Exploring Formalisation

A Primer in Human-Readable Mathematics in Lean 3 with Examples from Simplicial Topology

Clara Löh

Book project
Extended version of: ProofLab: Seminar on Simplicial Topology, WS 2021/22

Version of June 20, 2022
clara.loeh@mathematik.uni-regensburg.de
Fakultät für Mathematik, Universität Regensburg, 93040 Regensburg

Contents

0
Introduction

Proof assistants allow us to formalise mathematical definitions, examples, theorems, and proofs, and to verify these proofs. The key feature of such a formalisation is the associated machine-checkable guarantee for correctness of proofs. The formalisation of mathematics in a proof assistant also has the following benefits:

- Implementing mathematics requires a very thorough understanding of the subject and all corner cases. Therefore, a successful formalisation leads to a better knowledge of the details.
- A formalisation in a proof assistant can lead to executable code and thus provides the opportunity to run experiments on concrete examples.
- Proof assistants encourage abstraction. It is often easier to formalise concepts declaratively through universal properties instead of through concrete, potentially cumbersome, constructions. This can lead to a clearer view on core issues and ideas.
- Interacting with a proof assistant is fun. And addictive.

All of these benefits are not only valuable for research, but also in a teaching setting: A better understanding of details can increase the awareness for potential pitfalls, short-cuts that might not be obvious to novices in the field, and other struggles of students. If students learn basic proof techniques via a proof assistant, they can obtain immediate feedback [52] and can experiment with proofs and examples. Moreover, students can see in more practical terms how abstractions simplify mathematical life.

These notes are an extended version of the material for the seminar *ProofLab: Simplicial Topology* at Universität Regensburg in WS 2021/22. The

goal of this seminar was to learn the basics of formalising mathematics in a proof assistant. In addition to basic examples from first-year mathematics, as sample theory we chose simplicial topology, which is a higher-dimensional version of graph theory. For the implementation we used the Lean 3 proof assistant and the mathlib library, which covers a significant part of undergraduate mathematics.

The focus of this seminar was to find straightforward formalisations that lead to human-readable machine-verifiable proofs. While some of the challenges are specific to simplicial complexes and Lean, most of them are generic and will occur also with other fields and other proof assistants.

These notes are *not* a comprehensive treatment of the full programming language Lean and the underlying dependent type theory. Instead, we study the transition between pen-and-paper mathematics and the machine counterpart as well as the design options in this process. This is illustrated with various examples, proceeding in three stages: Examples from first year courses, examples from simplicial topology, and recent sample results from algebraic and geometric topology.

Overview of these notes

- **Chapter 1** We start with a quick introduction to proof assistants and to Lean in particular. This requires a basic understanding of foundations of mathematics, i.e., how objects, statements, and proofs can be described in a formal language. In particular, this includes the ability to assemble and disassemble logical statements.

- **Chapter 2** We then consider a first selection of simple examples: Injectivity and surjectivity of maps, induction and sums over finite sets, commutators in group theory, and a fundamental property of real numbers. We always follow the same structure: We first develop a pen-and-paper version and then formalise it in Lean. The examples are chosen in such a way that basic proof patterns occur. The later examples also illustrate the interaction with the mathlib library.

- **Chapter 3** As a larger example theory, we consider basic concepts and examples from simplicial topology. We formalise simplicial complexes, simplicial maps, and the Euler characteristic. During this formalisation, we discuss typical design questions and experiment with concrete examples.

- **Chapter 4** Finally, we consider situations from recent research in algebraic and geometric topology. These serve as examples of how proof assistants encourage abstraction and thus can lead to interesting perspectives.

Prerequisites I tried to keep things as simple as possible. On the mathematical side, basic experience with proofs and logical formulae as well as with basic

mathematical objects (e.g., sets, maps, groups, reals) is required. Chapter 4 uses basic terminology on categories and functors; Section 4.2 is probably a bit mysterious without background knowledge in basic algebraic topology (fundamental groups, universal coverings, cohomology, CW-complexes), but the formalisation strategy will be understandable without knowing the semantics of these notions.

On the programming side, acquaintance with basic programming concepts is helpful (e.g., declarations, types, basic control structures, recursion). Prior exposure to the functional programming paradigm is a bonus.

Exercises Each chapter ends with a selection of exercises around the formalisation of mathematical terminology, theorems, and proofs. Skeleton files and solutions to selected exercises of Chapter 2 and Chapter 3 are provided in the git repository:

Source code All source code discussed in theses notes is available in a git repository:

https://gitlab.com/polywuisch/mapa_notes update!

Further reading These notes are merely a primer in the formalisation of mathematics. To get a deeper understanding of proof assistants and formalisation it is highly recommended to dive into the literature, tutorials, and larger formalisation projects.

- J. Avigad, L. de Moura, S. Kong. *Theorem Proving in Lean*, Release 3.23.0, https://leanprover.github.io/theorem_proving_in_lean/, 2021.
- J. Avigad, K. Buzzard, R.Y. Lewis, P. Massot. Mathematics in Lean. https://leanprover-community.github.io/mathematics_in_lean/
- A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, J. Limperg. *The Hitchhiker's Guide to Logical Verification*, 2021 Standard Edition, 2021. https://github.com/blanchette/logical_verification_2021/raw/main/hitchhikers_guide.pdf
- Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, An EATCS Series, Springer, 2004.
- K. Buzzard, J. Commelin, P. Massot. Lean perfectoid spaces. https://leanprover-community.github.io/lean-perfectoid-spaces/
- K. Buzzard, M. Pedramfar. The natural number game. https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/
- D. P. Friedman, D. T. Christiansen. *The Little Typer*, MIT Press, 2018.
- G. Gonthier. Formal proof–the four-color theorem, *Notices Amer. Math. Soc.*, 55(11), 1382–1393, 2008.
  Implementation in Coq: https://github.com/math-comp/fourcolor

- T. Hales. Formal Abstracts.
  https://formalabstracts.github.io/
- L. Lamport. How to write a 21st century proof, *J. Fixed Point Theory Appl.*, 11(1), 43–63, 2012.
- Lean community. Learning Lean.
  https://leanprover-community.github.io/learn.html
- Lean community. Get started with Lean.
  https://leanprover-community.github.io/get_started.html
- Lean community. mathlib.
  https://leanprover-community.github.io/mathlib-overview.html
- Lean community. Using leanproject.
  https://leanprover-community.github.io/leanproject.html
- Lean community. Papers about Lean.
  https://leanprover-community.github.io/papers.html
- Lean for the curious mathematician, 2020.
  https://leanprover-community.github.io/lftcm2020/
- P. Massot. The sphere eversion project.
  https://leanprover-community.github.io/sphere-eversion/blueprint/
- The Xena project.
  https://xenaproject.wordpress.com/

Errata Comments and corrections for these notes can be submitted by email to clara.loeh@mathematik.uni-r.de; errata will be collected at

https://loeh.app.uni-regensburg.de/mapa/ update!

A glimpse into the crystal ball These notes are based on Lean 3. The latest stable release of Lean is still within Lean 3, but the community is preparing the transition to the successor Lean 4 [39, 7, 54], including porting mathlib [38].

Lean 4 is *not* backwards compatible with Lean 3. However, most of the basic principles addressed in these notes will also be applicable to formalising mathematics in Lean 4 or even other proof assistants such as Coq or Isabelle. In my experience, the initial switch from pen-and-paper mathematics to a proof assistant is more challenging than switching between different proof assistants.

Acknowledgements update!

Happy hacking!

Regensburg, July 2022 Clara Löh

1

The Lean Proof Assistant

Proof assistants allow us to formalise mathematical statements and to verify formalised mathematical proofs.

The Lean proof assistant uses type theory as its foundation. We quickly explain how one can formalise statements and proofs in this setup.

This is a very minimalistic introduction to Lean "for the working mathematician". In particular, we will not explain the underlying dependent type theory and we will not give a systematic introduction to all concepts and programming paradigms available in Lean. More information on Lean can be found in the standard Lean introductions and documentation [6, 8, 33]. For dependent types, there is a step-by-step introduction available [23].

We will illustrate and practice basic proof techniques in Lean in Chapter 2.

Overview of this chapter.

1.1 Proof Assistants

Proofs are an essential part of mathematics and the formal, objective concept of proof distinguishes mathematics from most other sciences.

- What is important about proofs? Correctness!
- What is interesting about proofs? The underlying ideas.

Unfortunately, many conventional pen-and-paper proofs contain small (or substantial) inaccuracies or gaps. Most of these problems can be fixed; however, it would be beneficial for readers if there was an a priori guarantee for correctness.

- What is a proof assistant?

  A proof assistant is a programming language together with a corresponding interpreter/compiler that allows us to formalise mathematical objects and facts; this includes definitions, theorems, proofs, and examples. The main task of a proof assistant is not to *find* proofs, but to *check* proofs for correctness. Proof assistants can thus provide certificates for correctness, based on the assumption that the proof assistant in itself is correct.

- Why do we need proof assistants?

  Proof assistants help to detect and avoid mistakes. Moreover, indirectly, they also lead to a better overall understanding of mathematical connections and in the long run will lead to more systematic and structured ways to generalise results to new contexts. In addition to applications in theoretical mathematics, proof assistants are used in the analysis of complex processes, systems, and algorithms in computer science and in industrial applications [28, 2, 40].

- Why aren't proof assistants used by default by all mathematicians?

  As of today, the formalisation of mathematical theories in proof assistants is still more cumbersome than on paper (because one has to be much more precise and careful ...). As soon as a critical mass of mathematical basics is formalised, this will change. There are several ongoing projects in this direction [27, 64, 30, 18, 37, 65, 51, 12, 40] and so there is hope that in the not too distant future proof assistants are more widely used, both in research and in teaching.

  One of the big challenges is to use proof assistants in such a way that the formalisation is not only easy to check by the interpreter/compiler, but also comprehensible for human readers: The beauty of mathematics does not lie in complicated technical details, but in the underlying ideas.

Another difficulty is more subtle: Formalisation in a proof assistant requires a solid understanding of formalisation and foundations. In particular, one has to understand to which extent the logical foundations of the proof assistant coincide with the intended mathematical meaning. In this course, we will mostly ignore this delicate point.

- Which proof assistants are in use?

  There are many proof assistants; currently, the most popular general purpose proof assistants are Coq [10], Isabelle [57], Lean, ... In this course, we will use Lean [33, 6, 8, 5].

- Why Lean?

  The proof assistant Lean is an active and dynamically developing project; the Lean community already created many mathematical libraries and Lean is used in several ambitious formalisation projects in mathematics [65, 51, 12]. Moreover, Lean offers a convenient web interface [35] that allows to experiment with Lean without a proper installation.

  The vast progress currently seen in the Lean community implies that some of the details of these notes will be quickly outdated. However, the underlying principles and ideas will be more long-lived and also apply to a certain extent to other proof assistants.

1.2 Foundations

The formalisation of mathematics consists of the following components:

- a universe of objects,
- a language of logic,
- a concept of proof,
- and usually a meta language (in which all of this is formulated).

The fact that these different levels interact with each other in various ways makes it challenging to give a complete and rigorous treatment of foundations of mathematics.

In classical pen-and-paper proofs, we usually work with set theory (e.g., ZFC or NBG), with a classical logic (but there are also other interesting choices!), and a proof calculus that enables us to decompose, construct, and recombine logical statements.

Lean is a functional programming language. Lean is based on type theory and the calculus of inductive constructions [19] instead of set theory, offers a choice between classical and intuitionistic logic, and the proof calculus is

based on the Curry–Howard isomorphism (a correspondence between proofs and implementations of terms/functions with suitable types; Section 1.4).

Caveat 1.2.1. Thus, strictly speaking, statements formalised in Lean do not necessarily have the same meaning as their pen-and-paper counterparts (even though they might look "equal"). For our applications and large parts of mathematical research, these subtle differences will not be relevant.

1.3 Types

The programming language Lean is statically typed, i.e., all terms are subject to typing rules [14] and their types are declared or inferred at compile-time. We will see in subsequent chapters how Lean types can be used to formalise mathematical properties, statements, proofs, structures, and examples. On a first reading, the rest of this section could be skipped and only consulted when the need arises.

Types in Lean can be constructed by combining the following constructions; these constructions do have overlaps and the list is not complete.

- Function types (symbol $\to$): If `A` and `B` are types, then `A → B` denotes the type of all functions from `A` to `B`.
- Pair types (symbol $\times$): If `A` and `B` are types, then `A × B` denotes the type of all pairs with first component of type `A` and second component of type `B`.
- Inductive types (keyword `inductive`): Inductive types have a finite list of constructors that may recursively depend on each other.

  Examples of inductive types are simple enumeration types (p. 27), the natural numbers (p. 31), lists, trees, ...
- Record types (keyword `structure`): Record types have only a single constructor on a finite list of fields. These fields can be accessed through the corresponding projections.

  Examples of record types are pair types or combined mathematical structures that consist of multiple components or properties (p. 53). Records in Lean are extensible and thus can be used to build theories hierarchically.
- Moreover, Lean has some support for subtypes and quotients [14, Section 2.7.1].

In addition, Lean supports type classes:

- Type classes are a variant of record types (keyword `class` or, equivalently, `@[class] structure`). Type classes can be viewed as formalising a finite set of axioms on the given parameter types. In this view, giving an `instance` of a type class corresponds to proving the corresponding axioms on the parameter types. Type classes and their instances are subject to type class inference [63].

  Examples of type classes are all sorts of algebraic structures such as monoids, groups, etc. Analogously, also metric structures and concepts from category theory are axiomatised through type classes [37].

  It is possible to define multiple instances for the same type class over the same parameter types; mathematically, this corresponds to, e.g., defining multiple group structures on the same carrier set.

As Lean is based on *dependent* type theory, types are first-class citizens and types can involve constraints and dependencies in the arguments and result types. In technical terms, the sum and product types underlying function types, inductive types, and record types are *dependent* sums (dependent Sigma types: Σ) and *dependent* products (dependent Pi types: Π) [59, 16].

For example, the append function on vectors (of given length) has the following type – where the result type depends on the argument types:

```
Π {α : Type u_1} {n m : ℕ},
vector α n → vector α m → vector α (n + m)
```

Here, `n m : ℕ` states that `n` and `m` are of type ℕ. The type `Type u_1` is a type universe, as explained below.

To escape Russell-ish paradoxes involving "the type of all types" in the presence of powerful constructions, Lean types are organised in a hierarchy:

- `Prop` (equivalently, `Sort 0`) is the bottom of the type hierarchy.

  Expressions of type `Prop` are viewed as "propositions": The type `Prop` contains the boolean constants `true` and `false`; moreover, logical connectives turn members of type `Prop` into results of type `Prop`.

- `Type u` (equivalently, `Sort (u+1)`) for a natural number `u` is a member of type `Type (u+1)`.

  `Type` is an abbreviation for `Type 0` and `Type*` is an abbreviation for `Type u` for some unspecified universe level `u`.

  The type constructors such as → operate on these type universes.

In general, equality is a delicate matter. The standard notion of equality in Lean is inductive equality [6, Sections 4.2, 7.8, 7.9] and there is also support for heterogeneous equality.

In Lean, the type `Prop` enjoys a simple equality behaviour: Members of type `Prop` satisfy *propositional extensionality*: If `A : Prop` and `x, y : A`, then `x` and `y` are considered equal in every respect.

1.4 Proofs

Proofs derive statements from axioms and hypotheses via deduction rules. The central deduction rule in mathematical proofs is *modus ponens*:

> If statement A is proved and if the implication $A \implies B$ is proved, then it follows that also B is proved.

The *Curry–Howard isomorphism* [9, 53] identifies

- statements with types and
- proofs of statements with elements of the corresponding type.

Under this translation, modus ponens corresponds to function application:

> Given an element of type `A` and a function `A → B`, we obtain an element of type `B`.
>
> More explicitly: If `x : A` (i.e., `x` is of type `A`) and if `f : A → B`, then `f x : B`. As common in functional programming languages, function application is denoted by juxtaposition: `f x` stands for "`f` applied to `x`".

Hence, proofs in Lean are just implementations of functions (and also syntactically look like that):

A lemma of the form below thus says that under the hypothesis that `x` has type `A` ("satisfies `A`"), then `first_result` of `x` has type `φ x` (i.e., `φ x` is "satisfied"). Usually, `φ x` is a type that represents a concrete mathematical statement of type `Prop` (e.g., "`x` is prime"). Therefore, in this context, propositional extensionality is also referred to as *proof irrelevance*: For the system it only matters that we have a proof of a statement, but it will not distinguish between different proofs when using the result.

```
lemma first_result
      (x : A)
    : φ x
:=
begin

  ...

end
```

A proof of this lemma is nothing but an implementation of a function of type `Π (x : A), φ x` and this proof is enclosed in `:= begin ... end`. On the Lean side, `lemma` and `theorem` are shorthand for `@[irreducible] def`, i.e., definitions whose implementations will not be touched when the statement of the lemma is used somewhere else.

```
import tactic -- standard proof tactics

lemma binomial_solution
      (x : ℤ)
    : x^2 - 2 * x + 1 = 0 ↔ x = 1
:=
begin
  have one_is_solution : x = 1 → x^2 - 2 * x + 1 = 0, from
  begin
    assume x_is_1 : x = 1,
    show x^2 - 2 * x + 1 = 0,
         by {rw[x_is_1], ring},
  end,

  have solution_is_one : x^2 - 2 * x + 1 = 0 → x = 1, from
  begin
    assume x_is_solution : x^2 - 2 * x + 1 = 0,

    have xminus1_squared_is_0 : (x-1)^2 = 0, by
    calc (x-1)^2 = x^2 - 2 * x + 1 : by ring_nf
             ... = 0               : by exact x_is_solution,

    have xminus1_is_0 : x - 1 = 0,
         by exact pow_eq_zero xminus1_squared_is_0,

    calc x = x - 1 + 1 : by ring
       ... = 1         : by {rw[xminus1_is_0], ring},
  end,

  show _, by exact {mp  := solution_is_one,
                    mpr := one_is_solution},
end
```

Figure 1.1: A first Lean proof; try it online! [46]

Usually, proofs of statements involve several steps. Instead of writing such a proof directly as a single function term, one can alternatively construct the proof step by step in *tactic mode* (enclosed in `begin` and `end`). The claim of a lemma becomes a *goal* that has to be reached. During the proof such goals are manipulated; depending on the used deduction rules and intermediate claims, goals are resolved, reformulated, or new goals are added. The proof is complete once all goals are reached.

A first example of such a proof is given in Figure 1.1. Ignoring the Lean-specific language for now, the proof is comprehensible for a human reader.

Examples with full explanations of all essential steps will be the topic of the rest of these notes.

The proof state is tracked and checked by the Lean system. In interactive development environments, looking at the proof state helps to write (or follow) proofs. The proof state at each position in the source code can be queried, displaying the hypotheses and open goals that are active at this position.

However, in the source code itself, parts of the state remain implicit and sometimes also the implicit order can play a role; in order to highlight the underlying ideas and to increase readability and robustness it is therefore recommended to make proofs more explicit than Lean would require. While library code often strives for brevity, in these notes, we will aim at producing human-readable proofs. For example, in Figure 1.1, the proof of `solution_is_one` could be abbreviated in the following short (but intransparent) way:

```
assume x_is_solution : x^2 - 2 * x + 1 = 0,
nlinarith
```

The individual steps in proofs usually consist of the elimination or introduction of logical constructs (see Chapter 2.1.2):

- The proof of a combined statement requires an introduction (e.g., the introduction of quantifiers or logical connectors).
- The extraction of components of combined statements requires an elimination (e.g., the extraction of the components of an and-statement or the extraction of a witness from an existential statement).

While this plan sounds simple enough, formalisation can sometimes turn into a frustrating endeavour. As always [1]: Don't panic!

How to get stuck:

- Trying to do too many steps at once.
- Trying to prove a false claim.

How to get unstuck:

- Check on the pen-and-paper level that the claim is indeed correct.
- Try first to establish the overall proof structure: Try to subdivide your proof into well-structured arguments, using `sorry` to take intermediate steps for granted. If necessary, subdivide into further steps.
- Help the proof assistant to unfold the relevant definitions.
- Use `hint` to get a list of tactics that make progress on the current goal.

  One should be aware that local progress might not result in global progress. Often `hint` will suggest the normalising tactics `norm_num`, `norm_cast`, or the "finishing" tactics `tauto`, `finish` that make an attempt at automatically solving the current goal completely.

	proving it (introduction)		using it (elmination)	
$\forall$	`inj_comp_injfirst`	(p. 24)	`inj_comp_injfirst`	(p. 24)
$\exists$	`surj_comp_surjsecond`	(p. 25)	`surj_comp_surjsecond`	(p. 25)
$\wedge$	`surj_inj_bij`	(p. 24)	`bij_inj`	(p. 23)
$\vee$	`union_is_subset_closed`	(p. 78)	`union_is_subset_closed`	(p. 78)
$\rightarrow$	`inj_comp_injfirst`	(p. 24)	`inj_comp_injfirst`	(p. 24)
$\neg$	`not_inj_f`	(p. 27)	`iso_implies_not_not_isomorphic`	(p. 130)

Figure 1.2: Examples of standard proof patterns in these notes

`def`	definition
`let`	local definition
`lemma`, `theorem`, `have`, ...	claims a statement; introduces corresponding goals; requires a proof
`show`	claims/solves a goal; if successful, this goal will be removed from the active list of goals
`by`	gives a justification; can interact with other statements or proof strategies via `apply`, `exact`, `simp`, `rw`, `refine`, `arith`, ...
`calc`	a calculation chain of, e.g., equalities or inequalities

Figure 1.3: Basic Lean vocabulary

- Use `suggest` to get a list of theorems or definitions that apply via `exact` or `refine` to the current situation.

  These suggestions do not necessarily make progress in any way, but can still be helpful.

- Use `library_search` to try to solve the current goal by a *single* application of a result from the library.

- Browse mathlib to find relevant results.

- Look for a similar statement/proof in a library.

- Repeat!

A selection of the Lean vocabulary and tactics are collected in Figure 1.3 and Figure 1.4. More details on the exact syntax and arguments/options can be found in the documentation [5, 6, 33, 8]. Examples of standard proof patterns are listed in Figure 1.2. Standard Lean unicode symbols and their VSCode transcriptions are listed in Figure 1.5.

