forward and pulling back is merely a matter of building applications of the relevant lemmas about pseudoelements that we have proved and (in the case of pulling back) calling `tactic.intro` with the correct name. We use the builtin `get_unused_name` to avoid name clashes of hypotheses about our newly introduced elements.

We export our tactic as an interactive tactic called `chase` with the following syntax:

```
chase e using [f, g, h] with a b c at d
```

means that we chase an element e along the morphisms f, g, h (in that order), automatically detecting whether a chase step should be a pullback or a pushforward. The intermediate elements generated will be called `a`, `b` and `c` and the tactic will use `d` as an argument to lemmas marked with `@[chase]` in order to derive morphisms or hypotheses from them.

If there are too few names or the `with` parameter is omitted, then automatically generated names will be used. If the `at` parameter is omitted, then no lemma marked with `@[chase]` will be used (and in fact, the scan for these lemmas is skipped entirely).

5. Applications

We are now ready to use our tactics to give concise proofs of diagram lemmas.

5.1. The four lemma again. For a final time, we consider the four lemma. As sketched in Section 1, to prove the four lemma, we start with a pseudoelement c, chase it along g, β, f' and α to obtain elements b, b', a' and a, and finally show $f(a) = b$, which implies $0 = g(f(a)) = g(b) = c$. Using our automation, writing down this proof sketch is sufficient to give an automated proof of the four lemma.

```
lemma four [epi α] [mono β] [mono δ] : mono γ :=
mono_of_zero_of_map_zero _ $ λ c hc,
begin
  chase c using [g, β, f', α] with b b' a' a,
  have : f a = b, by commutativity,
  commutativity
end
```

5.2. The kernels lemma. In Section 6.2, we did not give a proof of the statement that the two constructed morphisms form an exact sequence. With our automation, giving a proof becomes very comfortable. Here is the full proof.

```
lemma kernels [mono δ] [mono ε] [mono ν] : exact α β :=
begin
  apply exact_of_pseudo_exact,
  split,
  { intro a,
    commutativity },
  { intros b hb,
    chase b using [δ, ζ, γ] with e d a,
    exact ⟨a, by commutativity⟩ }
end
```

5.3. The restricted snake lemma. As a final demonstration of our tactic, we present a proof of the *restricted snake lemma*. Before we give the statement, recall that the sequence

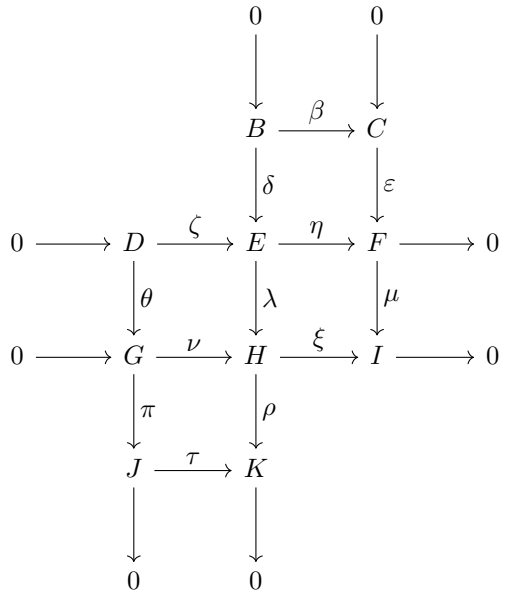
$$0 \longrightarrow X \xrightarrow{f} Y$$

is exact if and only if f is a monomorphism. Dually,

$$X \xrightarrow{f} Y \longrightarrow 0$$

is exact if and only if f is an epimorphism.

Lemma (Restricted Snake Lemma). Consider the following commutative diagram with exact rows and columns.



Then there is a morphism $\omega\colon C \to J$ such that

$$B \xrightarrow{\beta} C \xrightarrow{\omega} J \xrightarrow{\tau} K$$

is an exact sequence.

Remark. Some authors call the above theorem the snake lemma, while we call it the restricted snake lemma. The result we refer to as the snake lemma has the same statement as the restricted snake lemma, except that we do not require ζ to be a monomorphism and we do not require ξ to be an epimorphism. This weakening of the hypothesis is often necessary, for example in the construction of the long exact sequence in cohomology (cf. [27, Chapter 1]).

The snake lemma can be proved using the restricted snake lemma. We have formalized both the proof of the restricted snake lemma and the proof of the snake lemma, but we will not present the proof of the latter here. In the following, we discuss our formalization of the proof of the restricted snake lemma.

Proof. In a similar fashion to the kernels lemma, the proof consists of two parts: the construction of ω and the verification of the exact sequence. We will not give the details of either of the steps here, as the proof is rather long.

There are many benefits to splitting the proof of a statement like this into multiple parts. A problem we already observed for the kernels lemma is that this means that we have to carry around a large number of parameters and supply them whenever we need to refer to a previous definition or result. For the number of variables and the size of the proof of the restricted snake lemma, this is impractical. Instead, we collect all data of the commutative diagram in a structure[2]. We can then attach the `chase` attribute defined in Section 2.3 to the structure projections so that our automation is able to access the morphisms and exactness properties. In the remainder of the proof, `d` will be an instance of our structure.