`assume`	introduces an identifier, as preparation for a proof of an all-statement or an implication
`unfold`	unfolds a definition
`use`	prove an existential statement from an example
`rcases`	recursive pattern matching; extracts, e.g., a witness from an existential term
`cases`	proof by case distinction
`induction`	proof by induction (not only over natural numbers)
`hint`, `suggest`, `library_search`	tactics that search for ways to make progress in the proof
`sorry`	pretends to be a proof (useful for developing the overall structure of a proof) or a value of the given type (usually dangerous)
`exact`	solves the current goal by giving an exact proof term
`refine`	like `exact`, but can contain wildcards/holes `_` that may lead to new goals
`apply`	tries to match the conclusion of the argument to the current goal; might create new goals
`simp`, `dsimp`	tries to use lemmas and hypotheses to simplify the current goal; `dsimp` is like `simp`, using only definitional equalities
`rw`	applies the argument as a rewrite rule to the current goal
`refl`	tries to resolve the goal through definitional equality
`tauto`, `tauto!`, `finish`	finishing tactics, trying to solve the goal completely using basic logic, definitional equalities, etc.
`arith`, `linarith`, `omega`, `ring`, ...	tactics that handle arithmetic equalities and inequalities of various types; particularly convenient in combination with `calc`
`norm_num`, `norm_cast`	normalise expressions in various ways
`.mp`, `.mpr`	extracts implications from left to right (or right to left, respectively) from equivalences
`.symm`	converts equalities `x = y` into `y = x`.

Figure 1.4: Basic Lean tactics and conversions

$\to$	`\to`	function type/implication
$\leftrightarrow$	`\iff`	equivalence (on `Prop`)
$\times$	`\times`	product type
$\forall$	`\forall`	all quantifier
$\forall^f$	`\allf`	modified all quantifier (almost all)
$\exists$	`\exists`	existential quantifier
$\wedge$	`\and`	logical and
$\vee$	`\or`	logical or
$\neg$	`\not`	logical not
$\leq$	`\leq`	less or equal than
$\geq$	`\geq`	greater or equal than
$\in$	`\in`	is element of
$\notin$	`\nin`	is not an element of
$\subseteq$	`\sub`	is subset of
$\subset$	`\subset`	is proper subset of
$\cup$	`\cup`	union
$\cap$	`\cap`	intersection
$\circ$	`\circ`	function composition
$^{-1}$	`\^-1`	inverse
$\sum$	`\sum`	sum operator
$\prod$	`\prod`	product operator
$\uparrow$	`\u`	type coercion
$\mathbb{C}$	`\C`	the type `complex` of complex numbers
$\mathbb{N}$	`\N`	the type `nat` of natural numbers
$\mathbb{R}$	`\R`	the type `real` of real numbers
$\mathbb{Z}$	`\Z`	the type `int` of integers
$\longrightarrow$	`\hom`	type of morphisms
$\mathbb{1}$	`\b1`	identity morphism
$\cong$	`\cong`	type of isomorphisms
$\gg$	`\gg`	forward composition of morphisms
$\Rightarrow$	`\func`	type of functors
$\ggg$	`\ggg`	forward composition of functors
=	=	equality
==	==	heterogeneous equality (not the same as equality!)
Σ	`\Sigma`	dependent sum type
Π	`\Pi`	dependent product type

Figure 1.5: Standard Lean unicode symbols and their VSCode transcriptions

1.E Exercises

Exercise 1.E.1 (generating pen-and-paper formulas)**.** Give pen-and-paper descriptions of the following notions or statements, using logical formulas only (and basic arithmetic structures):

1. Squares of real numbers are non-negative.
2. There is no natural number whose double is 1.
3. The ring of integers has no zero divisors.
4. The equation $x^2 = 2022$ has exactly two solutions in the real numbers.
5. The triangle inequality for the absolute value on real numbers.
6. Convergence of a sequence of real numbers.

Exercise 1.E.2 (understanding pen-and-paper formulas)**.** Let X be a set and let $f, g\colon X \longrightarrow X$ be maps. What do the following formulas express?

1. $\exists_{y\in X} \quad \forall_{x\in X} \quad f(x) = y$
2. $\forall_{x\in X} \quad f\big(g(x)\big) = x$
3. $\forall_{x\in X} \quad g\big(f(x)\big) = x$
4. $\forall_{x\in X} \quad \exists_{x'\in X} \quad f(x') = g(x)$

Exercise 1.E.3 (theorems as pen-and-paper formulas)**.** Give pen-and-paper formulations of the following theorems, using logical formulas only (and basic algebraic structures):

1. Fermat's little theorem
2. Fermat's last theorem

Exercise 1.E.4 (structuring a proof)**.** Pick your favourite textbook in undergraduate mathematics and pick your favourite theorem/proof in this book.

1. Structure the proof into small steps.
2. Which of the steps in the proof are
   - introductions of logical constructors?
   - eliminations of logical constructors?
   - applications of previous results?
   - calculations?

Exercise 1.E.5 (Lean installation)**.** Install Lean and a suitable editor [34].

Exercise 1.E.6 (other proof assistants)**.** Compare the Lean example in Figure 1.1 with its relatives in Coq (Figure 1.6) and Isabelle/HOL (Figure 1.7).

```
Require Import ZArith_base ZArithRing Psatz.
Open Scope Z_scope.

Require Import ssreflect ssrfun ssrbool.

Lemma pow_eq_zero : forall (x : Z),
  x^2 = 0 -> x = 0.
Proof.
  nia.
Qed.

Theorem binomial_solution : forall (x : Z),
  x^2 - 2 * x + 1 = 0 <-> x = 1.
Proof.
  (* Let x : Z *)
  intro x.
  (* We show both implications individually *)

  have one_is_solution : x = 1 -> x^2 - 2 * x + 1 = 0.
       intro x_is_1.
       rewrite x_is_1.
       trivial.

  have solution_is_one : x^2 - 2 * x + 1 = 0 -> x = 1.
       intro x_is_solution.
       (* nia solves this right away,
          but we give a human-readable proof *)
       have binomi : (x-1)^2 = x^2 - 2 * x + 1.
            by ring.
       have xminus1_squared_is_0 : (x-1)^2 = 0.
            by rewrite binomi.
       have xminus1_is_0 : x - 1 = 0.
            by exact : pow_eq_zero (x-1) xminus1_squared_is_0.
       have x_is_xminus1plus1 : x = x - 1 + 1.
            by ring.
       have x_is_1 : x = 1.
            rewrite xminus1_is_0 in x_is_xminus1plus1.
            exact x_is_xminus1plus1.
       exact x_is_1.

  split.
  * exact solution_is_one.
  * exact one_is_solution.
Qed.
```

Figure 1.6: A first Coq proof

```
theory intro_example
  imports Main "HOL-Algebra.Group"
begin

theorem binomial_solution:
  fixes x :: "int"
  shows "x^2 - 2 * x + 1 = 0 \<longleftrightarrow> x = 1"
proof (intro iffI)
  text \<open>We show both implications individually\<close>

  show one_is_solution :
       "x = 1 \<Longrightarrow> x^2 - 2 * x + 1 = 0"
  proof -
    assume x_is_one : "x = 1"
    show "x^2 - 2 * x + 1 = 0"
         using x_is_one by simp
  qed

  show solution_is_one :
       "x^2 - 2 * x + 1 = 0 \<Longrightarrow> x = 1"
  proof -
    assume x_is_solution : "x^2 - 2 * x + 1 = 0"

    have xminus1_squared_is_0 : "(x-1) * (x-1) = 0"
    proof -
      have "(x-1) * (x-1) =  x^2 - 2 * x + 1"
           by (simp add : power2_eq_square algebra_simps)
      also have "\<dots> = 0"
           using x_is_solution by simp
      finally show ?thesis by this
    qed

    have xminus1_is_0 : "x - 1 = 0"
         using xminus1_squared_is_0
               divisors_zero[of "x-1" "x-1"]
         by simp

    show "x = 1"
         using xminus1_is_0 by simp
  qed
qed
end
```

Figure 1.7: A first Isabelle/HOL proof

2

Basic Examples

We first practice basic proof techniques in Lean at the example of injectivity and surjectivity of maps.

We then start interacting with the Lean library mathlib. The mathlib provides a wide range of proof tactics and mathematical libraries to simplify the task of formalising and proving mathematical statements. We illustrate this in the following situations:

- We get acquainted with basic induction proofs, using geometric sums as example. Such proofs are very common.
- As an example of algebraic structures provided by mathlib, we look at commutators in group theory.
- We give two proofs of a vanishing criterion for real numbers.

In each of the examples, we first develop a pen-and-paper version and then formalise it in Lean. Many Lean proofs in current libraries or other Lean code aim at brevity and will only be comprehensible when viewed in a Lean interpreter. We pursue a different path: We make all relevant steps in Lean proofs explicit enough that they can be understood and enjoyed by humans.

Overview of this chapter.

2.1 Injectivity and Surjectivity of Maps

We practice basic proof techniques in Lean by formalising simple examples of properties of maps, such as injectivity, surjectivity, etc. Of course, all these facts are available in the standard libraries (Exercise 2.E.5). In this section, the focus is on learning how to formulate statements and proofs in Lean.

2.1.1 Pen-and-Paper

As a first step, we note down what we want to state and prove in classical pen-and-paper style. As always, a theory consists of definitions, theorems, and examples. Being precise and well-structured in this phase will simplify the formalisation step.

Definition 2.1.1 (injective)**.** Let X and Y be sets and let $f\colon X \longrightarrow Y$ be a map. The map f is called *injective* if

$$\forall_{x,x' \in X} \quad f(x) = f(x') \Longrightarrow x = x'.$$

Definition 2.1.2 (surjective)**.** Let X and Y be sets and let $f\colon X \longrightarrow Y$ be a map. The map f is called *surjective* if

$$\forall_{y \in Y} \quad \exists_{x \in X} \quad f(x) = y.$$

Definition 2.1.3 (bijective)**.** Let X and Y be sets and let $f\colon X \longrightarrow Y$ be a map. The map f is called *bijective* if f is injective and f is surjective.

Proposition 2.1.4. *Let X and Y be sets and let $f\colon X \longrightarrow Y$ be a map. Then:*

1. *If f is bijective, then f is injective.*
2. *If f is bijective, then f is surjective.*
3. *If f is surjective and injective, then f is bijective.*

Proof. Ad 1. Let f be bijective, i.e., f is injective and surjective. In particular, f is injective (elimination property of and-clauses).

Ad 2. Let f be bijective, i.e., f is injective and surjective. In particular, f is surjective (elimination property of and-clauses).

Ad 3. Let f be surjective and injective. Then, f is also injective and surjective (commutativity of the logical operator "and"). Hence, f is bijective (by definition of "bijective"). □

Proposition 2.1.5. *Let X, Y, Z be sets and let $f\colon X \longrightarrow Y$, $g\colon Y \longrightarrow Z$ be maps.*

1. *If $g \circ f$ is injective, then f is injective.*

2. *If $g \circ f$ is surjective, then g is surjective.*

Proof. Ad 1. Let $g \circ f$ be injective. Let $x, x' \in X$ with $f(x) = f(x')$. Then

$$\begin{aligned} g \circ f(x) &= g\big(f(x)\big) && \text{(by definition of } \circ) \\ &= g\big(f(x')\big) && \text{(because } f(x) = f(x')) \\ &= g \circ f(x'). && \text{(by definition of } \circ) \end{aligned}$$

Because $g \circ f$ is injective, it follows that $x = x'$. Hence, f is injective.

Ad 2. Let $g \circ f$ be surjective. Let $z \in Z$. Because $g \circ f$ is surjective, there exists an $x \in X$ with $g \circ f(x) = z$. We consider $y := f(x) \in Y$. Then, we obtain

$$\begin{aligned} g(y) &= g\big(f(x)\big) && \text{(by definition of } y) \\ &= g \circ f(x) && \text{(by definition of } \circ) \\ &= z. && \text{(by the choice of } x) \end{aligned}$$

Hence, g is surjective. □

Corollary 2.1.6. *Let X be a set and let $f\colon X \longrightarrow X$ be a map such that $f \circ f$ is bijective. Then f is bijective.*

Proof. We show that f is injective and surjective:

- The map f is injective, because: As $f \circ f$ is bijective, $f \circ f$ is injective. Applying Proposition 2.1.5 (first part) shows that f is injective.

- The map f is surjective, because: As $f \circ f$ is bijective, $f \circ f$ is surjective. Applying Proposition 2.1.5 (second part) shows that f is surjective.

As f is injective and surjective, we conclude that f is bijective. □

Example 2.1.7. We consider the map

$$\begin{aligned} f\colon \{1,2,3\} &\longrightarrow \{1,2,3\} \\ 1 &\longmapsto 1 \\ 2 &\longmapsto 1 \\ 3 &\longmapsto 2. \end{aligned}$$

Then the map f is *not* injective (because $f(1) = 1 = f(2)$ but $1 \neq 2$) and f is *not* surjective (because 3 is not a value of f).

Example 2.1.8. The map

$$\begin{aligned} g\colon \{1,2\} &\longrightarrow \{1,2\} \\ 1 &\longmapsto 2 \\ 2 &\longmapsto 1 \end{aligned}$$

is bijective: Checking all elements shows that g is both injective and surjective and thus bijective.

2.1.2 Lean

We implement the material from Section 2.1.1 in Lean. The Lean sources for these notes are available as a git repository [47].

Source code 2.1.9. This is `maps.lean` of the git repo [47].

Try out the Lean programs in a local Lean installation [34] or in the Lean web interface [35]! For more complex projects and a more efficient workflow, a local installation is highly recommended [36].

We start with general declarations and imports:

```
import tactic     -- standard proof tactics
open   classical  -- we work in classical logic
```

The definitions of injectivity, surjectivity, and bijectivity are straightforward adaptions of their pen-and-paper counterparts (Definitions 2.1.1–2.1.3). The declarations before `:=` are the hypotheses of the definition. As Lean is based on type theory, we replace sets by Lean types and maps by Lean functions. The actual definition follows after `:=`. For the notions of injectivity, surjectivity, and bijectivity, these are just the corresponding logical formulas. For example, the definition of injectivity then reads as follows:

```
def is_injective
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
 := ∀ x : X, ∀ x' : X,
    f x = f x' → x = x'
```

To simplify the use of `is_injective`, we modify the definition as follows:

```
def is_injective
    {X Y : Type*}
    (f : X → Y)
 := ∀ x : X, ∀ x' : X,
    f x = f x' → x = x'
```

The curly braces around `X Y : Type*` turn `X` and `Y` into implicit arguments of type `Type*`. This means that when using `is_injective` we usually only need to supply the function argument; the types `X` and `Y` will be inferred whenever possible. In case Lean cannot infer these placeholders, we can make the arguments explicit by writing `@is_injective` and supplying all arguments.

Similarly, we define surjectivity and bijectivity:

```
def is_surjective
    {X Y : Type*}
    (f : X → Y)
 := ∀ y : Y,
    ∃ x : X, f x = y

def is_bijective
    {X Y : Type*}
    (f : X → Y)
 := is_injective f ∧ is_surjective f
```

In Lean, in logical formulas, implication is denoted by the function arrow → (Curry–Howard isomorphism!). Equality is denoted by = (and yields a truth/`Prop` value).

Simple inheritance properties As next step, we state and prove basic inheritance properties for injective, surjective, bijective maps (Proposition 2.1.4–Corollary 2.1.6).

The hypotheses are listed before `:` and the claimed conclusion after `:`. The proof follows after `:=`; it is useful to recall that under the Curry–Howard isomorphism proofs correspond to implementations of functions.

The first three lemmas correspond to Proposition 2.1.4. The first two parts (`bij_inj`, `bij_surj`) are proved by extracting the correct parts from the defining ∧-formula; these are elimination steps. These proofs are so simple that we can construct the proof term as a single function application; therefore, we do not switch to the tactic mode enclosed in `begin` and `end`.

```
lemma bij_inj
    {X Y : Type*}
    (f : X → Y)
    (f_bijective: is_bijective f)
  : is_injective f
:= and.elim_left f_bijective

lemma bij_surj
    {X Y : Type*}
    (f : X → Y)
    (f_bijective: is_bijective f)
  : is_surjective f
:= and.elim_right f_bijective
```

The third part is proved by re-assembling the ∧-formula in the correct order; this is an introduction/construction step.

```
lemma surj_inj_bij
    {X Y : Type*}
    (f : X → Y)
    (f_surjective: is_surjective f)
    (f_injective:  is_injective f)
  : is_bijective f
:= and.intro f_injective f_surjective
```

The lemmas `inj_comp_injfirst` and `surj_comp_surjsecond` are translations of Proposition 2.1.5; we took the liberty to shift the first part of the If-statements into the hypotheses. In pen-and-paper proofs, such modifications are usually implicit; in Lean, all of this is explicit (but can be easily converted into each other).

For the Lean proofs, we closely follow the pen-and-paper proofs, using suitable Lean concepts.

```
lemma inj_comp_injfirst
    {X Y Z : Type*}
    (f : X → Y)
    (g : Y → Z)
    (gf_injective : is_injective (g ∘ f))
  : is_injective f
:=
begin
  assume x  : X,
  assume x' : X,
  assume f_xx' : f x = f x',

  have gf_xx' : (g ∘ f) x = (g ∘ f) x', from
    calc (g ∘ f) x = g (f x)    : by simp
                ... = g (f x')   : by simp[f_xx']
                ... = (g ∘ f) x' : by simp,

  show x = x',
       by {apply gf_injective, apply gf_xx'},
end
```

What happens in this proof? In order to show `is_injective f`, we need to establish a double ∀-statement. Such statements can be proved/constructed by showing the corresponding inner statement for every possible candidate; these candidates are introduced by `assume`.

Inside of the double ∀-statement, we need to prove an implication. Such an implication can be proved/constructed by assuming the left-hand side and deriving the right-hand side; this assumption on the left-hand side is introduced by `assume` and is given the name `f_xx'`.

We then introduce an intermediate claim via `have` (with the name `gf_xx'`), which is proved through a calculation, as indicated by `calc`.

Finally, we can `apply` the hypothesis `gf_injective` and the computation `gf_xx'` to conclude that `x = x'` (which is the desired right-hand side of the implication). Because this argument after `by` consists of two tactic steps, we need to enclose it in curly braces. At this point, all goals are resolved and the proof is complete.

Alternatively, one could also prove `inj_comp_injfirst` in a more implicit proof style. While the proof becomes significantly shorter, it also becomes much harder to understand for a human reader without stepping through an interactive Lean environment:

```
begin
  intros _ _ h,
  apply gf_injective,
  simp[h],
end
```

We now turn to the second part of of Proposition 2.1.5:

```
lemma surj_comp_surjsecond
    {X Y Z : Type*}
    (f : X → Y)
    (g : Y → Z)
    (gf_surjective : is_surjective (g ∘ f))
  : is_surjective g
:=
begin
  assume z : Z,
  rcases gf_surjective z with ⟨ x : X, gf_x_z : (g ∘ f) x = z⟩,
  let y : Y := f x,

  use y,
  show g y = z, from
    calc g y = g (f x)   : by simp
         ... = (g ∘ f) x : by simp
         ... = z         : by exact gf_x_z,
end
```

Similarly, in order to prove the inheritance of surjectivity, we construct the desired ∀-statement. Surjectivity of the composition gives us existence of a preimage for the composition. To extract such a preimage from the ∃-statement, we can use `rcases` to eliminate the quantifier and extract a witness (whose defining property is named `gf_x_z`). Through `let`, we introduce a new name `y` for the term `f x` (which will serve as the desired preimage for `x` under `g`). To build the claimed ∃-statement, it suffices to give one suitable example; this is introduced via `use`. Finally, a small calculation finishes the proof by showing that `y` has the correct properties.

Also, the proof of the Lean-counterpart of Corollary 2.1.6 is a direct translation of our pen-and-paper proof:

```
lemma square_bij_bij
    {X : Type*}
    (f : X → X)
    (ff_bijective : is_bijective (f ∘ f))
  : is_bijective f
:=
begin
  have f_injective : is_injective f, from
  begin
    have ff_injective : is_injective (f ∘ f),
         by exact bij_inj (f ∘ f) ff_bijective,
    show _,
         by exact inj_comp_injfirst f f ff_injective,
  end,

  have f_surjective: is_surjective f, from
  begin
    have ff_surjective : is_surjective (f ∘ f),
         by exact bij_surj (f ∘ f) ff_bijective,
    show _,
         by exact surj_comp_surjsecond f f ff_surjective,
  end,

  show is_bijective f,
       by exact and.intro f_injective f_surjective,
end
```

The symbol _ asks Lean to try to infer suitable expressions fitting into these "holes". In particular, `show _`, refers to resolving the active claim. Using such holes can be convenient; however, one should keep in mind that sometimes being more explicit can help the reader to follow the arguments.

Examples Finally, we explain how to formalise Example 2.1.7 and Example 2.1.8 in Lean. At this point, we will deviate slightly from the pen-and-paper examples: In pen-and-paper mathematics, sets of the form $\{1, 2, 3\}$ and functions between such sets are quickly handled; however, implicitly, many statements would require a proof: e.g., in the definition of the map f in Example 2.1.7, it is implicit that all terms on the right-hand side indeed lie in $\{1, 2, 3\}$ and that all points in $\{1, 2, 3\}$ occur exactly once on the left-hand side. All of this can be done in Lean. However, for the purpose and the spirit of the Examples 2.1.7 and 2.1.8 it is much simpler to work with simple sum/enumeration types instead of with sets of natural numbers.

The following type `A` has exactly three values, namely `A_1`, `A_2`, `A_3`. Functions on this type can conveniently be defined by a case distinction. Similarly, also proofs can make use of such case distinctions via `cases`.

```
inductive A : Type
| A_1
| A_2
| A_3

def f
  : A → A
| A.A_1 := A.A_1
| A.A_2 := A.A_1
| A.A_3 := A.A_2
```

As in Example 2.1.7, we show that this map `f` is neither injective nor surjective. We follow the outline from Example 2.1.7; but we need to handle the implicit relocation of the negation in the pen-and-paper version. This is performed by simplification with `not_forall` (from the standard libraries).

```
lemma not_inj_f
    : ¬ is_injective f
:=
begin
  -- idea: f A_1 = f A_2, even though A_1 ≠ A_2
  let x  : A := A.A_1,
  let x' : A := A.A_2,

  have f_xx'_x_neq_x' : f x = f x' ∧ x ≠ x', from
  begin
    have f_xx' : f x  = f x',
         by simp[f],
    have x_neq_x' :  x ≠ x',
         by finish,

    show _, by exact and.intro f_xx' x_neq_x',
  end,

  show ¬ is_injective f, from
  begin
    simp only[is_injective, not_forall],
    use x,
    use x',
    exact f_xx'_x_neq_x',
  end
end
```

```
lemma not_surj_f
    : ¬ is_surjective f
:=
begin
  simp only[is_surjective, not_forall],
  show ∃ y : A, ¬(∃ x : A, f x = y), by
  begin
    -- we show that A_3 does not lie in the image
    use A.A_3,
    have A3_not_in_im : ∀ x : A, ¬ f x = A.A_3, from
    begin
      assume x : A,
      -- we now just consider all three cases
      cases x,
        case A.A_1 : {simp[f]}, -- alternatively: {finish}
        case A.A_2 : {simp[f]},
        case A.A_3 : {simp[f]},
    end,
    show _, by {simp only[not_exists], exact A3_not_in_im}
  end
end
```

Similarly, we can also transform Example 2.1.8 into Lean. We check injectivity and surjectivity by going through all the cases:

```
inductive B
| B_1
| B_2

def g : B → B
| B.B_1 := B.B_2
| B.B_2 := B.B_1

lemma bij_g
    : is_bijective g
:=
begin
  have inj_g : is_injective g, from
  begin
    assume x  : B,
    assume x' : B,
    assume g_xx' : g x = g x',
    cases x,
      case B.B_1 : begin cases x', finish, finish end,
      case B.B_2 : begin cases x', finish, finish end,
  end,
```

```
  have surj_g :  is_surjective g, from
  begin
    assume y : B,
    cases y,
      case B.B_1 : begin use B.B_2, finish end,
      case B.B_2 : begin use B.B_1, finish end,
  end,

  show _, by exact surj_inj_bij g surj_g inj_g
end
```

Alternative proofs for `not_inj_f` and `bij_g` are developed in Exercise 2.E.3 and Exercies 2.E.4.

The source code `maps.lean` also contains an indication of how to do these examples with sets $\{1, 2, 3\}$ and $\{1, 2\}$ of natural numbers instead of via the enumeration types `A` and `B`, respectively. The interplay between sets and types is discussed in Section 3.1.1.

Caveat 2.1.10 (empowering the simplifier)**.** Equational lemmas and theorems (`lemma`, `theorem`) can be given the attribute `@[simp]`, adding these statements to the simplifier. The simplifier can then use these equations, thus possibly leading to shorter proofs. However, two things should be kept in mind:

- The statements should correspond to actual simplifications, i.e., the right-hand side should be "simpler" than the left-hand side.
- Adding statements to the simplifier will shorten proofs, but might obscure the ideas behind arguments as statements are used implicitly.

In general, it is considered good practice to avoid the use of non-terminal `simp`. A `simp` (or `dsimp`) is *non-terminal* if it does not resolve the current goal, but only performs an intermediate step. The problem with non-terminal `simp` is that the set of simplifications available to the simplifier can change in future library versions. In particular, the simplifier might then simplify more than intended, thus breaking proofs. Non-terminal occurences of `simp` should therefore be limited by using `simp only`.

2.2 Induction

The natural numbers are built on the induction principle. Therefore, inductive definitions and inductive proofs play a prominent role in the context of natural numbers.

We will get acquainted with basic inductive definitions and proofs (over the natural numbers), using geometric sums as example. The goal is to give a closed expression for the geometric sums $\sum_{j=0}^{n} 2^j$ with $n \in \mathbb{N}$.

2.2.1 Pen-and-Paper

As a first step, we note down what we want to formulate and prove in classical pen-and-paper style.

Proposition 2.2.1. *Let $n \in \mathbb{N}$. Then*

$$\sum_{j=0}^{n} 2^j = 2^{n+1} - 1.$$

Proof. We prove the claim by induction on n:

- *Base case.* For $n = 0$, we obtain

  $$\sum_{j=0}^{n} 2^j = 2^0 = 1 = 2^{0+1} - 1,$$

  as claimed.

- *Induction hypothesis.* Let $m \in \mathbb{N}$. We assume that the claim is proved for m, i.e., that $\sum_{j=0}^{m} 2^j = 2^{m+1} - 1$.

- *Induction step.* We show that the claim then also holds for $m + 1$. To this end, we calculate

  $$\begin{aligned}
  \sum_{j=0}^{m+1} 2^j &= \sum_{j=0}^{m} 2^j + 2^{m+1} && \text{(by definition of } \textstyle\sum\text{)} \\
  &= 2^{m+1} - 1 + 2^{m+1} && \text{(by the induction hypothesis)} \\
  &= 2 \cdot 2^{m+1} - 1 \\
  &= 2^{m+1+1} - 1, && \text{(by definition of exponentiation)}
  \end{aligned}$$

  as claimed. □

2.2.2 Lean

We implement the material from Section 2.2.1 in Lean.

Source code 2.2.2. This is `induction.lean` of the git repo [47].

We start with general declarations and imports:

```
import tactic     -- standard proof tactics
open   classical  -- we work in classical logic
```

In order to define geometric sums, we first pretend that we don't know anything about the sum operators provided by mathlib (Section 2.2.3). Thus, we first need to define the geometric sum at base 2 up to a given natural number. This is an inductive definition over the natural numbers.

Before we give this inductive definition, we briefly explain how natural numbers appear in Lean: In Lean, the inductive nature of natural numbers is reflected in the (inductive) construction of the datatype `nat`:

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

In other words, the datatype `nat` has two constructors:

- The constructor `zero` (which is a constant of type `nat`) and
- The constructor `succ` (which turns natural numbers into natural numbers).

The Peano axioms require that zero is not the successor of any natural number (this is guaranteed by the property that Lean constructors are injective), that no two different natural numbers can have the same successor (again, this is guaranteed by the injectivity of Lean constructures), and the induction principle that all natural numbers can be reached as iterated successors of zero (this is guaranteed by the property that the Lean declaration above also includes that there is no other way of constructing natural numbers). Moreover, the type of natural numbers can also be denoted by $\mathbb{N}$ in Lean.

We can then define functions with arguments in `nat` by induction over this structure; in the case of the geometric sums at base 2, we thus define:

```
def geometric_sum
  : nat → nat
| 0            := 1
| (nat.succ n) := geometric_sum n + 2^(n+1)
```

The Lean version of Proposition 2.2.1 then reads as follows:

```
lemma geometric_sum_eval
    (n : nat)
  : geometric_sum n = 2^(n+1) - 1
:=
```

As in the pen-and-paper situation, we prove this claim by induction over the `nat`-argument. The induction proof is initialised with the `induction` keyword; moreover, we can name the induction variable (`m`) in the induction step and the induction hypothesis (`ind_hyp`). The base case and induction step are indicated by `case`. This syntax already suggests that Lean induction proofs are much more general than proofs over natural numbers: We can use induction proofs for all inductively defined datatypes (e.g., also for the types `A` and `B` from Chapter 2.1.2).

```
begin
  induction n with m ind_hyp,

  -- base case: 0
  case nat.zero : {simp[geometric_sum]},

  -- induction step: m -> m+1
  case nat.succ :
  begin
    calc geometric_sum (m+1) = geometric_sum m + 2^(m+1)
                             : by simp[geometric_sum]
                         ...  = 2^(m+1) - 1 + 2^(m+1)
                             : by simp[ind_hyp]
                         ...  = 2^(m+1) + 2^(m+1) - 1
                             : by omega
                         ...  = 2 * 2^(m+1) - 1
                             : by ring
                         ...  = 2^(m+2) - 1
                             : by ring,
  end
end
```

Here, we used the `ring` tactic to perform simple calculations in rings and the `omega` tactic for specifics of `nat` arithmetic.

Finally, we note that our definition of geometric sums can also be used by Lean to evaluate this function on given natural numbers, i.e., to actually compute geometric sums at base 2:

```
#eval geometric_sum 0 -- 1
#eval geometric_sum 5 -- 63
```

Thus, in contrast with pen-and-paper mathematics, a formalisation in Lean also can allow us to compute simple cases of definitions etc. to test hypotheses.

2.2.3 Lean with Sums

In Section 2.2.2, we considered geometric sums in Lean through an explicit inductive definition. As finite sums of a variable number of summands are widely used in mathematics, such sums and ways to handle them are provided by the main Lean mathematical library: mathlib [37].

In this section, we first consider a very simple example and then also the geometric sums from Section 2.2.2, using sums as provided by the mathlib library `big_operators`. We continue the file `induction.lean` from Section 2.2.2, adding the following declarations:

```
open finset               -- for the range operator
open_locale big_operators -- to enable ∑ notation
```

We first consider the sum $\sum_{j=0}^{n-1} 1$ with $n \in \mathbb{N}$. Using the $\sum$-notation, this sum can be written as follows:

```
def one_sum
  : nat → nat
:= λ n : nat,
   ∑ (i : nat) in range n, 1
```

The sum notation $\sum$ is provided by the library `big_operators`; the notation `range n` (which corresponds to $\{0, \ldots, n-1\}$) is provided by the library `finset`. Moreover, λ is the constructor for (unnamed) functions; for example λ `x, x` corresponds to the pen-and-paper notation $x \longmapsto x$.

Of course, we have $\sum_{j=0}^{n-1} 1 = n$ for all $n \in \mathbb{N}$. This statement can be formalised and proved as in the case of geometric sums in Section 2.2.2:

```
lemma one_sum_eval
    (n : nat)
  : one_sum n = n
:=
begin
  induction n with m ind_hyp,

  -- base case: 0
  case nat.zero : {simp[one_sum]},

  -- induction step: m -> m+1
  case nat.succ :
  begin
    calc one_sum (m+1) = ∑ (i : nat) in range (m+1), 1
                       : by simp[one_sum]
                   ... = ∑( (i : nat) in range m, 1) + 1
                       : by simp
                   ... = m + 1
                       : by simp[ind_hyp],
  end
end
```

Such proofs are common. Therefore, there is a suitable abstraction available, the lemma `sum_range_induction` (in the library `big_operators.basic`). How can one find out that such a lemma exists? One can either browse the mathlib documentation [37] or one can use the tactic `library_search` that searches mathlib for statements that can resolve the corresponding goal (in a single step). As mathlib does not contain anything on our function `one_sum`, we first have to `unfold` its definition:

```
begin
  unfold one_sum,
  -- found by library_search:
```

```
  by exact sum_range_induction (λ (k : ℕ), 1) (λ (n : ℕ), n)
                               rfl (congr_fun rfl) n,
end
```

It is the objective of Exercise 2.E.8 to figure out what `sum_range_induction` exactly is about and how it is proved in mathlib.

We return to the geometric sums. Using the sum notation, we can handle this example as follows:

```
def geometric_sum'
  : nat → nat
:= λ n,
   ∑ (i : nat) in range n, 2^i

lemma geometric_sum_eval'
    (n : nat)
  : geometric_sum' n =  2^n - 1
:=
begin
  induction n with m ind_hyp,

  -- base case: 0
  case nat.zero : {simp[geometric_sum'], ring},

  -- induction step: m -> m+1
  case nat.succ :
  begin
    begin
    calc geometric_sum' (m+1)
              = ∑ i in range (m+1), 2^i
              : by simp[geometric_sum']
          ... = ∑ i in range m, 2^i + 2^m
              : by exact sum_range_succ (λ x, 2^x) m
          ... = geometric_sum' m + 2^m
              : by simp[geometric_sum']
          ... = 2^m - 1 + 2^m
              : by simp[ind_hyp]
          ... = 2^m + 2^m - 1
              : by omega
          ... = 2^(m+1) - 1
              : by ring_nf,
    end,
end
```

2.3 Commutators

Groups are basic algebraic structures that are used in many ways. The Lean mathlib provides a formalisation of many concepts and statements from basic group theory (in `algebra.group`).

We will experiment with these libraries by considering commutators in groups.

2.3.1 Pen-and-Paper

As a first step, we note down what we want to formulate and prove in classical pen-and-paper style:

Definition 2.3.1 (commutator)**.** Let G be a group, let $g, h \in G$. The *commutator of g and h* is defined as

$$[g,h] := g \cdot h \cdot g^{-1} \cdot h^{-1} \in G.$$

Proposition 2.3.2. *Let G, H be groups, let $f\colon G \longrightarrow H$ be a group homomorphism, and let $g, h \in G$. Then*

$$f\big([g,h]\big) = \big[f(g), f(h)\big].$$

Proof. We compute that

$$\begin{aligned}
f\big([g,h]\big) &= f\big(g \cdot h \cdot g^{-1} \cdot h^{-1}\big) && \text{(definition of the commutator)}\\
&= f(g) \cdot f(h) \cdot f(g^{-1}) \cdot f(h^{-1}) && \text{(as } f \text{ is a group homomorphism)}\\
&= f(g) \cdot f(h) \cdot \big(f(g)\big)^{-1} \cdot \big(f(h)\big)^{-1} && \text{(as } f \text{ is a group homomorphism)}\\
&= \big[f(g), f(h)\big], && \text{(definition of the commutator)}
\end{aligned}$$

as claimed. □

Proposition 2.3.3. *Let G be a group, let $a, b \in G$, and let $A := a^{-1}$, $B := b^{-1}$. Then, we have*

$$[a,b]^3 = [abA, BabA^2] \cdot [Bab, b^2].$$

Proof. This is a straightforward computation: We have

$$\begin{aligned}
[abA, BabA^2] \cdot [Bab, b^2] &= abA \cdot BabA^2 \cdot aBA \cdot a^2BAb \cdot Bab \cdot b^2 \cdot BAb \cdot B^2\\
&= abAB \cdot abAB \cdot Aa^2 \cdot BAbBab \cdot b^2B \cdot A \cdot bB^2\\
&= [a,b] \cdot [a,b] \cdot a \cdot 1 \cdot b \cdot A \cdot B\\
&= [a,b] \cdot [a,b] \cdot [a,b]\\
&= [a,b]^3.
\end{aligned}$$

□

The previous proposition is important in the study of stable commutator length in groups [13].

2.3.2 Lean

We implement the material from Section 2.3.1 in Lean.

Source code 2.3.4. This is `commutator.lean` of the git repo [47].

We start with general declarations and imports; in particular, we import basics on groups from the mathlib library `algebra.group.basic`.

```
import tactic               -- standard proof tactics
import algebra.group.basic -- basic group theory

open classical  -- we work in classical logic
```

Many algebraic structures are represented in Lean by type classes. *Type classes* are record types, parametrised over type arguments. Elements of such record types are called *instances* for the corresponding type arguments. Instances of type classes are inferred and tracked by the type class inference mechanism.

For instance, the type class `semigroup` of semi-groups roughly looks as follows:

```
@[class]
structure semigroup
   (G : Type u)
:= (mul : G → G → G)
   (mul_assoc : ∀ (a b c : G), (a * b) * c = a * b * c)
```

An instance `semigroup G` for a type `G` thus corresponds to defining a composition `mul` on `G` and giving a proof of associativity of `mul`.

Type classes are extensible and hence can be used to build a hierarchy of structures. In fact, in mathlib, also `semigroup` is an extension of another type class. The type class `group` is obtained by a chain of type class extensions.

Definition 2.3.1 translates directly to Lean. The expression `group G` denotes an instance of `G` for the type class `group`. In this definition, `[group G]` is also an argument/hypothesis, but the square brackets turn this into an implicit argument; this means that when applying `cmtr`, we do not need to pass a proof that `G` is a group as explicit argument but can leave it to the type class inference mechanism. This unclutters notation.

```
def cmtr
    {G : Type*} [group G]
    (g : G)
    (h : G)
:= g * h * g⁻¹ * h⁻¹
```

Proposition 2.3.2 can be formalised as follows:

```
lemma cmtr_hom
    {G : Type*} [group G]
    {H : Type*} [group H]
    (f : monoid_hom G H) -- f is a group homomorphism
    (g : G)
    (h : G)
  : f (cmtr g h) = cmtr (f g) (f h)
:=
```

Here, `f` is assumed to be a group homomorphism; as `G` and `H` are groups, this amounts to saying that `f` is compatible with multplication, i.e., a monoid homomorphism between the underlying multiplicative monoids.

The proof is a straightforward computation, using from mathlib that `f` is compatible with multiplication (`mul_hom.map_mul`) and compatible with taking inverses (`monoid_hom.map_inv`).

The `congr` tactic iteratively splits goals of the form `f x = f' x'` into two goals `f = f'` and `x = x'`. This is *not* always helpful as it might be too greedy.

```
begin
  calc f (cmtr g h) = f (g * h * g⁻¹ * h⁻¹)
                    : by simp[cmtr]
                ... = f g * f h * f (g⁻¹) * f (h⁻¹)
                    : by simp[mul_hom.map_mul]
                ... = f g * f h * (f g)⁻¹ * (f h)⁻¹
                    : by {congr, simp only[monoid_hom.map_inv],
                                 simp only[monoid_hom.map_inv]}
                ... = cmtr (f g) (f h)
                    : by simp[cmtr],
end
```

Finally, we prove Proposition 2.3.3 on triple powers of commutators: triple powers of commutators can be written as a product of only *two* commutators. To this end, we first note that $g^3 = g \cdot g \cdot g$ holds for every group element g (where $\cdot^3$ is defined by induction ...):

```
lemma pow_three
    {G : Type*} [group G]
    (g : G)
  : g^3 = g * g * g
:=
begin
  group,
end
```

Using this lemma, Lean can basically perform the computation in the proof of Proposition 2.3.3 on its own, using the `group` tactic:

```
lemma cmtr_pow_three
    {G : Type*} [group G]
    (a : G) {A : G} {A_def : A = a⁻¹}
    (b : G) {B : G} {B_def : B = b⁻¹}
  : (cmtr a b)^3
    = cmtr (a*b*A) (B*a*b*A^2) * cmtr (B*a*b) (b^2)
:=
begin
  unfold cmtr,
  by {simp[pow_three,A_def,B_def], group},
end
```

2.4 The Real Zero

We consider a fundamental criterion for a real number to be zero, based on the Archimedean property. Real numbers in the Lean mathlib are actual real numbers, constructed via the Cauchy completion of the rationals, and thus satisfy the usual axioms of the reals. Such real numbers are *not* to be confused with floating point numbers as provided by many programming languages.

2.4.1 Pen-and-Paper

Proposition 2.4.1 (a criterion for being 0)**.** *Let $x, c \in \mathbb{R}$ and suppose that*

$$\forall_{n \in \mathbb{N}_{>0}} \quad |x| \leq \frac{c}{n}.$$

Then $x = 0$.

Proof. Let $y := |x|$. Because of the definiteness of the absolute value on $\mathbb{R}$, it suffices to show that $y = 0$.

On the one hand, by definition, $y = |x| \geq 0$.

On the other hand, *assume* for a contradiction that $y > 0$. Because $\mathbb{R}$ is Archimedean, there exists an $m \in \mathbb{N}$ with $m > c/y$. Let $n := m + 1$. Then $n > 0$ and we have

$$\begin{aligned}
y &= \frac{1}{n} \cdot n \cdot y && \text{(because } n > 0\text{)} \\
&> \frac{1}{n} \cdot m \cdot y && \text{(because } 1/n > 0 \text{ and } y > 0 \text{ and } m > n\text{)} \\
&\geq \frac{1}{n} \cdot \frac{c}{y} \cdot y && \text{(by the choice of } m\text{)} \\
&\geq y, && \text{(by the hypothesis on } y = |x|\text{)}
\end{aligned}$$

and thus $y > y$. This contradiction shows that $y \leq 0$.

In summary, we have $y = 0$. □

2.4.2 Lean

We implement the material from Section 2.4.1 in Lean.

Source code 2.4.2. This is `zero.lean` of the git repo [47].

The real numbers are provided in `data.real.basic`.

```
import tactic -- standard proof tactics
import data.real.basic
import analysis.specific_limits -- for the proof via limits
import order.filter.basic       -- dito

open classical -- we work in classical logic
```

The translation of Proposition 2.4.1 and its proof into Lean is mostly straightforward. For the formulation of the statement, we need to keep in mind that the natural numbers "n" implicitly are used as real numbers. In Lean, we thus need to cast from the type `nat` to `real`; this coercion is already pre-defined (along with its basic properties) and can be invoked by adding the explicit type signature `n : real` or by writing `↑n`. The coercion framework is provided by the core library `init.has_coe`.

```
lemma zero_via_1_over_n
    (x : real)
    (c : real)
    (x_le_c_over_n : ∀ n : nat, n > 0
                                → abs x ≤ c / (n : real))
  : x = 0
:=
```

Proofs by contradiction are initiated by `by_contradiction`. This requires classical logic. We only slightly reformulate the computation to simplify matters for the arithmetic tactics of Lean. The basics `abs_nonneg` etc. on reals are provided by mathlib.

```
begin
  let y : real := abs x,

  have y_is_0 : y = 0, from
  begin
    have y_geq_0 : y ≥ 0,
         by apply abs_nonneg,
```

```
    have y_leq_0 : y ≤ 0, from
    begin
      by_contradiction,
      have y_pos : y > 0,
           by linarith,

      -- we use that ℝ is archimedean
      have ex_m_big : ∃ m : nat, ↑m > c/y,
           by exact exists_nat_gt (c / y),
      rcases ex_m_big with ⟨ m : nat, m_gt_cy ⟩,
      -- we enforce positivity by adding 1
      let n : nat := m + 1,
      have n_pos : n > 0,
           by exact nat.succ_pos m,
      have n_big : ↑n > c/y, by
           calc ↑n > ↑m  : by simp
                ... > c/y : by exact m_gt_cy,

      -- using n, we show that n * y < n * y
      -- (which is the desired contradiction)
      have : ↑n * y > ↑n * y, by
           calc ↑n * y > (c/y) * y
                       : by exact (mul_lt_mul_right y_pos).mpr
                                  n_big
                ... ≥ c
                       : by finish
                ... ≥ ↑n * y
                       : by {apply (le_div_iff' _).mp
                                   (x_le_c_over_n n n_pos),
                             exact nat.cast_pos.mpr n_pos},

      show false,
           by linarith,
    end,

    show y = 0,
         by exact le_antisymm y_leq_0 y_geq_0, -- or: finish
  end,

  show x = 0,
    by {dsimp only[y] at y_is_0,
        exact abs_eq_zero.mp y_is_0},
end
```

The suffixes `.mp` and `.mpr` extract the implications from left to right and from right left, respectively, from equivalences.