[2]This is inaccurate. The truth is that we use two structures, as a workaround for an undiagnosed Lean performance problem which causes the direct definition using a single structure to time out.

The proof uses the full range of characterizations in terms of pseudoelements that we have seen, including "subtraction" and pullbacks. At the heart of the proof is a description of the constructed morphism ω in terms of pseudoelements:

```
lemma ω_char (c : d.C) (e : d.E) (g : d.G) (h₁ : d.η e = d.ε c) (h₂ : d.ν g = d.κ e) :
  d.π g = (ω d) c
```

Note the different notations: For a morphism that is part of the diagram, we use dot notation like `d.π`, whereas for objects that we constructed as part of the proof,we use d as a paramater, as in ω `d`.

Using this characterization, proving the exactness of (ω, τ) is a breeze for our automation.

```
theorem ωτ : exact (ω d) d.τ :=
begin
  apply exact_of_pseudo_exact,
  split,
  { intro c,
    chase c using [d.ε, d.η, d.κ, d.ν] with f e h g at d,
    have := ω_char _ c e g (by commutativity) (by commutativity),
    commutativity at d },
  { intros j hj,
    chase j using [d.π, d.ν, d.κ, d.η, d.ε] with g h e f c at d,
    have := ω_char _ c e g (by commutativity) (by commutativity),
    exact ⟨c, by commutativity⟩ }
end
```

The proof of exactness of (β, ω) is not quite as concise, but we can still delegate all the heavy lifting to our automation. We refer to the source code for the details. □

CHAPTER 4

Related work and conclusion

We have developed the theory of abelian categories in version 3 of the Lean interactive theorem prover. While numerous libraries of formal mathematics cover some amount of category theory, the author is aware of only a few that contain substantial results on limits. As far as we are aware, this is the second time that abelian categories have been formalized in a theorem prover, after UniMath [26], a library based on univalent foundations [25] written for the theorem prover Coq [9]. Our work is not a port of the results in UniMath, but develops the theory from scratch using the integrated approach to limits in mathlib and using Lean's advanced features such as quotient types, which significantly improves readability of the results. Furthermore, no attempt has been made in UniMath to provide automation similar to what we have implemented.

For these reasons, we suspect that stating and proving the snake lemma and constructing the long exact sequence in cohomology would be a large undertaking in `UniMath`, whereas we have already made significant progress towards this goal in Lean by giving a proof of the snake lemma.

By using native quotient types and coercions, we were able to develop the theory of pseudoelements, which constitute an important proof technique in the theory of abelian categories, in a way that is very comfortable to use. Indeed, consider the statement that a sequence is exact on pseudoelements if and only if it is exact. In UniMath, this lemma is stated as in the following listing.

```
Lemma PEq_isExact {x y z : ob A} (f : x ⟶ y) (g : y ⟶ z)
    (H : f · g = ZeroArrow (to_Zero A) _ _) :
  isExact A hs f g H ↔
  ∏ (b : PseudoElem y) (H : b · g = ZeroArrow (to_Zero A) _ _),
    ∑ (a : PseudoElem c), PEq (PseudoIm a f) b.
```

Compare this to the way we stated this lemma in Lean:

```
theorem exact_iff_pseudo_exact {P Q R : C} (f : P ⟶ Q) (g : Q ⟶ R) :
  exact f g ↔ (∀ a, g (f a) = 0) ∧ (∀ b, g b = 0 → ∃ a, f a = b)
```

Our statement has a lot less visual clutter, and is easy to understand even for the layman, because we use function application syntax rather than `PseudoIm`, the equality symbol rather than `PEq` and the zero symbol for zero morphisms and pseudoelements. In addition, we have developed automation that can semiautomatically synthesize proofs within this theory.

Using these tools, we have given the first ever formally verified proof of the snake lemma for general abelian categories[1]. This result is an important milestone in the construction of the exact sequence in cohomology associated to a short exact sequence of cochain complexes. This construction is a staple in many parts of modern mathematics, for example as a computational tool in algebraic topology [17], or for constructing the long exact sequences for Ext and Tor [6], to name just the most basic examples.

[1] In fact, as far as we are aware, as a special case this is also the first ever formally verified proof of the snake lemma for abelian groups or modules.

However, the work presented in this thesis is just a first step and there are several required next steps before our work can be put to proper use. Our work needs to be integrated with the formalization of enriched category theory under development by Morrison et al. and then upstreamed into the Lean mathematical library. Furthermore, our work pretends that abelian categories are the most general setting in which concepts like exactness can be studied. This is not quite true—for example, there is a concept of regular category (cf. [5]), which generalizes the concept of abelian category to include, for example, the category of sets. In this thesis, we have made no attempt to find the most general setting in which a concept can be defined, so definitions and proofs need to be reworked where possible in an effort to increase generality.