A much weaker version of the same criterion Let us consider the following version of the vanishing criterion from Proposition 2.4.1:

```
lemma zero_via_1_over_n'
    (x : real)
    (c : real)
    (x_le_c_over_n : ∀ n : nat, abs x ≤ c / (n : real))
  : x = 0
:=
```

The only difference with `lemma zero_via_1_over_n` above is that we require the estimate for *all* natural numbers and not only for the positive ones. This statement can be proved by the same strategy as before.

```
begin
  let y : real := abs x,

  have y_is_0 : y = 0, from
  begin
    have y_geq_0 : y ≥ 0,
         by apply abs_nonneg,

    have y_leq_0 : y ≤ 0, by
         calc y = |x|   : by refl
            ... ≤ c/0   : by exact x_le_c_over_n 0
            ... = 0     : by ring, -- !!

    show y = 0,
         by exact le_antisymm y_leq_0 y_geq_0,
  end,

  show x = 0,
    by {dsimp only[y] at y_is_0,
        exact abs_eq_zero.mp y_is_0},
end
```

However, the proof is now suspiciously simple: The key estimate (showing that the absolute value is non-positive) is shorter and neither uses the Archimedean property nor an argument by contradiction.

What happened? Looking at the proof, we use that `c/0 = 0`. This convention on arithmetic operations in Lean is *not* what we expect from the usual conventions on division by 0 (if allowed at all). Thus, the hypothesis on `abs x` in fact already contains the assumption that `abs x ≤ 0`. Therefore, the statement is trivially true and not particularly useful.

In conclusion, we see that it is important to watch out for corner cases, especially in arithmetic operations. It is good practice to test out statements on examples or in proofs of more involved statements.

2.4.3 Lean with Limits

Alternatively, instead of using the Archimedean property directly, we can also prove the criterion from Proposition 2.4.1 through limits.

Source code 2.4.3. This is `zero.lean` of the git repo [47].

```
lemma zero_via_1_over_n_usinglimits
    (x : real)
    (c : real)
    (x_le_c_over_n : ∀ n : nat, n > 0
                                → abs x ≤ c / (n : real))
  : x = 0
:=
```

The proof follows the same overall strategy as in Section 2.4.2.

```
begin
  let y : real := abs x,

  have y_is_0 : y = 0, from
  begin
    have y_geq_0 : y ≥ 0,
         by apply abs_nonneg,
```

It remains to show $y \leq 0$. The idea is to use that $\lim_{n\to\infty} c/n = 0$ holds for all real numbers c (which in turn is a consequence of the Archimedean property of the reals) and the fact that limits respect the ordering: If $(a_n)_{n\in\mathbb{N}}$ is a convergent sequence of real numbers and y is a real number that satisfies

$$\text{for almost all } n \in \mathbb{N}, \text{ we have } y \leq a_n,$$

then $y \leq \lim_{n\to\infty} a_n$. In combination, this will yield $|x| \leq 0$.

Limits of real numbers in Lean are an instance of a rather general Bourbaki-style concept: limits along filters [11]. More precisely, `filter.tendsto` takes three arguments:

- The function, whose convergence is considered; in our case, this is the function $n \longmapsto c/n$.
- The underlying filter with respect to which convergence is considered; in our case, this is the filter `filter.at_top` consisting of all cofinite subsets of $\mathbb{N}$.

- The filter that describes the "limit" of the convergence; in our case, this is the filter `nhds 0` of all open neighbourhoods around 0.

Thus, `filter.tendsto (λ n : nat, c/↑n) filter.at_top (nhds 0)` expresses that $\lim_{n\to\infty} c/n = 0$. If one works a lot with limits, it is helpful to establish shorthand notation for such limits.

```
  have y_leq_0 : y ≤ 0, from
  begin
    have lim_c_over_n_eq_0 :
         filter.tendsto (λ n : nat, c/↑n)
                        filter.at_top (nhds 0),
         by exact tendsto_const_div_at_top_nhds_0_nat c,
```

We feed this limit into the compatiblity with the ordering of reals. We first establish that $y \leq c/n$ holds for almost all $n \in \mathbb{N}$, namely for all $n \in \mathbb{N}_{\geq 1}$. We can then apply `ge_of_tendsto`.

```
      have y_leq_c_over_almost_all_n :
           ∀ᶠ n : nat in filter.at_top, y ≤ c/n, from
      begin
        have y_leq_c_over_ngeq1:
             ∃ a : nat, ∀ n : nat, n ≥ a → y ≤ c / n,
             by {use 1, apply x_le_c_over_n},
        exact filter.eventually_at_top.mpr y_leq_c_over_ngeq1,
      end,

      show _,
           by exact ge_of_tendsto lim_c_over_n_eq_0
                                  y_leq_c_over_almost_all_n,
  end,

  show y = 0,
       by exact le_antisymm y_leq_0 y_geq_0,
end,

show x = 0,
  by {dsimp only[y] at y_is_0,
      exact abs_eq_zero.mp y_is_0},
end
```

This completes the formalisation of a second proof of Proposition 2.4.1.

2.E Exercises

Exercise 2.E.1 (the identity map)**.** Show that the identity map is bijective.

1. Give a pen-and-paper proof of this statement.
2. Formalise this statement and its proof in Lean. The identity map (on the type `X`) is given by:

```
def id_map
    (X : Type*)
  : X → X
:= λ x, x
```

Source files [47]: `maps_exercise.lean`, `maps_solution.lean`

Exercise 2.E.2 (compositions of injective maps)**.** Show the following statement: The composition of injective maps is injective.

1. Give a pen-and-paper proof of this statement.
2. Formalise this statement and its proof in Lean.

Source files [47]: `maps_exercise.lean`, `maps_solution.lean`

Exercise 2.E.3 (formalising Example 2.1.7)**.** Let X and Y be sets and let $f\colon X \longrightarrow Y$ be a map with the following property: There exist $x, x' \in X$ with $x \neq x'$ and $f(x) = f(x')$. Show that then f is not injective.

1. Give a pen-and-paper proof of this statement.
2. Formalise this statement and its proof in Lean.
3. Use this to give a proof of `lemma not_inj_f`.

Source files [47]: `maps_exercise.lean`, `maps_solution.lean`

Exercise 2.E.4 (formalising Example 2.1.8)**.**

1. Show that the map g from Example 2.1.8 satisfies $g \circ g = \mathrm{id}_{\{1,2\}}$. How can this be used to show that g is bijective?
2. Formalise this argument in Lean to give an alternative proof of `lemma bij_g`.

Source files [47]: `maps_exercise.lean`, `maps_solution.lean`

Exercise 2.E.5 (injectivity/surjectivity in the core libraries)**.**

1. Where in the Lean core libraries are injectivity and surjectivity treated?
2. Which statements on injectivity/surjectivity are provided?
3. How human-readable are the proofs?

Exercise 2.E.6 (the sum of the first natural numbers)**.**

1. Define a Lean function `first_nat_sum` that formalises the map
$$\begin{aligned} s\colon \mathbb{N} &\longmapsto \mathbb{N} \\ n &\longmapsto 2 \cdot \sum_{j=0}^{n} j. \end{aligned}$$
2. Give a pen-and-paper proof that $s(n) = n \cdot (n+1)$ for all $n \in \mathbb{N}$.
3. Formalise this statement/proof in Lean.

Hints. It might be easier *not* to use the $\sum$-functionality.
Source files [47]: `induction_exercise.lean`, `induction_solution.lean`

Exercise 2.E.7 (powers in groups)**.** Let G be a group, let $a, b \in G$, and $n \in \mathbb{N}$.

1. Pen-and-paper: Prove that $(a \cdot b \cdot a^{-1})^n = a \cdot b^n \cdot a^{-1}$.
2. Formalise this statement/proof in Lean.
3. Pen-and-paper: Prove that $b^n \cdot a = a \cdot b^n$ if $a \cdot b = b \cdot a$.
4. Formalise this statement/proof in Lean.

Hints. Lean and its tactics can be pedantic about associativity in groups. When in doubt, add extra steps that spell out such transformations.
Source files [47]: `induction_exercise.lean`, `induction_solution.lean`

Exercise 2.E.8 (general sums and inductive computation)**.** We consider the lemma `sum_range_induction` from `algebra.big_operators.basic`.

1. Pen-and-paper: What does this lemma say?
2. Pen-and-paper: How would you prove this lemma?
3. Pen-and-paper: How could you use it to show $\sum_{j=0}^{n-1} 1 = n$ for all $n \in \mathbb{N}$?
4. How is `sum_range_induction` proved in the Lean library?

Exercise 2.E.9 (cyclic groups)**.**

1. Recall a pen-and-paper definition of *cyclic groups*.
2. Find a definition of cyclic groups in mathlib.
3. Translate this Lean-definition into a pen-and-paper definition.
4. Compare these two definitions!
5. Which statements on cyclic groups are proved in the corresponding Lean library?

Exercise 2.E.10 (an estimate)**.** Let $x, y \in \mathbb{R}$.

1. Pen-and-paper: Prove that $x^2 + 2 \cdot x + y^2 - 2 \cdot y + 2024 \geq 2022$.
2. Formalise this statement/proof in Lean.

Source files [47]: `squares_exercise.lean`, `squares_solution.lean`

Exercise 2.E.11 (two points in a square)**.** Let $a \in \mathbb{R}_{\geq 0}$. If x and y lie in the square $[0, a]^2$ of side length a, then the Euclidean distance between x and y is at most $\sqrt{2} \cdot a$.

1. Pen-and-paper: Prove this statement.
2. Formalise this statement/proof in Lean.

Hints. It might help to first consider the case of intervals instead of squares.
Source files [47]: `squares_exercise.lean`, `squares_solution.lean`

Exercise 2.E.12 (limits and sums)**.** We consider the formula

$$\text{“} \lim_{n \to \infty} (a_n + b_n) = \lim_{n \to \infty} a_n + \lim_{n \to \infty} b_n \text{”}$$

for sequences $(a_n)_{n \in \mathbb{N}}$ and $(b_n)_{n \in \mathbb{N}}$ of real numbers.

1. Pen-and-paper: Under which hypotheses does this formula hold?
2. Find a suitable statement in the mathlib to prove this statement.
3. Pen-and-paper: Give examples for which this formula does *not* hold.

3

Design Choices

Formalising mathematical concepts involves structures and properties and usually these components are intertwined. When modelling such concepts in a proof assistant one is faced with a number of design decisions.

Some of these design decisions also appear in pen-and-paper mathematics. However, in a proof assistant these choices become more apparent and more pronounced.

Typical design options are, for example, the choice between types and sets, the trade-off between structures and properties, the description/construction of examples, and the question of how explicit and evaluable the setup should be.

We illustrate a basic design process at the example of simplicial complexes.

Overview of this chapter.

3.1 Recurring Design Options

3.1.1 Types and Sets

Traditionally, mathematical objects are constructed in terms of set theory. The key property of sets is extensional equality; i.e., two sets are equal if and only if they contain the same elements. For example, a group is a tuple, consisting of a set (the carrier) and maps and properties on this set. However, for groups, equality of carrier sets is not a relevant concept; a more meaningful way to compare groups is to compare the full structure of groups and pass to the notion of isomorphism.

Lean is based on dependent type theory. On top of types, Lean offers typed sets, including basic notation and operations on them. Such typed sets over a base type `A` are modelled as functions `A → Prop`; i.e., sets are described through selection predicates on the base type, which define membership in the given set. Lean sets satisfy extensional equality (`set.ext` in `data.set.basic`). Elements of Lean sets can be viewed as pairs, consisting of a member of the base type and a proof that this member satisfies the defining selection predicate of the set. In particular, handling elements of sets comes at the price of also handling these membership proofs.

Therefore, it is more straightforward and more elegant in Lean to formalise notions in terms of types instead of sets. Sets should only be used in those situations in which the set properties are essential.

For example, when formalising the notion of a group, it is beneficial to formalise the carrier of the group as a type instead of as a set. An example of a formalisation of a mathematical concept that builds on Lean sets is given in Section 3.2.

3.1.2 Structures and Properties

In pen-and-paper mathematics, it is common to extract distinguished maps or elements out of defining existence properties. For example, many versions of the definition of groups contain the axiom that for every element there exists an inverse element. Usually, this is followed by a proof of uniqueness of inverses (using a neutral element and associativity); subsequently, one defines the map $\cdot^{-1}$ from the carrier of the group to itself that maps each element to its unique inverse. Alternatively, one could make the inversion map part of the structure of a group and formulate the axiom of inverses in terms of this map. Similarly, and even more fundamentally, there is the choice of whether

the neutral element of the group is part of the structure or whether only its existence is required by the axioms.

In pen-and-paper mathematics, usually, such choices have no actual consequences as it is straightforward to translate between them. Strictly speaking, these choices are relevant and in a more constructive setting, such choices might have an effect. Implicit transitions between structures and properties are difficult in proof assistants. Moreover, in most situations, structures can be handled with less overhead than properties. Therefore, in general, preference should be given to structures.

Many mathematical concepts are organised in hierarchies (e.g., algebraic structures). Modelling hierarchies appropriately can make the formalisation slicker and more robust.

3.1.3 Restrictive Types and Cutting Corners

The dependent type theory underlying Lean allows us to express restricted domains of operations. On the one hand, this gives fine-grained control and a faithful translation of usual conventions. On the other hand, when applying such operations, such restricted domains lead to proof obligations. In contrast, keeping unrestricted domains leads to simple function types, but shifts the proof obligations to, e.g., theorems on properties of these functions.

For example, we would expect the type of division on the real numbers `real` to be `Π (x : real), Π (y : real), y ≠ 0 → real`. But, in fact, division on the field `real` in mathlib has the type `real→ real → real` and uses the convention that division by `0` always leads to `0` (which differs from the usual mathematical conventions!). Cutting corners in this way can lead to misunderstandings, e.g., as in `zero_via_1_over_n'` on p. 41, or worse. We will therefore use restricted domains whenever necessary and will refrain from introducing non-standard default cases.

3.1.4 Constructing Examples and Evaluation

Testing and applying theories in concrete examples is a core discipline in mathematics. On the one hand, one purpose of theories is to solve concrete problems. On the other hand, concrete examples form a basic test whether the theory is sound. During formalisation, examples also can give evidence as to whether the formalised theory reflects its pen-and-paper counterpart well enough. Ideally, formalising examples in a proof assistant also leads to executable code and thus provides the opportunity to run experiments.

A point-free style as promoted by the language of category theory simplifies the formalisation of concepts, theorems, and proofs. In contrast, handling concrete examples in such a framework can become difficult as the burden of proof is shifted to the construction of objects. Part of the formalisation

process should be finding a good balance between slick formalisation and accessibility of examples.

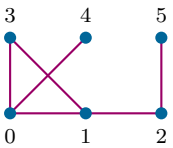
3.2 Simplicial Complexes

We illustrate a basic design process at the example of simplicial complexes. Simplicial complexes are a higher-dimensional version of graphs [55]. The combinatorial flavour of simplicial complexes makes the theory well-suited for applications of algebraic topology to other fields [21, 24, 29].

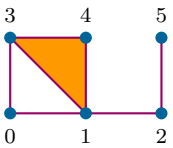
We will focus on design choices and thus stick with a very basic setup. Mathematically, our goal is to formalise the notion of simplicial complexes, simplicial maps, and the Euler characteristic. And examples. In addition, we will aim at a framework that allows for Lean computations in examples.

3.2.1 Pen-and-Paper

We begin with an informal description of the concept of simplicial complexes (Figure 3.1). We first take a step back and quickly recall graphs: A graph consists of vertices and edges between vertices. In the simple, loop-free, unoriented setting, edges can be modelled as two-element sets of vertices.



Edges:
$\{0,1\}$, $\{0,3\}$, $\{0,4\}$,
$\{1,2\}$, $\{1,3\}$,
$\{2,5\}$



Simplices:
$\emptyset$
$\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{5\}$
$\{0,1\}$, $\{0,3\}$, $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{2,5\}$, $\{3,4\}$
$\{1,3,4\}$

Figure 3.1: A graph and a simplicial complex, schematically

Similarly, a simplicial complex is a set of simplices. These simplices are combinatorial versions of vertices, edges, triangles, tetrahedra, ..., i.e., of the standard simplices

$$\Delta^n := \text{convex hull of } \{e_0, \dots, e_n\} \subset \mathbb{R}^{n+1}.$$

The combinatorics of a simplicial complex is mainly driven by knowing which vertices span a common simplex. Therefore, simplices are modelled by finite sets, which should be thought of as the set of vertices of the corresponding simplex. As faces of geometric simplices geometrically also are simplices, each subset of such a finite set should also appear as a simplex of the simplicial complex.

A minimalistic definition of simplicial complexes is thus:

Definition 3.2.1 (simplicial complex)**.** A *simplicial complex* is a set X of finite sets that is closed under taking subsets:

$$\forall_{\sigma \in X} \quad \forall_{\tau} \quad \tau \subset \sigma \Longrightarrow \tau \in X.$$

The elements of X are called *simplices of* X.

Definition 3.2.2 (the set of vertices)**.** Let X be a simplicial complex. The *set of vertices of* X is defined as $\bigcup X = \{x \mid \exists_{\sigma \in X} x \in \sigma\}$. The elements of $\bigcup X$ are called *vertices of* X.

At this point, we already made a design choice: The set of "vertices" of the simplicial complex is left implicit in the definition. In the literature, several versions of the notion of (abstract) simplicial complexes are in use. For example, some authors define simplicial complexes as a pair (V, S), consisting of a set V and a set S of finite subsets of V that is closed under taking subsets. In addition, it is sometimes required that S contains all singletons $\{x\}$ with $x \in V$.

Moreover, another choice involves the role of the empty set: Some authors do not require closure under taking all subsets, but only under taking all non-empty subsets.

Example 3.2.3 (empty)**.** The empty set is a simplicial complex, the *empty simplicial complex*.

Example 3.2.4 (standard simplex)**.** If V is a finite set, then the power set $P(V)$ of V is a simplicial complex, the *simplex spanned by* V.

For $n \in \mathbb{N}$, we call $\Delta(n) := P(\{0, \ldots, n\})$ the *standard* n*-simplex*. This simplicial complex $\Delta(n)$ can be viewed as a combinatorial model of the affine simplex Δ^n.

Example 3.2.5 (the line)**.** The set

$$\big\{\{n, n+1\} \;\big|\; n \in \mathbb{Z}\big\} \cup \big\{\{n\} \;\big|\; n \in \mathbb{Z}\big\} \cup \{\emptyset\}$$

is a simplicial complex. It can be viewed as a combinatorial model of the real line.

Further examples will be considered in Section 3.5 and Section 3.6.

Affine simplices in Euclidean spaces have a canonical notion of dimension. The dimension can be expressed in terms of the number of (affinely independent) vertices. This leads to the notion of dimension of simplicial complexes.

Definition 3.2.6 (dimension of a simplex)**.** Let X be a simplicial complex and let $\sigma \in X$. The *dimension of* σ is defined as

$$\dim \sigma := |\sigma| - 1 \in \mathbb{N} \cup \{-1\}.$$

Here, $|\sigma|$ denotes the cardinality of σ.

Definition 3.2.7 (dimension of a simplicial complex)**.** Let X be a simplicial complex and let $n \in \mathbb{N}$. The simplicial complex X *has dimension* n if the following conditions are satisfied:

- for all $\sigma \in X$, we have $\dim \sigma \leq n$; and
- there exists a $\sigma \in X$ with $\dim \sigma \geq n$.

Proposition 3.2.8 (dimension of the standard simplex)**.** *Let $n \in \mathbb{N}$. Then, $\Delta(n)$ has dimension n.*

Proof. We first prove the upper bound: Let $\sigma \in \Delta(n) = P(\{0, \ldots, n\})$; i.e., σ is a subset of $\{0, \ldots, n\}$. Therefore, we obtain

$$\dim \sigma = |\sigma| - 1 \leq \big|\{0, \ldots, n\}\big| - 1 = n + 1 - 1 = n.$$

We now show the lower bound: The set $\sigma := \{0, \ldots, n\}$ is a simplex of $\Delta(n)$ and satisfies

$$\dim \sigma = \big|\{0, \ldots, n\}\big| - 1 = n + 1 - 1 \geq n.$$

Therefore, $\Delta(n)$ has dimension n. □

3.2.2 Lean

We implement the material on simplicial complexes from Section 3.2.1.

Source code 3.2.9. This is `simplicial_complex.lean` of the git repo [47].

When formalising the notion of simplicial complex from Definition 3.2.1, we need to decide how to model sets (of sets), the finiteness property, and the closure property.

In the definition of simplicial complexes, extensionality of sets plays an important role, both for the overall set of simplices and for the individual simplices. Therefore, it is natural to formalise simplicial complexes as sets of sets. Because sets in Lean are typed sets, we need to specify a base type. This step is an example of a types-and-sets choice (Section 3.1.1).

For the finiteness property, there are two canonical options: On the one hand, we could formalise a simplicial complex as a set S of sets such that the following holds: For each $\sigma \in S$, the set σ is finite. On the other hand, we could formalise a simplicial complex as a set of finite-sets, where "finite-sets" are a type of sets (over the given base type) with a built-in finiteness

guarantee. We will choose the latter option, using Lean's `finset` types. This step is an example of a structures-and-properties choice (Section 3.1.2).

```
import tactic          -- standard proof tactics
import data.set        -- basics on sets
import data.set.finite -- basics on finite sets
import data.finset     -- type-level finite sets

open classical  -- we work in classical logic
```

The closure property can be formalised in a straightforward manner. Anticipating that we will need to reason with and about this closure property in a modular way, we give it a separate definition:

```
@[simp]
def is_subset_closed
    {a : Type*}
    (S : set (finset a))
:= ∀ s ∈ S, ∀ t, t ⊆ s → t ∈ S
```

In order to unclutter notation later on, we make the base type argument `a` in `is_subset_closed` implicit.

Simplicial complexes are then formalised as records, consisting of two fields: a set of `finset` over a base type and (a proof of) the closure property:

```
structure simplicial_complex (a : Type*)
:= mk :: (simplices : set (finset a))
         (subset_closed : is_subset_closed simplices)
```

In this setting, we also formalise the set of vertices:

```
def vertices
    {a : Type*}
    (X : simplicial_complex a)
  : set a
:= { x : a | ∃ s ∈ X.simplices, x ∈ s }
```

Basic examples In order to formalise the empty simplicial complex and the standard simplex, we need to provide proofs of `is_subset_closed` in the corresponding situations. As these situations are simple enough, basic Lean tactics can provide such proofs.

```
def empty_sc
    {a : Type*}
  : (simplicial_complex a)
:= simplicial_complex.mk
     (∅ : finset (finset a))
     (by tauto)
```

```
@[simp]
def simplex
    {a : Type*}
    (V : finset a)
  : (simplicial_complex a)
:= simplicial_complex.mk
    (finset.powerset V)
    (by {unfold_coes, simp, tauto})

@[simp]
def std_simplex
    (n : ℕ)
  : (simplicial_complex ℕ)
:= simplex (finset.range (n+1))
```

The "real line" from Example 3.2.5 is formalised in Exercise 3.E.5.

Dimension Finally, we formalise the notion of dimension of simplices and simplicial complexes.

```
@[simp]
def dim
    {a : Type*}
    (s : finset a)
  : int
:= finset.card s - 1
```

We can evaluate `dim` on simplices:

```
#eval dim (finset.range 5) -- 4
#eval dim (finset.range 0) -- -1
```

Definition 3.2.7 is already stated in a formalisation-friendly way. It might have seemed more natural to define the dimension of a simplicial complex as the "maximum" of dimensions of all simplices in the given simplicial complex. However, such a maximum might not exist (because the set of simplices could be empty or contain simplices of arbitrarily large dimension). In pen-and-paper mathematics, we usually ignore this problem and handle it implicitly. In a formalisation, it is safer to keep such corner cases explicit. Therefore, we chose the formulation in Definition 3.2.7 that only defines when a simplicial complex has a given number as dimension.

```
@[simp]
def has_dimension_leq
    {a : Type*}
    (X : simplicial_complex a)
    (n : ℕ)
:= ∀ s ∈ X.simplices, dim s ≤ n
```

```
@[simp]
def has_dimension_geq
    {a : Type*}
    (X : simplicial_complex a)
    (n : ℕ)
:= ∃ s ∈ X.simplices, dim s ≥ n

@[simp]
def has_dimension
    {a : Type*}
    (X : simplicial_complex a)
    (n : ℕ)
:= has_dimension_leq X n
 ∧ has_dimension_geq X n
```

Proposition 3.2.8 and its proof are straightforward to formalise:

```
lemma dim_std_simplex
    (n : ℕ)
  : has_dimension (std_simplex n) n
:=
begin
  let Dn := std_simplex n,

  have dim_leq_n : has_dimension_leq Dn n, from
  begin
    assume s : finset nat,
    assume s_in_Dn : s ∈ Dn.simplices,

    have s_sub_n : s ⊆ finset.range (n+1),
         by exact finset.mem_powerset.mp s_in_Dn,

    have card_s_leq_n1 : finset.card s ≤ n + 1, from
    begin
      calc finset.card s ≤ finset.card (finset.range (n+1))
                         : by exact finset.card_le_of_subset
                                    s_sub_n
                     ... ≤ n + 1
                         : by finish,
    end,

    show dim s ≤ n, from
    begin
      calc dim s = finset.card s - 1 : by simp only [dim]
             ... ≤ n + 1 - 1         : by linarith
```

```
            ... ≤ n                    : by linarith,
    end,
  end,

  have dim_geq_n : has_dimension_geq Dn n, from
  begin
    let s := finset.range (n+1), -- {0,...,n}
    use s,

    have s_in_Dn : s ∈ (std_simplex n).simplices,
         by simp,

    have dim_s : dim s ≥ n,
         by finish,

    show _, by exact ⟨s_in_Dn, dim_s⟩,
  end,

  show _, by exact ⟨dim_leq_n, dim_geq_n⟩,
end
```

In a similar way, one can show that the (infinite) simplicial complex from Example 3.2.5 has dimension 1 (Exercise 3.E.5).

3.3 Simplicial Maps

Every theory comes with morphisms between their objects. Simplicial maps are structure-preserving maps between simplicial complexes.

3.3.1 Pen-and-Paper

A simplicial map between simplicial complexes is a map between the sets of vertices that maps simplices to simplices:

Definition 3.3.1 (simplicial map)**.** Let X and Y be simplicial complexes. A *simplicial map* $X \longrightarrow Y$ is a map $f\colon \bigcup X \longrightarrow \bigcup Y$ with

$$\forall_{\sigma \in X} \quad f(\sigma) \in Y.$$

Here, "$f(\sigma)$" denotes the image of the finite set σ under f, i.e., $\{f(x) \mid x \in \sigma\}$.

In this definition of simplicial maps, the dimensions of the simplices are not necessarily preserved: Simplices can be mapped to simplices of smaller dimension (but not to simplices of larger dimension).

Proposition 3.3.2. *Let X and Y be simplicial complex, let $f\colon X \longrightarrow Y$ be a simplicial map, and let $\sigma \in X$. Then*

$$\dim f(\sigma) \leq \dim \sigma.$$

Proof. We have

$$\begin{aligned} \dim f(\sigma) &= \big|f(\sigma)\big| - 1 && \text{(definiiton of dim)} \\ &\leq |\sigma| - 1 && \text{(monotonicity of } |\cdot| \text{ under direct images)} \\ &= \dim \sigma, && \text{(definition of dim)} \end{aligned}$$

as claimed. □

Basic examples for simplicial maps are the identity map and compositions of simplicial maps.

Example 3.3.3 (identity map)**.** Let X be a simplicial complex. Then, clearly, the identity map $\mathrm{id}_{\bigcup X}\colon \bigcup X \longrightarrow \bigcup X$ on the set of vertices of X is a simplicial map. This simplicial map $X \longrightarrow X$ is also denoted by id_X.

Proposition 3.3.4 (composition of simplicial maps)**.** *Let X, Y, Z be simplicial complexes and let $f\colon X \longrightarrow Y$, $g\colon Y \longrightarrow Z$ be simplicial maps. Then the composition*

$$h := g \circ f\colon \bigcup X \longrightarrow \bigcup Z$$

of the underlying maps $f\colon \bigcup X \longrightarrow \bigcup Y$ and $g\colon \bigcup Y \longrightarrow \bigcup Z$ (by abuse of notation denoted by the same letters) is a simplicial map $X \longrightarrow Z$.

Proof. Let $\sigma \in X$. Then

$$h(\sigma) = (g \circ f)(\sigma) = g\big(f(\sigma)\big).$$

As f is simplicial, we have $f(\sigma) \in Y$. Because g is simplicial, we thus obtain $h(\sigma) = g(f(\sigma)) \in Z$. □

In fact, simplicial complexes and simplicial maps form a category (Exercise 3.E.6). In particular, we obtain a notion of simplicial isomorphism. We spell out the definition of isomorphisms explicitly:

Definition 3.3.5 (simplicial isomorphism)**.** Let X and Y be simplicial complexes.

- A simplicial map $f\colon X \longrightarrow Y$ is a *simplicial isomorphism* if there exists a simplicial map $g\colon Y \longrightarrow X$ such that $g \circ f = \mathrm{id}_X$ and $f \circ g = \mathrm{id}_Y$.
- We call X and Y *isomorphic* if there exists a simplicial isomorphism $X \longrightarrow Y$.

Another basic example of simplicial maps are constant maps.

Example 3.3.6 (constant maps)**.** Let X and Y be simplicial complexes and let y be a vertex of Y. Then the constant map

$$\begin{aligned} f\colon \bigcup X &\longrightarrow \bigcup Y \\ x &\longmapsto y \end{aligned}$$

is a simplicial map.

But, wait, aren't we forgetting something? We have to *prove* that these constant maps indeed are simplicial maps. This can be done in the following steps:

- As y is a vertex of Y, the singleton $\{y\}$ is a simplex of Y:

  Because y is a vertex of Y, there exists a simplex $\tau \in Y$ with $y \in \tau$. Therefore, $\{y\} \subset \tau$. As Y closed under taking subsets, also $\{y\} \in Y$.

- If $\sigma \in X$, then $f(\sigma) \subset \{y\}$, by definition of f as constant map at y and direct images.

  Because $\{y\} \in Y$ and because Y is closed under taking subsets, we obtain also $f(\sigma) \in Y$.

  Thus, f is simplicial.

3.3.2 Lean

We implement the material on simplicial maps from Section 3.3.1.

Source code 3.3.7. This is `simplicial_map.lean` of the git repo [47].

We begin with imports, including `simplicial_complex.lean`.

```
import tactic          -- standard proof tactics
import data.set        -- basics on sets
import data.set.finite -- basics on finite sets
import data.finset     -- type-level finite sets
import simplicial_complex -- basics on simplicial complexes

open classical  -- we work in classical logic
```

Simplicial maps are defined as maps between the sets of vertices. Modelling this directly in Lean would be cumbersome. We therefore, take a slightly different approach: A simplicial map is a function between the underlying base types that maps simplices to simplices. Strictly speaking, this is not the same as Definition 3.3.1. However, for all practical purposes, this formalisation is close enough. This is an example for maps of a types-and-sets choice (Section 3.1.1).

As in the case of simplicial complexes, we first define the core property separately and then package the map and a proof that the map satisfies the property into a record:

```
@[simp]
def is_simplicial_map
    {a : Type*} {b : Type*} [decidable_eq b]
    (X : simplicial_complex a)
    (Y : simplicial_complex b)
    (f : a → b)
:= ∀ s, s ∈ X.simplices → finset.image f s ∈ Y.simplices

structure simplicial_map
   {a : Type*} {b : Type*} [decidable_eq b]
   (X : simplicial_complex a)
   (Y : simplicial_complex b)
:= mk :: (map : a → b)
         (is_simplicial : is_simplicial_map X Y map)
```

The image of a `finset` under a map again is a `finset`, provided that the target type has a decidable equality relation. This requirement is necessary in view of the constructive nature of `finset`.

```
structure simplicial_map
   {a : Type*} {b : Type*} [decidable_eq b]
   (X : simplicial_complex a)
   (Y : simplicial_complex b)
:= mk :: (map : a → b)
         (is_simplicial : is_simplicial_map X Y map)
```

Simplicial maps on vertices Before continuing with the material from Section 3.3.1, we perform an intermediate step, which is related to the discrepancy between the original definition and the formalisation: We show that simplicial maps as in `simplicial_map` indeed map vertices to vertices.

As a preparation, we show that singletons of vertices are simplices (as in Example 3.3.6) and that elements of singleton simplices are vertices:

```
lemma vertex_to_singleton
    {a : Type*}
    (X : simplicial_complex a)
    (x : a)
    (x_in_X : x ∈ vertices X)
  : {x} ∈ X.simplices
:=
begin
  -- As x is a vertex, x is contained in a simplex s of X.
  -- Therefore, {x} ⊆ s.
  rcases x_in_X with ⟨ s, ⟨ s_in_SX, x_in_s⟩ ⟩,
  have x_sub_s : {x} ⊆ s,
       by finish,
```

```
  -- As the set of simplices of X is closed under subsets,
  -- also {x} is a simplex of X.
  exact X.subset_closed s s_in_SX {x} x_sub_s,
end

lemma singleton_to_vertex
    {a : Type*}
    (X : simplicial_complex a)
    (x : a)
    (x_in_SX : {x} ∈ X.simplices)
  : x ∈ vertices X
:=
begin
  -- {x} witnesses that x is contained in a simplex
  -- and thus is a vertex
  simp only[vertices, set.mem_def, set.mem_set_of_eq],
  use {x},
  finish,
end

lemma simplicial_map_on_vertices
    {a : Type*} {b : Type*} [decidable_eq b]
    (X : simplicial_complex a)
    (Y : simplicial_complex b)
    (f : simplicial_map X Y)
    (x : a)
    (x_in_X : x ∈ vertices X)
  : f.map x ∈ vertices Y
:=
begin
  simp only[vertices, set.mem_def, set.mem_set_of_eq],

  -- We use t := {f(x)} as a witness.
  let t : finset b := {f.map x},
  use t,

  -- Thus, we need to show that t is a simplex of Y
  have t_in_SY : t ∈ Y.simplices, from
  begin
    have x_in_SX : {x} ∈ X.simplices,
         by exact vertex_to_singleton X x x_in_X,
    have fx_is_t : finset.image f.map {x} = t,
         by finish,
    have fx_in_SY : finset.image f.map {x} ∈ Y.simplices,
         by exact (f.is_simplicial {x} x_in_SX),
```

```
    finish,
  end,

  -- ... and that f(x) ∈ t.
  have y_in_t : f.map x ∈ t,
       by finish,

  show _, by finish,
end
```

Dimension monotonicity We continue as in Section 3.3.1 with the dimension monotonicity of simplicial maps.

```
lemma simplicial_map_dim_mono
    {a : Type*} {b : Type*} [decidable_eq b]
    (X : simplicial_complex a)
    (Y : simplicial_complex b)
    (f : simplicial_map X Y)
    (s : finset a)
    (s_in_SX : s ∈ X.simplices)
  : dim (finset.image f.map s) ≤ dim s
:=
begin
  calc dim (finset.image f.map s)
         = finset.card (finset.image f.map s) - 1
         : by unfold dim
     ... ≤ finset.card s - 1
         : by simp[finset.card_image_le]
     ... ≤ dim s
         : by unfold dim,
end
```

Examples Finally, we formalise the examples from Section 3.3.1 and the notion of simplicial isomorphism in a straightforward way. We begin with the identity map.

```
lemma id_is_simplicial_map
    {a : Type*} [decidable_eq a]
    (X : simplicial_complex a)
  : is_simplicial_map X X (λ x : a, x)
:=
begin
  simp,
end
```

```
def id_simplicial_map
    {a : Type*} [decidable_eq a]
    (X : simplicial_complex a)
  : (simplicial_map X X)
:= simplicial_map.mk
     (λ x : a, x)
     (id_is_simplicial_map X)
```

To show that constant maps are simplicial, we proceed as in Example 3.3.6, making use of `vertex_to_singleton`.

```
lemma const_is_simplicial
      {a : Type*} {b :  Type*} [decidable_eq b]
      (X : simplicial_complex a)
      (Y : simplicial_complex b)
      (y_0 : b)
      (y0_vertex : y_0 ∈ vertices Y)
    : is_simplicial_map X Y (λ x : a, y_0)
:=
begin
  -- As y_0 is a vertex of Y, the set {y_0} is a simplex of Y
  have y0_in_SY : {y_0} ∈ Y.simplices,
       by exact vertex_to_singleton Y y_0 y0_vertex,

  -- Setup for the main argument
  assume s : finset a,
  assume s_in_SX : s ∈ X.simplices,

  let f  := λ x : a, y_0,
  let fs := finset.image f s,

  -- We have f(s) ⊆ {y_0}
  have fs_sub_y0 : fs ⊆ {y_0}, from
  begin
    assume y,
    assume y_in_fs : y ∈ fs,
    finish,
  end,

  -- and thus f(s) is a simplex of Y.
  have fs_in_SY : fs ∈ Y.simplices,
       by exact Y.subset_closed {y_0} y0_in_SY fs fs_sub_y0,

  show _, by finish,
end
```

```
def const_simplicial_map
    {a : Type*} {b :  Type*} [decidable_eq b]
    (X : simplicial_complex a)
    (Y : simplicial_complex b)
    (y_0 : b)
    (y0_vertex : y_0 ∈ vertices Y)
  : (simplicial_map X Y)
:= simplicial_map.mk
     (λ x : a, y_0)
     (const_is_simplicial X Y y_0 y0_vertex)
```

Also, for compositions and isomorphisms, we translate the pen-and-paper versions quite directly:

```
lemma is_simplicial_comp
    {a : Type*} {b : Type*} {c : Type*}
    [decidable_eq b] [decidable_eq c]
    {X : simplicial_complex a}
    {Y : simplicial_complex b}
    {Z : simplicial_complex c}
    (f : a → b)
    (f_simpl : is_simplicial_map X Y f)
    (g : b → c)
    (g_simpl : is_simplicial_map Y Z g)
  : is_simplicial_map X Z (g ∘ f)
:=
begin
  assume s : finset a,
  assume s_in_SX : s ∈ X.simplices,

  let t : finset b := finset.image f s,

  have t_in_SY : t ∈ Y.simplices,
       by {apply (f_simpl s), apply s_in_SX}, -- or: tauto,

  have gfs_in_SZ : finset.image g t ∈ Z.simplices,
       by {apply (g_simpl t), apply t_in_SY},

  show finset.image (g ∘ f) s ∈ Z.simplices,
    by calc finset.image (g ∘ f) s
          = finset.image g (finset.image f s)
          : by exact finset.image_image.symm
      ... = finset.image g t
          : by refl
      ... ∈ Z.simplices
          : by exact gfs_in_SZ,
end
```

```
def simplicial_map.comp
    {a : Type*} {b : Type*} {c : Type*}
    [decidable_eq b] [decidable_eq c]
    {X : simplicial_complex a}
    {Y : simplicial_complex b}
    {Z : simplicial_complex c}
    (f : simplicial_map X Y)
    (g : simplicial_map Y Z)
  : (simplicial_map X Z)
:= simplicial_map.mk
     (g.map ∘ f.map)
     (is_simplicial_comp f.map f.is_simplicial
                         g.map g.is_simplicial)

@[simp]
def is_inverse_simplicial_iso
    {a : Type*} {b : Type*} [decidable_eq a] [decidable_eq b]
    {X : simplicial_complex a}
    {Y : simplicial_complex b}
    (f : simplicial_map X Y)
    (g : simplicial_map Y X)
:= simplicial_map.comp f g = id_simplicial_map X
 ∧ simplicial_map.comp g f = id_simplicial_map Y

@[simp]
def is_simplicial_iso
    {a : Type*} {b : Type*} [decidable_eq a] [decidable_eq b]
    {X : simplicial_complex a}
    {Y : simplicial_complex b}
    (f : simplicial_map X Y)
:= ∃ g : simplicial_map Y X, is_inverse_simplicial_iso f g

-- For example, the identity map is a simplicial isomorphism
-- because it is its own inverse
lemma id_is_simplicial_iso
    {a : Type*} [decidable_eq a]
    (X : simplicial_complex a)
  : is_simplicial_iso (id_simplicial_map X)
:=
begin
  use id_simplicial_map X,
  finish,
end
```

3.4 Finite Simplicial Complexes

So far, we only have a small collection of examples of simplicial complexes. There are two elementary ways to construct more simplicial complexes:

- by specifying a set of simplices and closing up (Section 3.5);
- by combining simplicial complexes (Section 3.6, Exercise 3.E.17).

In principle, all of this could be done for general simplicial complexes. However, in view of later applications and computations, we will focus on the case of *finite* simplicial complexes. Therefore, we will first digress to introduce finite simplicial complexes. This is an example of an examples-and-evaluation choice (Section 3.1.4).

3.4.1 Pen-and-Paper

A simplicial complex is finite if it has only finitely many simplices; this is equivalent to finiteness of the set of vertices.

Definition 3.4.1 (finite simplicial complex)**.** A simplicial complex X is *finite* if the set X is finite.

Proposition 3.4.2 (finiteness of simplicial complexes)**.** *Let X be a simplicial complex. Then the following are equivalent:*

1. *The simplicial complex X is finite in the sense of Definition 3.4.1.*
2. *The set $\bigcup X$ of vertices of X is finite.*

Proof. Let X be finite. We show that $\bigcup X$ is finite: Because each element of X is a finite set, $\bigcup X$ is a finite union of finite sets and thus finite.

Conversely, let $\bigcup X$ be finite. Then X is finite: The set X is a subset of the power set $P(\bigcup X)$. With $\bigcup X$ also $P(\bigcup X)$ is finite. Therefore, the subset X of the finite set $P(\bigcup X)$ is also finite. □

3.4.2 Lean

We implement the material on finite simplicial complexes from Section 3.4.1.

Source code 3.4.3. This is `fin_simplicial_complex.lean` of the git repo [47].

```
import tactic           -- standard proof tactics
import data.set         -- basics on sets
import data.set.finite -- basics on finite sets
import data.finset      -- type-level finite sets
import simplicial_complex -- basics on simplicial complexes

open classical  -- we work in classical logic
```

Finiteness of simplicial complexes We first formalise the definition of finiteness of simplicial complexes and the characterisation in terms of finiteness of the vertex set. In Lean, finiteness of sets can be encoded by the `set.finite` property.

```
def is_finite_simplicial_complex
    {a : Type*}
    (X : simplicial_complex a)
:= set.finite (X.simplices)
```

For the proof of the different characterisations of finiteness of simplicial complexes (Proposition 3.4.2), we follow the pen-and-paper arguments. We have to carefully handle the transition between `set` and `finset`. Therefore, the proofs look bulkier than in the pen-and-paper case.

```
lemma fin_simplices_fin_vertices
    {a : Type*}
    (X : simplicial_complex a)
    (X_fin_S : is_finite_simplicial_complex X)
  : set.finite (vertices X)
:=
begin
  let S := X.simplices,
  let V := vertices X,

  -- Because S is finite,
  -- also V (as finite union of finsets) is finite
  let S' := { (↑s : set a) | s ∈ S },
  let U  := set.sUnion S',

  have U_finite : set.finite U, from
  begin
    -- finite unions of finite sets are finite
    apply set.finite.sUnion,

    -- the set S' is finite because S is finite
    show set.finite S',
         by {simp only[S', bex_def],
             apply set.finite.image (λs, ↑s) (by assumption)},
```

```
    -- all elements of S' are finite sets
    show ∀ (t : set a), t ∈ S' → t.finite, from
    begin
      assume t : set a,
      assume t_in_S' : t ∈ S',
      dsimp only[S'] at t_in_S',
      rcases t_in_S' with ⟨s, ⟨s_in_S, t_is_s⟩⟩,
      have s_finite : set.finite ↑s,
           by exact finset.finite_to_set s,
      induction t_is_s,
      assumption,
    end,
  end,

  have V_sub_U : V ⊆ U, from
  begin
    assume x,
    assume x_in_V : x ∈ V,
    dsimp only[V, vertices] at x_in_V,
    finish,
  end,

  show set.finite V,
       by exact set.finite.subset U_finite V_sub_U,
end

lemma fin_vertices_fin_simplices
    {a : Type*}
    (X : simplicial_complex a)
    (X_fin_V : set.finite (vertices X))
  : is_finite_simplicial_complex X
:=
begin
  -- S is finite,
  -- because S is a subset of the powerset of vertices X,
  -- which is finite (as powerset of a finite set)
  let S : set (finset a) := X.simplices,

  let PV := finset.powerset (set.finite.to_finset X_fin_V),

  have S_sub_PV : S ⊆ PV, from
  begin
    assume s,
    assume s_in_S : s ∈ S,
```

```
    show s ∈ PV, from
    begin
      simp only[finset.mem_powerset],
      assume x,
      assume x_in_s : x ∈ s,

      show x ∈ X_fin_V.to_finset,
           by {apply set.mem_to_finset.mpr, unfold vertices,
               use s, exact ⟨ s_in_S, x_in_s ⟩},
    end,
  end,

  have PV_finite : set.finite ↑PV,
       by exact finset.finite_to_set PV,

  show set.finite S,
       by exact set.finite.subset PV_finite S_sub_PV,
end
```

Finite simplicial complexes However, in terms of computability, we did not yet gain an advantage: The property `set.finite` is not computable and does not give a way to handle finite simplicial complexes in a better way than general ones.

Therefore, we make another types-and-sets choice and introduce a version of simplicial complexes with a built-in finiteness guarantee for the set of simplices: Instead of taking a `set (finset a)` as set of simplices, we restrict ourselves to `finset (finset a)`:

```
structure fin_simplicial_complex (a : Type*)
:= mk :: (simplices : finset (finset a))
         (subset_closed : is_subset_closed
                            (simplices : set (finset a)))
```

In the pen-and-paper version, we can implicitly switch between finite and general simplicial complexes. In Lean, we have to make conversion explicit, because different types are involved.

As every `finset` can be coerced into a `set`, it is straightforward to view a `fin_simplicial_complex` as a `simplicial_complex`.

```
def to_sc
    {a : Type*}
    (X : fin_simplicial_complex a)
  : simplicial_complex a
:= simplicial_complex.mk
     (↑X.simplices)
     (X.subset_closed)
```

In order to enable Lean's automatic coercion through the type class inference mechanism, we also provide a corresponding instance of the type class `has_coe`.

```
instance {a : Type*}
: has_coe (fin_simplicial_complex a) (simplicial_complex a)
:= { coe := to_sc }
```

Moreover, we show that this conversion is compatible with our notion `is_finite_simplicial_complex` of finiteness of simplicial complexes.

```
lemma fin_sc_is_finite
    {a : Type*}
    (X : fin_simplicial_complex a)
  : is_finite_simplicial_complex (↑X : simplicial_complex a)
:=
begin
  exact finset.finite_to_set X.simplices,
end
```

Conversely, finite simplicial complexes can be converted to the ones in the sense of `fin_simplicial_complex`. However, as the conversion from finite sets to `finset` is non-computable, this conversion is not well-suited for computations.

```
noncomputable
def to_fin_sc
    {a : Type*}
    (X : simplicial_complex a)
    (fin_X : is_finite_simplicial_complex X)
  : fin_simplicial_complex a
:=
begin
  let SX : set (finset a) := X.simplices,
  have fin_SX : set.finite SX, by assumption,

  let S : finset (finset a)
        := set.finite.to_finset fin_SX, -- non-computable!

  -- The finset S is closed under taking subsets,
  -- because this holds for the set of simplices of X
  have sub_S : is_subset_closed (↑S : set (finset a)), from
  begin
    assume s : finset a,
    assume s_in_S : s ∈ S,
    have s_in_SX : s ∈ X.simplices, by finish,

    assume t : finset a,
```

```
    assume t_sub_s : t ⊆ s,

    show t ∈ S,
         by simp[X.subset_closed s s_in_SX t t_sub_s],
  end,

  -- We now have all the finiteness/subset guarantees
  -- to build the desired finite simplicial complex:
  exact fin_simplicial_complex.mk S sub_S,
end
```

3.5 Generating Simplicial Complexes

We can generate a simplicial complex by specifying a set of finite sets and then taking the set of all subsets of these sets.

As indicated in Section 3.4, we will restrict to the case of finite simplicial complexes. These can be generated in this way from a finite set of finite sets.

Thus, we can easily specify standard examples of simplicial complexes that model the cylinder, the Möbius strip, the torus, spheres, ...

3.5.1 Pen-and-Paper

Definition 3.5.1 (generated simplicial complex)**.** Let S be a finite set of finite sets. We then write

$$\langle S \rangle := \left\{ \tau \;\middle|\; \exists_{\sigma \in S} \quad \tau \subset \sigma \right\}$$

for the *simplicial complex generated by* S.

Implicit in this definition is the observation that $\langle S \rangle$ indeed is a simplicial complex. Moreover, this simplicial $\langle S \rangle$ complex is finite.

Example 3.5.2 (cylinder)**.** A cylinder can be obtained from a rectangle by identifying one pair of opposite sides (in the same direction). This is illustrated in Figure 3.2; the vertical sides are glued. Subdividing the rectangle into sufficiently many triangles and respecting the glueing condition on the vertical sides, leads to a finite simplicial complex that models the cylinder:

$$\big\langle \{0,1,4\}, \{0,3,4\}, \{1,2,5\}, \{1,4,5\}, \{2,3,0\}, \{2,5,3\} \big\rangle$$

A subdivision into four triangles would not be sufficient: any three vertices can span at most one 2-simplex (extensionality of sets!). Therefore, in order to avoid a "collapse" of the structure, we need to introduce more simplices.

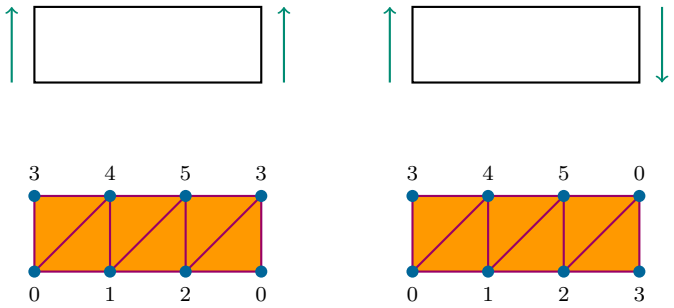


Figure 3.2: A cylinder and a Möbius strip; the outer vertical edges describe the same edge, but the direction of identification differs between the cylinder and the Möbius strip.

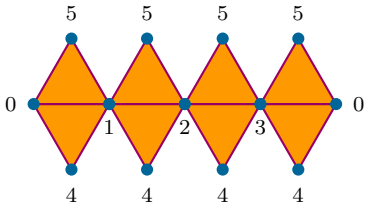


Figure 3.3: An octahedron

Example 3.5.3 (Möbius strip)**.** Similarly, to the cylinder, the Möbius strip is obtained from a rectangle by identifying one pair of opposite sides in the *opposite* direction. This is illustrated in Figure 3.2 and can be modelled by the following finite simplicial complex:

$$\big\langle \{0,1,4\}, \{0,3,4\}, \{1,2,5\}, \{1,4,5\}, \{2,3,0\}, \{2,5,0\} \big\rangle$$

Example 3.5.4 (octahedron)**.** The (hollow) octahedron already comes with an obvious simplicial structure (Figure 3.3) and thus can be modelled by the following finite simplicial complex:

$$\big\langle \{0,1,5\}, \{0,1,4\}, \{1,2,5\}, \{1,2,4\}, \{2,3,5\}, \{2,3,4\}, \{3,0,5\}, \{3,0,4\} \big\rangle$$

Example 3.5.5 (torus)**.** The torus is obtained from a square by glueing opposite sides (in the same direction). This is illustrated in Figure 3.4. Subdividing the square into sufficiently many triangles and respecting the glueing condition on the sides, leads to a finite simplicial complex that models the torus:

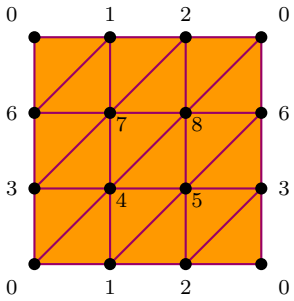


Figure 3.4: A torus

$$\big\langle\{0,1,4\},\{0,3,4\},\{1,2,5\},\{1,4,5\},\{2,0,3\},\{2,5,3\},\\ \{3,4,7\},\{3,6,7\},\{4,5,8\},\{4,7,8\},\{5,3,6\},\{5,8,6\},\\ \{6,7,1\},\{6,0,1\},\{7,8,2\},\{7,1,2\},\{8,6,0\},\{8,2,0\}\big\rangle$$

Example 3.5.6 (Klein bottle)**.** Similarly, the Klein bottle is obtained from a square by glueing opposite sides, one pair in the same direction and one pair in the opposite direction (Figure 3.5). Thus, a finite simplicial complex modelling the Klein bottle is, for instance:

$$\big\langle\{0,1,4\},\{0,3,4\},\{1,2,5\},\{1,4,5\},\{2,0,6\},\{2,5,6\},\\ \{3,4,7\},\{3,6,7\},\{4,5,8\},\{4,7,8\},\{5,3,6\},\{5,8,3\},\\ \{6,7,1\},\{6,0,1\},\{7,8,2\},\{7,1,2\},\{8,3,0\},\{8,2,0\}\big\rangle$$

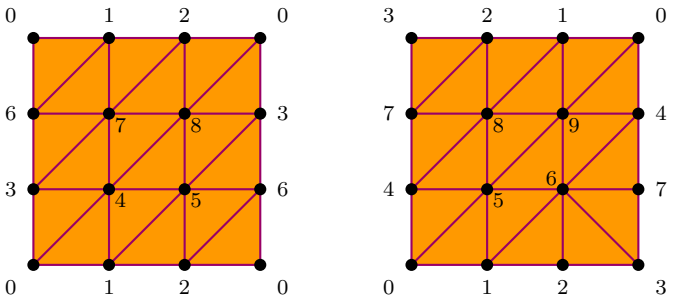


Figure 3.5: A Klein bottle and a projective plane

Example 3.5.7 (projective plane)**.** The projective plane is obtained from a square by glueing the sides as indicated in Figure 3.5. In order to avoid the unintentional identification of triangles, we need to be a bit careful and flip the triangles in the "lower right corner" (otherwise, we would get $\{2,3,7\}$ "twice"). The corresponding finite simplicial complex is:

$$\big\langle \{0,1,5\},\{0,4,5\},\{1,2,6\},\{1,5,6\},\{2,3,6\},\{3,6,7\},\\ \{4,5,8\},\{4,7,8\},\{5,6,9\},\{5,8,9\},\{6,7,4\},\{6,4,9\},\\ \{7,8,2\},\{7,3,2\},\{8,9,1\},\{8,2,1\},\{9,4,0\},\{9,1,0\}\big\rangle$$

Example 3.5.8 (simplicial sphere)**.** Let $n \in \mathbb{N}$. Then, the simplicial sphere of dimension n is the finite simplicial complex generated by all proper faces of the standard $(n+1)$-simplex:

$$\big\langle P(\{0,\dots,n+1\}) \setminus \{0,\dots,n+1\}\big\rangle$$

Geometrically, this corresponds to a hollow standard simplex and thus to a sphere.

3.5.2 Lean

We implement the material on the generation of simplicial complexes from Section 3.5.1.

Source code 3.5.9. This is `gen_simplicial_complex.lean` of the git repo [47].

```
import tactic          -- standard proof tactics
import data.set        -- basics on sets
import data.set.finite -- basics on finite sets
import data.finset     -- type-level finite sets
import data.finset.slice
import simplicial_complex     -- basics on simplicial complexes
import fin_simplicial_complex -- basics on finite complexes

open classical  -- we work in classical logic
```

This is a straightforward translation with one small exception: In contrast to the pen-and-paper situation, we cannot skip over the detail that generated simplicial complexes indeed are simplicial complexes. The proof of finiteness of the resulting simplicial complex is hidden in the fact that the union `bUnion` and the powerset `finset.powerset` operate on `finset` and thus their implementation already contains the necessary finiteness proofs.

```
@[simp]
def fingen_simplices
    {a : Type*} [decidable_eq a]
```

```
    (S : finset (finset a))
  : finset (finset a)
:= S.bUnion (λ x, finset.powerset x)

-- this set is indeed closed under subsets
lemma fingen_simplices_sub_closed
      {a : Type*} [decidable_eq a]
      (S : finset (finset a))
    : is_subset_closed (↑(fingen_simplices S) : set (finset a))
:=
begin
  assume s : finset a,
  assume s_in_gS: s ∈ fingen_simplices S,

  assume t : finset a,
  assume t_sub_s : t ⊆ s,

  show t ∈ ↑(fingen_simplices S), from
  begin
    simp only[fingen_simplices, finset.mem_coe,
              finset.mem_bUnion, finset.mem_powerset] at *,
    rcases s_in_gS with ⟨ s' : finset a, ⟨ s'_in_S, s_sub_s'⟩⟩,

    -- We use s' as witness that t lies in fingen_simplices S
    use s',
    show _, by exact ⟨ s'_in_S, by tauto ⟩,
  end,
end

def fingen_simplicial_complex
    {a : Type*} [decidable_eq a]
    (S : finset (finset a))
  : fin_simplicial_complex a
:= fin_simplicial_complex.mk
     (fingen_simplices S)
     (fingen_simplices_sub_closed S)
```

Examples Finally, we reach the stage at which our choice to integrate built-in finiteness guarantees pays off: We can let Lean perform computations on our finitely generated examples. For instance, we can compute the set of all vertices, the set of all simplices, or the set of all simplices of a given (non-negative) dimension.

```
def simplices_of_dim
    {a : Type*} -- [decidable_eq a]
    (X : fin_simplicial_complex a)
```

```
    (n : nat)
  : finset (finset a)
:= X.simplices.slice (n+1)
```

We generate the examples as in Section 3.5.1 by specifying a finite generating set of simplices.

```
def cylinder
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
     { {0,1,4}, {0,3,4}, {1,2,5}, {1,4,5}, {2,3,0}, {2,5,3} }

#eval cylinder.simplices
/- {{0, 1, 4}, {0, 1}, {0, 2, 3}, {0, 2}, {0, 3, 4}, {0, 3},
   {0, 4}, {0}, {1, 2, 5}, {1, 2}, {1, 4, 5}, {1, 4}, {1, 5},
   {1}, {2, 3, 5}, {2, 3}, {2, 5}, {2}, {3, 4}, {3, 5}, {3},
   {4, 5}, {4}, {5}, {}}
-/

#eval simplices_of_dim cylinder 0
      -- {{0}, {1}, {2}, {3}, {4}, {5}}
#eval simplices_of_dim cylinder 1 -- ...
#eval simplices_of_dim cylinder 2 -- ...
#eval simplices_of_dim cylinder 3 -- {}

def moebius
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
    ({ {0,1,4}, {0,3,4}, {1,2,5}, {1,4,5}, {2,3,0}, {2,5,0} })

def octahedron
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
    ({ {0,1,5}, {0,1,4}
     , {1,2,5}, {1,2,4}
     , {2,3,5}, {2,3,4}
     , {3,0,5}, {3,0,4} })

def torus
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
     { {0,1,4}, {0,3,4}, {1,2,5}, {1,4,5}, {2,0,3}, {2,5,3}
     , {3,4,7}, {3,6,7}, {4,5,8}, {4,7,8}, {5,3,6}, {5,8,6}
     , {6,7,1}, {6,0,1}, {7,8,2}, {7,1,2}, {8,6,0}, {8,2,0} }

#eval torus.simplices
```

```
#eval simplices_of_dim torus 0
#eval simplices_of_dim torus 1
#eval simplices_of_dim torus 2

def klein_bottle
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
     { {0,1,4}, {0,3,4}, {1,2,5}, {1,4,5}, {2,0,6}, {2,5,6}
     , {3,4,7}, {3,6,7}, {4,5,8}, {4,7,8}, {5,3,6}, {5,8,3}
     , {6,7,1}, {6,0,1}, {7,8,2}, {7,1,2}, {8,3,0}, {8,2,0} }

def projective_plane
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
     { {0,1,5}, {0,4,5}, {1,2,6}, {1,5,6}, {2,3,6}, {3,6,7}
     , {4,5,8}, {4,7,8}, {5,6,9}, {5,8,9}, {6,7,4}, {6,4,9}
     , {7,8,2}, {7,3,2}, {8,9,1}, {8,2,1}, {9,4,0}, {9,1,0} }

#eval finset.card (simplices_of_dim projective_plane 2) -- 18

def sphere
    (n : nat)
  : fin_simplicial_complex nat
:= fingen_simplicial_complex
     (finset.ssubsets (finset.range (n+2)))

#eval (sphere 0).simplices -- {{0}, {1}, {}}
#eval (sphere 1).simplices -- ...
#eval (sphere 2).simplices -- ...
#eval simplices_of_dim (sphere 3) 3
      -- {{0, 1, 2, 3}, {0, 1, 2, 4}, {0, 1, 3, 4},
      -- {0, 2, 3, 4}, {1, 2, 3, 4}}
```

However, we cannot let Lean directly compute the dimension of finite simplicial complexes: We defined the dimension in an indirect way.

3.6 Combining Simplicial Complexes

We can generate new simplicial complexes from existing simplicial complexes through combinators. There are many interesting such combinators such as products, joins, suspensions, subdivisions, ... For the sake of brevity and simplicity, we only consider two basic examples: Unions and intersections of two simplicial complexes. Wedges are introduced in Exercise 3.E.17.

Again, we restrict to the case of finite simplicial complexes so that we retain the benefit of evaluability.

3.6.1 Pen-and-Paper

The union and intersection of two finite simplicial complexes is literally defined by the union and intersection, respectively.

Definition 3.6.1 (union of simplicial complexes)**.** Let X and Y be finite simplicial complexes. The *union of X and Y* is the finite simplicial complex $X \cup Y$.

Definition 3.6.2 (intersection of simplicial complexes)**.** Let X and Y be finite simplicial complexes. The *intersection of X and Y* is the finite simplicial complex $X \cap Y$.

Both definitions come with the obligation of proving that the results indeed are simplicial complexes and finite:

Proposition 3.6.3. *Let X and Y be finite simplicial complexes.*

1. *The set $X \cup Y$ is closed under taking subsets.*
2. *The set $X \cap Y$ is closed under taking subsets.*
3. *The sets $X \cup Y$ and $X \cap Y$ are finite.*

Proof. Ad 1. Let $\sigma \in X \cup Y$ and let $\tau \subset \sigma$. We show that $\tau \in X \cup Y$: We distinguish two cases:

- If $\sigma \in X$, then $\tau \in X$, because X is closed under taking subsets. Therefore, $\tau \in X \cup Y$.
- If $\sigma \in Y$, then we can argue in the same way.

Ad 2. Let $\sigma \in X \cap Y$ and let $\tau \subset \sigma$. We show that $\tau \in X \cap Y$: We have $\sigma \in X$ and $\sigma \in Y$. Because X and Y are closed under taking subsets, we obtain $\tau \in X$ and $\tau \in Y$. Thus, $\tau \in X \cap Y$.

Ad 3. Because X and Y are finite sets also their union and their intersection is finite. □

Example 3.6.4 (zigzag)**.** The finite simplicial complexes Z_n with $n \in \mathbb{N}$ sketched in Figure 3.6 can be defined inductively via the union combinator:

- We define Z_0 as the finite simplicial complex generated by the singleton $\{(0, 0)\}$.

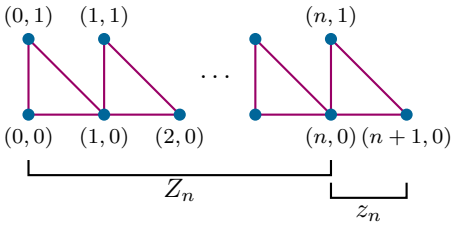


Figure 3.6: The zigzag complex from Example 3.6.4, schematically

- For $n \in \mathbb{N}$, we write

$$z_n := \big\langle \{(n,0),(n+1,0)\}, \{(n,0),(n,1)\}, \{(n,1),(n+1,0)\} \big\rangle$$

  and then define inductively

$$Z_{n+1} := Z_n \cup z_n.$$

3.6.2 Lean

We implement the material on unions and intersections of finite simplicial complexes from Section 3.6.1.

Source code 3.6.5. This is `comb_simplicial_complex.lean` of the git repo [47].

```
import tactic           -- standard proof tactics
import data.set         -- basics on sets
import data.set.finite -- basics on finite sets
import data.finset      -- type-level finite sets
import simplicial_complex      -- basics on simplicial complexes
import gen_simplicial_complex -- generation of finite complexes

open classical  -- we work in classical logic
```

Because `finset` supports the union and intersection of two `finset` entities over the same base type, the formalisation is straightforward:

```
lemma union_is_subset_closed
    {a : Type*} [decidable_eq a]
    (S : finset (finset a))
    (T : finset (finset a))
    (S_sub_closed : is_subset_closed (S : set (finset a)))
    (T_sub_closed : is_subset_closed (T : set (finset a)))
  : is_subset_closed (↑(S ∪ T) : set (finset a))
```

```
:=
begin
  assume s,
  assume s_in_ST : s ∈ ↑(S ∪ T),
  assume t,
  assume t_sub_s : t ⊆ s,
  simp only [is_subset_closed, finset.mem_coe] at *,

  have case_s_in_S : s ∈ S → t ∈ S ∪ T, from
  begin
    assume s_in_S: s ∈ S,
    have t_in_S : t ∈ S,
         by exact S_sub_closed s s_in_S t t_sub_s,
    show _, by finish,
    -- or: exact or.inl (S_sub_closed s s_in_S t t_sub_s),
  end,

  have case_s_in_T : s ∈ T → t ∈ S ∪ T, from
  begin
    assume s_in_T : s ∈ T,
    have t_in_T : t ∈ T,
         by exact T_sub_closed s s_in_T t t_sub_s,

    show _, by finish,
  end,

  show _,
       by {simp only[finset.mem_union] at s_in_ST,
           exact or.elim s_in_ST case_s_in_S case_s_in_T},
end

def union_sc
    {a : Type*} [decidable_eq a]
    (X : fin_simplicial_complex a)
    (Y : fin_simplicial_complex a)
  : fin_simplicial_complex a
:= fin_simplicial_complex.mk
      (X.simplices ∪ Y.simplices)
      (by exact union_is_subset_closed X.simplices Y.simplices
                  X.subset_closed Y.subset_closed)
```

Alternatively, we could have implemented `union_sc` as the finite simplicial complex generated by the union of simplices. This would simplify the definition, but might make reasoning with this definition more complicated. In a completionist treatment of the topic, both description should be available.

```
lemma inter_is_subset_closed
    {a : Type*} [decidable_eq a]
    (S : finset (finset a))
    (T : finset (finset a))
    (S_sub_closed : is_subset_closed (S : set (finset a)))
    (T_sub_closed : is_subset_closed (T : set (finset a)))
  : is_subset_closed (↑(S ∩ T) : set (finset a))
:=
begin
  assume s,
  assume s_in_ST : s ∈ ↑(S ∩ T),
  assume t,
  assume t_sub_s : t ⊆ s,
  simp only [is_subset_closed, finset.mem_coe] at *,

  have t_in_S : t ∈ S, from
  begin
    have s_in_S : s ∈ S, by finish,
    show t ∈ S, by exact S_sub_closed s s_in_S t t_sub_s,
  end,

  have t_in_T : t ∈ T, from
  begin
    have s_in_T : s ∈ T, by finish,
    show t ∈ T, by exact T_sub_closed s s_in_T t t_sub_s,
  end,

  show _,
       by {simp only[finset.mem_inter],
           exact and.intro t_in_S t_in_T},
end

def inter_sc
    {a : Type*} [decidable_eq a]
    (X : fin_simplicial_complex a)
    (Y : fin_simplicial_complex a)
  : fin_simplicial_complex a
:= fin_simplicial_complex.mk
     (X.simplices ∩ Y.simplices)
     (by exact inter_is_subset_closed X.simplices Y.simplices
                 X.subset_closed Y.subset_closed)
```

The zigzag complex As an example of the union combinator, we formalise the construction of the zigzag complex from Example 3.6.4.

```
def zig
    (n : nat)
  : fin_simplicial_complex (int × int)
:= fingen_simplicial_complex
     { {(n,0), (n+1,0)}, {(n,0), (n,1)}, {(n,1), (n+1,0)} }

def zigzag
  : nat → fin_simplicial_complex (int × int)
| 0            := fingen_simplicial_complex { {(0,0)} }
| (nat.succ n) := union_sc (zigzag n) (zig n)
```

3.7 The Euler Characteristic

The Euler characteristic is one of the first invariants of algebraic topology and can be computed directly from simplicial structures.

3.7.1 Pen-and-Paper

The Euler characteristic of a finite simplicial complex is the alternating sum of the number of simplices in the given dimension.

Definition 3.7.1 (Euler characteristic)**.** Let X be a finite simplicial complex. The *Euler characteristic of* X is defined by

$$\chi(X) := \sum_{n\in\mathbb{N}} (-1)^n \cdot \left|\{\sigma \in X \mid \dim\sigma = n\}\right| \in \mathbb{Z}.$$

As X is finite, this sum only has finitely many non-zero summands and thus gives a well-defined integer. Moreover, the Euler characteristic has the following alternative description:

Proposition 3.7.2 (Euler characteristic, reorganisation)**.** *Let X be a finite simplicial complex. Then*

$$\chi(X) = \sum_{\sigma\in X} p(\dim\sigma),$$

where p is the truncated parity function:

$$p\colon \mathbb{Z} \longrightarrow \mathbb{Z}$$
$$n \longmapsto \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n \geq 0 \text{ is even} \\ -1 & \text{if } n \geq 0 \text{ is odd} \end{cases}$$

Proof. The claim follows by reordering the terms of the (finite) sums: In both descriptions, each non-empty simplex σ contributes $(-1)^{\dim \sigma}$.

The only subtle point is that the empty simplex contributes 0 in both sums. In the definition of $\chi(X)$ this is ensured by only taking simplices of non-negative dimension into account; in the alternative description, this is contained in the definition of p. □

Example 3.7.3 (Euler characteristic)**.** Careful counting shows the following:

- The cylinder has Euler characteristic $6 - 12 + 6 = 0$.
- The Möbius strip has Euler characteristic $6 - 12 + 6 = 0$.
- The octahedron has Euler characteristic $6 - 12 + 8 = 2$.
- The torus has Euler characteristic $9 - 27 + 18 = 0$.
- The Klein bottle has Euler characteristic $9 - 27 + 18 = 0$.
- The projective plane has Euler characteristic $10 - 27 + 18 = 1$.
- The simplicial spheres have Euler characteristic

$$\begin{aligned} 2 &\quad \text{in dimension } 0\\ 3 - 3 = 0 &\quad \text{in dimension } 1\\ 4 - 6 + 4 = 2 &\quad \text{in dimension } 2\\ 5 - 10 + 10 - 5 = 0 &\quad \text{in dimension } 3. \end{aligned}$$

A key property of the Euler characteristic is additivity:

Proposition 3.7.4 (Euler characteristic of unions)**.** *Let X and Y be finite simplicial complexes. Then*

$$\chi(X \cup Y) = \chi(X) + \chi(Y) - \chi(X \cap Y).$$

Proof. We use the description of the Euler characteristic from Proposition 3.7.2. The claimed additivity formula for the Euler charcteristic is then a consequence of the usual inclusion-exclusion principle for finite sums: We have

$$\begin{aligned} \chi(X \cup Y) &= \sum_{\sigma \in X \cup Y} p(\dim \sigma) && \text{(Proposition 3.7.2)}\\ &= \sum_{\sigma \in X} p(\dim \sigma) + \sum_{\sigma \in Y} p(\dim \sigma)\\ &\quad - \sum_{\sigma \in X \cap Y} p(\dim \sigma) && \text{(inclusion/exclusion principle)}\\ &= \chi(X) + \chi(Y) - \chi(X \cap Y) && \text{(Proposition 3.7.2),} \end{aligned}$$

as claimed. □

Proposition 3.7.5 (Euler characteristic of zigzags)**.** *Let $n \in \mathbb{N}$. For the simplicial complex Z_n from Example 3.6.4, we have*

$$\chi(Z_n) = 1 - n.$$

Proof. We proceed by induction over n:

- *Base case.* For $n = 0$, the simplicial complex $Z_n = Z_0$ is a single point. We obtain
$$\chi(Z_0) = 1 = 1 - n,$$
as claimed.
- *Induction hypothesis.* Let $m \in \mathbb{N}$. We assume that the claim is proved for m, i.e., $\chi(Z_m) = 1 - m$.
- *Induction step.* We show that the claim then also holds for $m + 1$. To this end, we calculate
$$\begin{aligned}\chi(Z_{m+1}) &= \chi(Z_m \cup z_m) && \text{(by definition of } Z_{m+1}\text{)}\\ &= \chi(Z_m) + \chi(z_m) - \chi(Z_m \cap z_m) && \text{(Proposition 3.7.4)}\\ &= 1 - m + \chi(z_m) - \chi(Z_m \cap z_m). && \text{(by the induction hypothesis)}\end{aligned}$$
Because z_m is isomorphic to the simplicial sphere of dimension 1 and because the Euler characteristic is invariant under simplicial isomorphisms, we obtain $\chi(z_m) = 0$.
Moreover, $Z_m \cap z_m = \{\{(m, 0)\}, \emptyset\}$ and so $\chi(Z_m \cap z_m) = 1$.
In combination, we obtain
$$\chi(Z_{m+1}) = 1 - m + 0 - 1 = 1 - (m + 1),$$
as claimed. □

3.7.2 Lean

We implement the material on the Euler characteristic of finite simplicial complexes from Section 3.7.1.

Source code 3.7.6. This is `euler_characteristic.lean` of the git repo [47].

```
import tactic          -- standard proof tactics
import simplicial_complex
import gen_simplicial_complex
import comb_simplicial_complex
import algebra.big_operators.basic
import algebra.big_operators.order
```

```
open_locale big_operators -- to enable ∑ notation

open classical  -- we work in classical logic
```

For the definition of the Euler characteristic, we use the description from Proposition 3.7.2. The advantage of this description over the "original" definition from Definition 3.7.1 is that it does not incur any obligations to prove finiteness of the sum. Also, the proof of additivity of the Euler characteristic is most transparent in this description.

```
def parity
    (x : int)
  : int
:= if x < 0 then 0
            else if int.mod x 2 = 0 then  1
                                    else -1

#eval parity (-1) -- 0
#eval parity 0    -- 1
#eval parity 2022 -- 1
#eval parity 2023 -- -1

def euler_char
    {a : Type*}
    (X : fin_simplicial_complex a)
  : int
:= ∑ (s : finset a) in X.simplices, parity (dim s)
```

To increase readability, we introduce the common notation for the Euler characteristic:

```
notation `χ` := euler_char
```

Examples As we opted for a finite and explicit setup, we can let Lean compute the Euler characteristic of concrete simplicial complexes:

```
#eval χ cylinder -- 0
#eval χ moebius -- 0
#eval χ octahedron -- 2
#eval χ torus -- 0
#eval χ klein_bottle -- 0
#eval χ projective_plane -- 1

#eval χ (sphere 0) -- 2
#eval χ (sphere 1) -- 0
#eval χ (sphere 2) -- 2
#eval χ (sphere 3) -- 0
#eval χ (sphere 4) -- 2
```

Such computations can easily be turned into statements that can be reused in other places:

```
lemma euler_char_sphere_2
    : χ (sphere 2) = 2
:=
begin
  refl,
end
```

The Euler characteristic of unions The additivity of the Euler characteristic follows directly from the inclusion/exclusion formula for sums over `finset`s.

```
theorem euler_char_union
    {a : Type*} [decidable_eq a]
    (X : fin_simplicial_complex a)
    (Y : fin_simplicial_complex a)
  : χ (union_sc X Y)
  = χ X + χ Y - χ (inter_sc X Y)
:=
begin
  calc χ (union_sc X Y)
       = ∑ s in X.simplices ∪ Y.simplices, parity (dim s)
       : by refl
   ... = ∑ s in X.simplices ∪ Y.simplices, parity (dim s)
       + ∑ s in X.simplices ∩ Y.simplices, parity (dim s)
       - ∑ s in X.simplices ∩ Y.simplices, parity (dim s)
       : by ring
   ... = ∑ s in X.simplices, parity (dim s)
       + ∑ s in Y.simplices, parity (dim s)
       - ∑ s in X.simplices ∩ Y.simplices, parity (dim s)
       : by simp[finset.sum_union_inter]
   ... = χ X + χ Y - χ (inter_sc X Y)
       : by refl,
end
```

The Euler characteristic of zigzags The Euler characteristic of an individual zigzag can be computed directly:

```
#eval χ (zigzag 0) -- 1
#eval χ (zigzag 1) -- 0
#eval χ (zigzag 2) -- -1
#eval χ (zigzag 3) -- -2
```

Finally, we indicate how we can inductively compute the Euler characteristic of the zigzag complexes, following the same strategy as in Proposition 3.7.5. We only carry out the main steps; the missing proofs are provisionally admitted through the tactic `sorry`.

```
lemma euler_char_zig
    (n : nat)
  : χ (zig n) = 0
:=
begin
  sorry,
end

lemma euler_char_zigzag_inter
    (n : nat)
  : χ (inter_sc (zigzag n) (zig n)) = 1
:=
begin
  let X := inter_sc (zigzag n) (zig n),

  have inter_simplices : X.simplices = { {(n,0)}, ∅ }, from
  begin
    sorry,
  end,

  show χ X = 1,
       by {simp only[euler_char, inter_simplices], finish},
end

lemma euler_char_zigzag
      (n : nat)
    : χ (zigzag n) = 1 - n
:=
begin
  induction n with m ind_hyp,

  -- base case: 0
  case nat.zero : {refl},

  -- induction step: m -> m + 1
  case nat.succ :
  begin
    calc χ (zigzag (m+1))
         = χ (zigzag m)
         + χ (zig m)
         - χ (inter_sc (zigzag m) (zig m))
         : by exact euler_char_union (zigzag m) (zig m)
     ... = 1 - m + 0 - 1
         : by simp[ind_hyp,
                   euler_char_zig, euler_char_zigzag_inter]
```

```
    ... = 1 - (m+1)
        : by ring,
  end
end
```

These proof outlines are an example of a prototyping process; we will discuss such processes in more detail in Chapter 4. The gaps in the outlines above are filled in Exercise 3.E.13, Exercise 3.E.14, and Exercise 3.E.16.

3.8 Towards a Library

The material covered in this chapter is a part of the basic theory of simplicial complexes. Building on this formalisation, one could now introduce further constructions of simplicial complexes, add the layer of geometric realisation, proceed in the direction of simplicial homology, or connect the theory with their point-free siblings: simplicial sets.

Providing basics as libraries unleashes the full power of proof assistants: Others can build on previous work and thus contribute to the construction of the formalisation palace.

On a technical level, providing a library in particular entails using state-of-the-art source code deployment mechanisms (or direct contribution to mathlib [42]), providing documentation [41], and adhering to some extent to naming conventions [3, 4].

In the following, we briefly outline several structural steps on the path towards a library.

3.8.1 Interaction with Other Libraries

Good integration of new formalisation bits with existing libraries maximises usability, applicability, and robustness.

For example, making simplicial complexes and simplicial maps as considered in this chapter an instance of the standard category framework of mathlib, gives access to all standard notation and inheritance properties for morphisms, isomorphisms, etc. Defining such an instance is the goal of Exercise 3.E.6. In contrast, if we do *not* provide such an instance, then we would need to reprove lots of basic properties, leading to extra work and introducing more opportunities for omissions or confusion.

3.8.2 Completeness

Like good textbooks, libraries should aim at a level of completeness within a well-defined area. In particular, this includes:

- Common equivalent characterisations of the notions under consideration should be provided.
- Common inheritance properties should be provided.
- Typical arguments should be lemmas or tactics [6, Chapter 5].
- Standard examples should be provided.

These items should be interpreted both on the pen-and-paper level and on the Lean level. The latter usually also leads to useful reformulations of results that on pen-and-paper would be performed implicitly.

Whether a reasonable level of completeness is reached can (and should) be tested by formalising example situations using the library.

For example, in mathlib, the library `algebra.group` on basic group theory contains many elementary statements on manipulating terms and calculations in groups and their interaction with group homomorphisms. Many of these elementary statements only differ by simple transformations. Having these statements readily available in different forms substantially increase usability. Moreover, this library on group theory is complemented by the `group` tactic in `tactic.group`, thereby further simplifying use of this library.

3.8.3 Substructures and Quotients

As a particular case of the quest for completeness, typically in libraries on mathematical theories, the need for a treatment of substructures and quotients arises.

In Lean, the formalisation of substructures often uses subtypes; for quotients, some support for quotient types is provided [14, Section 2.7.1].

Substructures For example, subcomplexes of simplicial complexes are introduced in Exercise 3.E.2. This is an atypical example of substructures: We formalised simplicial complexes in terms of sets and therefore subcomplexes can be directly formalised as simplicial complexes related by actual inclusion.

The formalisation of substructures of structures based on type classes over carrier types usually involves an additional step:

- One first introduces a record consisting of a set over the original carrier type and suitable closure properties for the original operations of the structure on this set.
- One then shows that such substructures admit a canonical instance of the original type class and that the "inclusion" defines a morphism in the appropriate sense.

A concrete example of such substructures is the notion `subgroup` in the mathlib, formalising subgroups.

How is this related to subtypes? Lean subtypes of a type `a` are basically modelled by predicates `a → Prop`. Similarly, sets over a basetype `a` are modelled by predicates `a → Prop`. For substructures, often sets over the original basetype are used.

Quotients Quotient types, in general, are a tricky subject in type theory [15, 17]. Lean takes on a pragmatic approach and introduces quotient types via an axiomatic description. These axioms provide "quotients" of a type `a` by a binary relation `a → a → Prop`. The axioms for quotient types are contained in the Lean core (`init_quotient`, which is called in `core.lean`) and extended to the full quotient mechanism in `init.data.quot` and `data.quot`. In particular, Lean includes the assumption of the soundness axiom `quot.sound`.

In mathematics, one usually considers quotients by equivalence relations (and not general binary relations). This case is taken care of by setoids: A *setoid* is a pair, consisting of a set and an equivalence relation on this set. In Lean, the `setoid` type class is provided in `init.data.setoid`. Quotients of setoids are packaged into the `quotient` concept.

For example, the type `real` of real numbers in mathlib is constructed as the Cauchy completion of the rationals. The Cauchy completion is a quotient of the type of Cauchy sequences by the equivalence relation generated by zero sequences. Quotient groups are defined in `group_theory.coset` as quotients by the coset equivalence relation of a normal subgroup. In the simplicial world, wedges of (pointed) simplicial complexes can be defined in terms of quotient types (Exercise 3.E.17).

3.8.4 Generality

Libraries should provide a compromise between high-level abstraction mechanisms, which can be instantiated to further situations, and straightforward applicability.

In the situation of simplicial complexes, one might wonder whether simplicial sets would be a "better" framework. On the one hand, simplicial sets aim at a more point-free description of simplicial phenomena, which leads to slick formalisations. On the other hand, many concrete applications involve actual finite simplicial complexes and thus it is desirable to also provide an accessible framework for simplicial topology that can handle automatic computations.

For these notes, we decided in favour of concreteness and simplicial complexes. A more completionist approach to the subject, of course, should also spell out the relation with simplicial sets.

3.E Exercises

Source code. Solutions [47]: `simplicial_solution.lean`

General Simplicial Complexes

Exercise 3.E.1 (a small simplicial complex)**.**

1. Pen-and-paper: Give an example of a simplicial complex that has exactly two non-empty simplices.
2. Formalise this example in Lean, including a proof that this example has the desired property.

Exercise 3.E.2 (subcomplexes)**.**

1. Pen-and-paper: Introduce a notion of "subcomplexes" of simplicial complexes such that the "inclusion of a subcomplex" is a simplicial map.
2. Formalise these notions/statements/proofs in Lean.

Exercise 3.E.3 (nerves)**.** Let X be a set and let $U \subset P(X)$ be a set of subsets. The *nerve of* U is the simplicial complex

$$\big\{V \subset U \;\big|\; V \text{ is finite and } \bigcap V \neq \emptyset\big\} \cup \{\emptyset\}.$$

Formalise this construction in Lean.

Exercise 3.E.4 (Rips complexes)**.** Let (X, d) be a metric space. Given a radius $r \in \mathbb{R}_{\geq 0}$, the *Rips complex of* (X, d) *of radius* r is defined as

$$R_r(X, d) := \big\{\sigma \subset X \;\big|\; \sigma \text{ is finite and } \forall_{x,y \in \sigma} \quad d(x, y) \leq r\big\}.$$

1. Formalise this construction in Lean.
2. Pen-and-paper: Prove that $R_r(X, d)$ is a subcomplex of $R_s(X, d)$ for all $r, s \in \mathbb{R}_{\geq 0}$ with $r \leq s$. Subcomplexes are introduced in Exercise 3.E.2.
3. Formalise this statement/proof in Lean.

Exercise 3.E.5 (the line)**.**

1. Formalise the line from Example 3.2.5 in Lean.
2. Pen-and-paper: Prove that the line from Example 3.2.5 has dimension 1.
3. Formalise this statement/proof in Lean.

Exercise 3.E.6 (the category of simplicial complexes)**.** Define an instance

```
instance {a : Type*} : category (simplicial_complex a)
```

for the type class `category` of `category_theory.category.basic`, using the composition of simplicial maps defined Section 3.3.2.

Finite Simplicial Complexes and Generation

Exercise 3.E.7 (the cube)**.** Formalise the surface of a cube as a finite simplicial complex in Lean. It might be useful to first formalise squares and then to construct a cube from six squares.

Exercise 3.E.8 (simplicial maps via generators)**.** Let S be a finite set of finite sets and let Y be a simplicial complex.

1. Pen-and-paper: State and prove a principle that allows to generate simplicial maps $\langle S \rangle \longrightarrow Y$ from maps defined on S.
2. Formalise this statement/proof in Lean.

Exercise 3.E.9 (dimension of generated simplicial complexes)**.**

1. Pen-and-paper: How can the dimension of a generated simplicial complex be described in terms of dimensions of the given generating set?
2. Formalise this statement/proof in Lean.
3. Use this result to show in Lean that the torus has dimension 2.

Exercise 3.E.10 (vertices of unions and intersections)**.** Let X and Y be finite simplicial complexes.

1. Pen-and-paper: How can the sets of vertices of $X \cup Y$ and $X \cap Y$ be computed from the sets of vertices of X and Y ?

   Which of these two statements is more difficult to prove?
2. Formalise the set of vertices of finite simplicial complexes in Lean as a `finset`.
3. Formalise the statements/proofs on the sets of vertices of unions and intersections of finite simplicial complexes in Lean.

Exercise 3.E.11 (simplicial complexes with a single vertex)**.**

1. Pen-and-paper: Show that a simplicial complex X that has exactly one vertex x is of the form $X = \{\{x\}, \emptyset\}$.
2. Formalise this statement/proof in Lean for finite simplicial complexes, using the `finset` of vertices of Exercise 3.E.10.

The Euler Characteristic

Exercise 3.E.12 (an estimate for the Euler characteristic)**.** Let X be a finite simplicial complex. Then $\left|\chi(X)\right| \leq |X|$.

1. Pen-and-paper: Prove this statement.
2. Formalise this statement/proof in Lean. It might be useful to first prove a corresponding claim for the function `parity`.

Exercise 3.E.13 (Euler characteristic and isomorphisms)**.** Show that the Euler characteristic of finite simplicial complexes is invariant under isomorphisms of simplicial complexes:

1. Pen-and-paper: Show that simplicial isomorphisms between simplicial complexes induce dimension-preserving bijections between the sets of simplices.
2. Pen-and-paper: Conclude that isomorphic finite simplicial complexes have the same Euler characteristic.
3. Formalise both statements and proofs in Lean.

Exercise 3.E.14 (the Euler characteristic of zig)**.** Complete the computation `euler_char_zig` of the Euler characteristic of the zig complexes in Section 3.7.2, using the isomorphism invariance of the Euler characteristic (Exercise 3.E.13) and the Euler characteristic of `zig 0`.

Exercise 3.E.15 (vertices of zig and zigzag)**.** We use the notation from Example 3.6.4. Let $n \in \mathbb{N}$ and let $k \in \mathbb{N}$ with $k + 1 < n$.

1. Pen-and-paper: Show that

$$\bigcup z_n \cap \bigcup z_{n+1} = \{(n+1, 0)\}, \quad \bigcup z_k \cap \bigcup z_n = \emptyset,$$
$$\bigcup Z_k \cap \bigcup z_n = \emptyset, \quad \bigcup Z_n \cap \bigcup z_n = \{(n, 0)\}.$$

2. Formalise these statements/proofs in Lean, using the `finset` of vertices from Exercise 3.E.10.

   For the empty intersections, it might be helpful to first inductively establish bounds on the first coordinate for vertices of the zig and zigzag complexes.

Exercise 3.E.16 (the Euler characteristic of zigzags)**.** Complete the computation `euler_char_zigzag` of the Euler characteristic of zigzags in Section 3.7.2, using Exercise 3.E.14, Exercise 3.E.15, and Exercise 3.E.11.

The Wedge of Finite Simplicial Complexes

Exercise 3.E.17 (wedges of finite simplicial complexes)**.** Let X and Y be simplicial complexes and let x_0 and y_0 be vertices of X and Y, respectively. Let

$$Z := (\bigcup X \sqcup \bigcup Y)/(x_0 \sim y_0)$$

be the set obtained from the vertices of X and Y by "identifying x_0 and y_0" and let $i_X \colon \bigcup X \longrightarrow Z$ and $i_Y \colon \bigcup Y \longrightarrow Z$ be the canonical maps induced by the inclusions of X and Y into the disjoint union. We consider the sets

$$i_X X := \big\{ i_X(\sigma) \bigm| \sigma \in X \big\},$$
$$i_Y Y := \big\{ i_Y(\sigma) \bigm| \sigma \in Y \big\}.$$

The *wedge* of X and Y with respect to x_0 and y_0 is the union complex $\langle i_X X \rangle \cup \langle i_Y Y \rangle$. In fact, $i_X X$ and $i_Y Y$ are already simplicial complexes, but for simplicity, we will not prove this.

Formalise this construction in Lean for finite simplicial complexes:

1. Formalise the wedge relation "$(x_0 \sim y_0)$". It will pay off later to formalise it in such a way that one can easily prove that the wedge relation is decidable and an equivalence relation.

2. Formalise the wedge type by using a quotient type of a suitable setoid; disjoint union types can be modelled by `sum` types.

3. Formalise the generating sets $i_X X$ and $i_Y Y$. Here, decidability of the wedge relation can be useful.

4. Formalise the construction of the wedge.

Exercise 3.E.18 (roses)**.** Define a function `rose` in Lean that, given a natural number n, returns the n-fold iterated wedge of `zig 0` with itself with respect to the basepoints induced by the vertex `(0,0)` of `zig 0`.

Geometrically, `rose n` can then be viewed as a "rose with n petals". For example, `rose 3` has exactly seven vertices and exactly nine simplices of dimension 1.

Hints. This is trickier than it looks: What is the type of `rose` ?!

4

Abstraction and Prototyping

We now move on to the formalisation of recent mathematical developments. Depending on the mathematical field, different challenges might appear. Two types of situations are particularly convenient for a direct formalisation:

- Theories and arguments that depend only on elementary notions or notions that already have been implemented; such theories can be formalised in a straightforward way, as we have already seen in the classical examplse treated in Chapter 2 and Chapter 3.
- Theories that are built on high-level abstraction mechanisms such as categories and universal properties; such theories can be formalised directly in a declarative style. An example is given in Section 4.1.

Not all theories fall into one of these two types. In Section 4.2, we will explain how results can be formalised indirectly through a suitable intermediate abstraction step. In this way, proof assistants encourage abstraction and emphasise the declarative perspective on mathematics.

Overview of this chapter.

4.1 Direct Formalisation: Functorial Semi-Norms

We give an example of a direct formalisation of a result that is already in abstract formalisation-friendly form. The example is motivated by geometric topology, but the abstracted version only requires a basic understanding of category theory language.

4.1.1 Pen-and-Paper

Functorial semi-norms on general functors are a generalisation of the notion of functorial semi-norms on singular (co)homology, originally introduced by Gromov in the context of simplicial volume [26, 25]. For simplicity, we only consider the case of finite homogeneous functorial semi-norms.

We will introduce the basic notions and prove the simple but fundamental observation that functorial semi-norms are trivial on so-called weakly flexible classes [44, Proposition 3.1]; in particular, this shows that (finite homogeneous) functorial semi-norms on representable functors are trivial [44, Corollary 4.1].

Definition 4.1.1 (semi-norm, homogeneous semi-norm)**.** Let A be an Abelian group. A *semi-norm on* A is a map $|\cdot|\colon A \longrightarrow \mathbb{R}_{\geq 0}$ with the following properties:

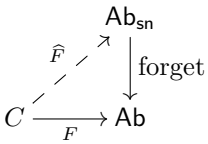
- We have $|0| = 0$.
- For all $x \in A$, we have $|-x| = |x|$.
- For all $x, y \in A$, we have $|x + y| \leq |x| + |y|$.

A semi-norm $|\cdot|$ on A is *homogeneous* if the following condition is satisfied: For all $x \in A$ and all $n \in \mathbb{Z}$, we have (where $|n|$ denotes the standard absolute value on $\mathbb{Z}$)

$$|n \cdot a| = |n| \cdot |a|.$$

Let Ab denote the category of Abelian groups and group homomorphisms. We write $\mathsf{Ab}_{\mathsf{sn}}$ for the category of all semi-normed Abelian groups and norm-non-increasing group homomorphisms.

Definition 4.1.2 (functorial semi-norm)**.** Let C be a category and let $F\colon C \longrightarrow \mathsf{Ab}$ be a functor. A *functorial semi-norm on* F is a factorisation functor $\widehat{F}\colon C \longrightarrow \mathsf{Ab}_{\mathsf{sn}}$ of F through the forgetful functor $\mathsf{Ab}_{\mathsf{sn}} \longrightarrow \mathsf{Ab}$



that is homogeneous on each object: For all $X \in \mathrm{Ob}(C)$, the semi-norm on $\widehat{F}(X)$ is homogeneous.

More explicitly, a functorial semi-norm on a functor $F\colon C \longrightarrow \mathsf{Ab}$ consists of a choice of a homogeneous semi-norm $|\cdot|_{\widehat{F}}$ on $F(X)$ for every $X \in \mathrm{Ob}(C)$ such that for all morphisms $f\colon X \longrightarrow Y$ in C and all $\alpha \in F(X)$, we have

$$\big|F(f)(\alpha)\big|_{\widehat{F}} \leq |\alpha|_{\widehat{F}}.$$

Definition 4.1.3 (weakly flexible)**.** Let C be a category, let $F\colon C \longrightarrow \mathsf{Ab}$ be a functor, and let $X \in \mathrm{Ob}(C)$. An element $\alpha \in F(X)$ is *weakly flexible (with respect to F)* if there exists an object $Y \in C$ and $\beta \in F(Y)$ such that the set

$$\deg(\beta, \alpha) := \big\{ n \in \mathbb{Z} \;\big|\; \exists_{f \in \mathrm{Mor}_C(Y,X)} \quad F(f)(\beta) = n \cdot \alpha \big\}$$

is infinite.

Proposition 4.1.4 (weak flexibility and functorial semi-norms [44, Proposition 3.4])**.** *Let C be a category, let $F\colon C \longrightarrow \mathsf{Ab}$ be a functor, let $X \in \mathrm{Ob}(C)$, and let $\alpha \in F(X)$ be weakly flexible. If $\widehat{F}\colon C \longrightarrow \mathsf{Ab}_{\mathsf{sn}}$ is a functorial semi-norm on F, then $|\alpha|_{\widehat{F}} = 0$.*

Proof. Because α is weakly flexible, there exists an object $Y \in \mathrm{Ob}(C)$ and $\beta \in F(Y)$ such that $\deg(\beta, \alpha)$ is infinite. If $\widehat{F}\colon C \longrightarrow \mathsf{Ab}_{\mathsf{sn}}$ is a functorial semi-norm on F, we obtain

$$\forall_{n \in \deg(\beta,\alpha)} \quad |\alpha|_{\widehat{F}} \leq \frac{1}{n} \cdot |\beta|_{\widehat{F}}.$$

As the set $\deg(\beta, \alpha)$ is infinite, we conclude that $|\alpha|_{\widehat{F}} = 0$. □

Corollary 4.1.5 (finite functorial semi-norms on representable functors [44, Corollary 4.1])**.** *Let C be a category and let $F\colon C \longrightarrow \mathsf{Ab}$ be a representable functor. If $\widehat{F}\colon C \longrightarrow \mathsf{Ab}_{\mathsf{sn}}$ is a functorial semi-norm on C, then $\widehat{F}$ is trivial, i.e., for all $X \in \mathrm{Ob}(C)$ and all $\alpha \in F(X)$, we have*

$$|\alpha|_{\widehat{F}} = 0.$$

Proof. By Proposition 4.1.4, it suffices to show that all classes are weakly flexible with respect to F.

Let $Y \in \mathrm{Ob}(C)$ be a representing object of F, i.e., $V \circ F \cong \mathrm{Mor}_C(Y, \cdot\,)$, where $V\colon \mathsf{Ab} \longrightarrow \mathsf{Set}$ is the forgetful functor; let $\beta \in F(Y)$ be the universal element, i.e., the element corresponding to $\mathrm{id}_Y \in \mathrm{Mor}_C(Y, Y)$.

Let $X \in \mathrm{Ob}(X)$ and let $\alpha \in F(X)$. We show that $\deg(\beta, \alpha) = \mathbb{Z}$: Clearly, $\deg(\beta, \alpha) \subset \mathbb{Z}$. Conversely, let $n \in \mathbb{Z}$. Then, $n \cdot \alpha$ corresponds to a morphism $f \in \mathrm{Mor}_C(Y, X)$, which means that

$$n \cdot \alpha = F(f)(\beta).$$

Therefore, $n \in \deg(\beta, \alpha)$.

Because $\deg(\beta, \alpha) = \mathbb{Z}$, the class α is weakly flexible with respect to F. □

Simplicial volume of weakly flexible manifolds Let M be an oriented closed connected manifold and let $n := \dim M$. We say that M is *weakly flexible* if there exists an oriented closed connected n-manifold N such that the set

$$\left\{ \deg f \;\middle|\; f \in \mathrm{map}(N, M) \right\} \subset \mathbb{Z}$$

of mapping degrees is infinite. If M is weakly flexible, then the $\mathbb{R}$-fundamental class $[M]_{\mathbb{R}}$ of M is weakly flexible with respect to the singular homology functor $H_n(\,\cdot\,;\mathbb{R})$ with $\mathbb{R}$-coefficients in degree n.

Hence, Proposition 4.1.4 shows: If M is weakly flexible, then every functorial semi-norm on $H_n(\,\cdot\,;\mathbb{R})$ is zero on $[M]_{\mathbb{R}}$.

A concrete example of a functorial semi-norm on $H_n(\,\cdot\,;\mathbb{R})$ is the ℓ^1-semi-norm [25, 44]. The *simplicial volume* of an oriented closed connected manifold is defined as the ℓ^1-semi-norm of the $\mathbb{R}$-fundamental class. In particular, we obtain: If M is an oriented closed connected weakly flexible manifold, then the simplicial volume of M is zero.

Irrepresentability of bounded cohomology Let $n \in \mathbb{N}_{>1}$. Then Corollary 4.1.5 shows that the bounded cohomology functor $H_b^n(\,\cdot\,;\mathbb{R})$ is *not* representable on the category of topological spaces, CW-complexes, groups, finitely presented groups, ... [44, Example 4.4]. Indeed, it is known that there exist corresponding X such that the functorial semi-norm $\|\cdot\|_\infty$ on $H_b^n(\,\cdot\,;\mathbb{R})$ is non-trivial on $H_b^n(X;\mathbb{R})$.

4.1.2 Lean

We implement the material from Section 4.1.1.

Source code 4.1.6. This is `ffsn_rep.lean` of the git repo [47].

We import basics on (semi-normed) Abelian groups and category theory. Moreover, we will make use of the criterion from Section 2.4.

```
import tactic -- standard proof tactics
import data.real.basic
import zero -- a criterion for real numbers to be zero
import algebra.category.Group.basic
import category_theory.functor
```

```
import analysis.normed.group.SemiNormedGroup

open classical -- we work in classical logic
```

Categories and functors Before going into the details of the implementation, we summarise basic Lean notation in categories:

- In a category, `X ⟶ Y` (with a long arrow!) denotes the type of all morphisms from `X` to `Y`.
- Forward composition of morphisms is denoted by ≫.
- If `C` and `D` are categories, then `C ⥤ D` denotes the type of all functors from `C` to `D`.
- Forward composition of functors is denoted by ⋙.
- The symbol ≅ does *not* mean "is isomorphic to", but denotes the type of all isomorphisms.

We use the categories of (semi-normed) Abelian groups from mathlib und define the corresponing forgetful functor. The categories `AddCommGroup` and $\texttt{SemiNormedGroup}_1$ are defined as categories of so-called bundled objects (i.e., the object type is a record type bundling a carrier type together with a type class instance of this carrier type).

```
notation `Ab`   := AddCommGroup
notation `Absn` := SemiNormedGroup₁

-- The forgetful functor Absn ⥤ Ab
noncomputable
instance has_forget_Absn_Ab
 : category_theory.has_forget₂ Absn Ab
:= { forget₂ :=
     { obj := λ X, AddCommGroup.of X,
       map := λ X Y, λ f : X ⟶ Y,
              AddCommGroup.of_hom
                (normed_group_hom.to_add_monoid_hom f) } }

notation `forget_sn` := has_forget_Absn_Ab.forget₂
```

Functorial semi-norms We take the opportunity to give a slightly cleaner definition of functorial semi-norms. Generally speaking, it is debatable whether equality of objects in categories is a meaningful concept or whether one should only work with isomorphisms. Similarly, equality of functors is not as well adapted to the category setting as natural isomorphisms. Therefore, we modify the definition of functorial semi-norm using natural isomorphisms instead of equality between functors. This is encoded in `ffsn_setup`.

```
def ffsn_on
    {C : Type*} [C_is_cat : category_theory.category C]
    (F : C ⇒ Ab)
    (Fsn : C ⇒ Absn)
:= F ≅ Fsn ⋙ forget_sn

def is_homogeneous_fsn
    {C : Type*} [C_is_cat : category_theory.category C]
    (Fsn : C ⇒ Absn)
  : Prop
:= ∀ X : C, ∀ a : (Fsn.obj) X,
   ∀ n : ℤ, ‖ n • a ‖ = |↑n| * ‖ a ‖

structure ffsn_setup
   (C : Type*)
   [C_is_cat : category_theory.category C]
:= mk :: (F : C ⇒ Ab)
         (Fsn : C ⇒ Absn)
         (Fsn_ffsn_on_F      : ffsn_on F Fsn)
         (Fsn_is_homogeneous : is_homogeneous_fsn Fsn)
```

The key property of functorial semi-norms is that norms of classes decrease under morphisms:

```
lemma fsn_est_explicit
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      {Y : C}
      {X : C}
      (f : Y ⟶ X)
      (b : s.Fsn.obj Y)
    : ‖ s.Fsn.map f b ‖ ≤ ‖ b ‖
:=
begin
  -- we just spell out the properties of morphisms in Absn
  let f' := s.Fsn.map f,
  have f_noninc : normed_group_hom.norm_noninc f'.1,
       by exact f'.2,
  show _,
       by exact f_noninc b,
end
```

Lifting elements Functors connected through `forget_sn` and a natural isomorphism, on objects, lead to isomorphic underlying Abelian groups.

```
noncomputable
def underlying_add_comm_group_iso
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      (X : C)
    : s.F.obj X ≅ AddCommGroup.of (s.Fsn.obj X).1
:= s.Fsn_ffsn_on_F.app X
```

Therefore, for a functorial semi-norm on `F`, we can view `F`-classes as `Fsn`-classes. In pen-and-paper mathematics, we quickly pass over this step. In Lean, we need to spell this out in more detail. We split this lifting into two steps:

- `lift1` lifts from `F` to the Abelian groups underlying the values of `Fsn`, using the natural isomorphism from `F` to `Fsn ⋙ forget_sn`;
- `lift2` lifts from there to the actual values of `Fsn` using the built-in conversions of carriers.

```
def lift1
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
  : s.F.obj X → AddCommGroup.of (s.Fsn.obj X)
:= (s.Fsn_ffsn_on_F.app X).hom

def lift2
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
  : AddCommGroup.of (s.Fsn.obj X) → (s.Fsn.obj X)
:= by tauto

@[simp]
noncomputable
def lift_elt
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
  : (s.F.obj X) → (s.Fsn.obj X)
:= (lift2 s) ∘ (lift1 s)
```

In particular, using a functorial semi-norm `Fsn` on `F`, we can measure the size of `F`-classes by viewing them as `Fsn`-classes.

```
@[simp]
noncomputable
def sn
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
    (a : s.F.obj X)
  : ℝ
:= ‖ lift_elt s a ‖
```

To use `lift_elt` in computations, it is useful to know that `lift_elt` is compatible with `zsmul` and that it is natural. In both cases, we prove the corresponding statements for `lift1` and `lift2` and then combine them.

```
lemma lift1_zsmul
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    (X : C)
  : ∀ n : int, ∀ a : s.F.obj X,
    lift1 s (n · a) = n · lift1 s a
:=
begin
  simp[lift1],
end

lemma lift2_zsmul
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
  : ∀ n : int, ∀ a : AddCommGroup.of (s.Fsn.obj X),
    lift2 s (n · a) = n · lift2 s a
:=
begin
  tauto,
end

lemma lift_elt_zsmul
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
  : ∀ n : int, ∀ a : s.F.obj X,
    lift_elt s (n · a) = n · lift_elt s a
:=
begin
  simp[lift1_zsmul, lift2_zsmul],
end
```

```
noncomputable
def underlying_hom
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {Y : C}
    {X : C}
    (f : Y ⟶ X)
:= AddCommGroup.of_hom
   (normed_group_hom.to_add_monoid_hom (s.Fsn.map f).1)

lemma lift1_map
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      {Y : C}
      {X : C}
      (f : Y ⟶ X)
      (b : s.F.obj Y)
    : lift1 s (s.F.map f b)
    = (underlying_hom s f) (lift1 s b)
:=
begin
  -- the natural iso from F to Fsn ⋙ forget_sn
  let t := s.Fsn_ffsn_on_F,

  calc lift1 s (s.F.map f b)
        = ((s.F.map f) ≫ (t.hom.app X)) b
        : by congr
    ... = ((t.hom.app Y) ≫ ((s.Fsn ⋙ forget_sn).map f)) b
        : by rw [t.hom.naturality']
    ... = underlying_hom s f (lift1 s b)
        : by refl,
end

lemma lift2_map
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      {Y : C}
      {X : C}
      (f : Y ⟶ X)
      (b : AddCommGroup.of (s.Fsn.obj Y))
    : lift2 s (underlying_hom s f b)
    = s.Fsn.map f (lift2 s b)
:=
begin
```

```
  tauto,
end

lemma lift_elt_map
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      {Y : C}
      {X : C}
      (f : Y ⟶ X)
      (b : s.F.obj Y)
    : lift_elt s (s.F.map f b) = s.Fsn.map f (lift_elt s b)
:=
begin
  simp[lift1_map, lift2_map],
end
```

Weakly flexible classes We translate the definition of degree sets and the definition of weak flexibility. For weak flexibility, we replace the condition that the degree set is infinite by an equivalent, more explicit, description.

```
def deg
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
    (a : s.F.obj X)
    {Y : C}
    (b : s.F.obj Y)
  : set ℤ
:= { n : ℤ | ∃ f : Y ⟶ X,
             (s.F.map f) b = n · a}

def is_weakly_flexible
    {C : Type*} [C_is_cat : category_theory.category C]
    (s : ffsn_setup C)
    {X : C}
    (a : s.F.obj X)
:= ∃ Y : C, ∃ b : s.F.obj Y,
   ∀ k : ℕ, ∃ n : ℤ, n ∈ deg s a b
                   ∧ |n| ≥ k
```

We have the straightforward norm estimate in terms of degrees:

```
lemma deg_estimate
      {C : Type*} [C_is_cat : category_theory.category C]
      (s : ffsn_setup C)
      {X : C}
      (a : s.F.obj X)
```

```
      {Y : C}
      (b : s.F.obj Y)
      (n : ℤ)
      (n_deg_ba : n ∈ deg s a b)
      (abs_n_pos : 0 < (|n| : real))
    : sn s a  ≤ sn s b / |↑n|
:=
begin
  -- we introduce notation
  let a' := lift_elt s a,
  let b' := lift_elt s b,
  -- hence: sn s a = ‖ a' ‖ and sn s b = ‖ b' ‖

  -- we extract a morphism witnessing the degree condition
  have ex_f_bna : ∃ f : Y ⟶ X, (s.F.map f) b = n • a, from
  begin
    dsimp only[deg] at n_deg_ba,
    assumption,
  end,
  rcases ex_f_bna with ⟨ f : Y ⟶ X, f_deg ⟩,

  -- using the monotonicity of f,
  -- we first prove a division-free version of the claim
  have n_times_thesis : |↑n| * ‖ a' ‖ ≤ ‖ b' ‖, from
  calc |↑n| * ‖ a' ‖ = ‖ n • a' ‖
                     : by rw s.Fsn_is_homogeneous
                 ... = ‖ lift_elt s ((s.F.map f) b) ‖
                     : by {congr,
                           simp only[f_deg], dsimp only[a'],
                           exact (lift_elt_zsmul s n a).symm}
                 ... = ‖ s.Fsn.map f b' ‖
                     : by {congr, exact (lift_elt_map s f b)}
                 ... ≤ ‖ b' ‖
                     : by exact fsn_est_explicit _ _ b',

  -- from the division-free version,
  -- we conclude the actual claim
  calc ‖ a' ‖ ≤ ‖ b' ‖ / |n|
         : by exact (le_div_iff' abs_n_pos).mpr n_times_thesis,
end
```

Vanishing of functorial semi-norms on weakly flexible classes It is now straightforward to formalise and prove the vanishing result Proposition 4.1.4. In order to conclude in the final step, we use the vanishing criterion for real numbers established in Section 2.4.

```
theorem weakly_flexible_zero_fsn
     {C : Type*} [C_is_cat : category_theory.category C]
     (s : ffsn_setup C)
     {X : C}
     (a : s.F.obj X)
     (a_is_weakly_flexible : is_weakly_flexible s a)
   : sn s a = 0
:=
begin
  let a' := lift_elt s a,
  -- thus, we need to show that ‖ a' ‖ = 0

  -- weak flexibility gives us a class b
  -- that has infinitely many degrees to a
  rcases a_is_weakly_flexible
         with ⟨ Y : C, ⟨ b : s.F.obj Y, deg_ba_infinite ⟩ ⟩,

  let c := sn s b,
  have c_geq_0 : 0 ≤ c,
       by {dsimp only[c], exact norm_nonneg (lift_elt s b)},

  -- from the degrees, we obtain the following estimate:
  have a_leq_c_over_n : ∀ n : nat,
          n > 0 → abs (‖ a' ‖) ≤ c / (n : real), from
  begin
    assume n : nat,
    assume n_pos : n > 0,

    -- we extract a suitable degree ...
    rcases (deg_ba_infinite n)
           with ⟨ m : ℤ, ⟨ m_is_deg, abs_m_geq_n ⟩⟩,

    -- ... make some basic observations ...
    have pos_n : 0 < (n : real),
         by exact nat.cast_pos.mpr n_pos,
    have n_leq_abs_m : (n : real) ≤ |↑m|,
         by {norm_cast at *, exact abs_m_geq_n},
    have pos_abs_m : 0 < (|m| : real),
         by exact gt_of_ge_of_gt n_leq_abs_m pos_n,

    -- ... and combine the estimates
    calc abs (‖ a' ‖)
             = ‖ a' ‖
             : by exact abs_norm_eq_norm a'
         ... ≤ ‖ lift_elt s