Next, we have to carry out the construction of the long exact sequence in cohomology. The basic definitions of cochain complexes and cohomology have recently been added to the Lean mathematical library, and based on these definitions we have already shown the existence of functorial induced morphisms in cohomology for cochain complexes over an abelian category. We are currently working on deriving the long exact sequence in cohomology using the snake lemma. Since the full embedding theorem is unavailable, this poses some interesting technical challenges which exceed the scope of this thesis.

Finally, an important milestone is connecting this theory with the existing algebraic structures defined in Lean and its mathematical library, making sure that the tools we are developing can actually be applied. We have already demonstrated using the category of R-modules that our definitions of abelian category and exact sequence are chosen correctly. In line with this, showing that the categorical definition of cohomology coincides with the direct definition in terms of R-modules will be very important to applications. Almost certainly, the two constructions will not be definitionally equal, so what we will be looking for is a natural isomorphism between the functors from the category of chain complexes of R-modules to the category of $\mathbb{Z}$-graded R-modules sending a chain complex to its categorical cohomology and its directly defined cohomology, respectively.

Bibliography

1. Paolo Aluffi. *Algebra: Chapter 0*. Reprinted with corrections by the American Mathematical Society. Vol. 104. Graduate Studies in Mathematics. American Mathematical Society, 2016.
2. Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem proving in Lean*. 2015. URL: `https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf`.
3. George M Bergman. "A note on abelian categories—translating element-chasing proofs, and exact embedding in abelian groups". In: *unpublished* (1974). URL: `http://math.berkeley.edu/~gbergman/papers/unpub/elem-chase.pdf`.
4. Francis Borceux. *Handbook of Categorical Algebra: Volume 1, Basic Category Theory*. Vol. 50. Encyclopedia of Mathematics. Cambridge University Press, 1994.
5. Francis Borceux. *Handbook of Categorical Algebra: Volume 2, Categories and Structures*. Vol. 51. Encyclopedia of Mathematics. Cambridge University Press, 1994.
6. Kenneth S Brown. *Cohomology of groups*. Vol. 87. Graduate Texts in Mathematics. Springer Science & Business Media, 2012.
7. Kevin Buzzard, Johan Commelin, and Patrick Massot. "Formalising perfectoid spaces". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA, 2020, pp. 299–312.
8. The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381.
9. Coq Development Team. *The Coq Proof Assistant*. 1989–2020. URL: `https://coq.inria.fr/` (visited on 04/06/2020).
10. Gabriel Ebner et al. "A Metaprogramming Framework for Formal Verification". In: *Proceedings of the ACM on Programming Languages* 1 (Aug. 2017). DOI: `10.1145/3110278`. URL: `https://doi.org/10.1145/3110278`.
11. Ramon Fernández Mir. "Schemes in Lean". MA thesis. London, UK: Imperial College London, June 2019.
12. Peter J Freyd. *Abelian categories*. Harper's Series in Modern Mathematics. New York: Harper & Row, 1964.
13. Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 163–179.
14. Masaki Kashiwara and Pierre Schapira. *Categories and sheaves*. Vol. 332. Grundlehren der mathematischen Wissenschaften. Berlin: Springer, 2006.
15. Robert Y Lewis. "A formal proof of Hensel's lemma over the p-adic integers". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Lisbon, Portugal, 2019, pp. 15–26.

16. Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Graduate Texts in Mathematics. Springer Science & Business Media, 2013.
17. J Peter May. *A concise course in algebraic topology*. University of Chicago press, 1999.
18. Bhavik Mehta et al. *Topos theory for Lean*. 2020. URL: `https://github.com/b-mehta/topos` (visited on 04/06/2020).
19. Leonardo de Moura et al. *The Lean Theorem Prover (community fork)*. 2020. URL: `https://github.com/leanprover-community/lean` (visited on 04/06/2020).
20. Leonardo de Moura et al. "The Lean theorem prover (system description)". In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388.
21. Greg Nelson and Derek C Oppen. "Fast decision procedures based on congruence closure". In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 356–364.
22. Robert Nieuwenhuis and Albert Oliveras. "Fast congruence closure and extensions". In: *Information and Computation* 205.4 (2007), pp. 557–580.
23. Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". In: *All about Proofs, Proofs for All*. Ed. by Bruno Woltzenlogel Paleo and David Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015. URL: `https://hal.inria.fr/hal-01094195`.
24. Daniel Selsam and Leonardo de Moura. "Congruence closure in intensional type theory". In: *International Joint Conference on Automated Reasoning*. Springer. 2016, pp. 99–115.
25. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: `https://homotopytypetheory.org/book`.
26. Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics*. URL: `https://github.com/UniMath/UniMath`.
27. Charles A Weibel. *An introduction to homological algebra*. Vol. 38. Cambridge studies in advanced mathematics. Cambridge university press, 1995.
28. Andrew Wiles. "Modular elliptic curves and Fermat's last theorem". In: *Annals of mathematics* 141.3 (1995), pp. 443–551.

Eidesstattliche Erklärung

Hiermit erkläre ich, Markus Himmel, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift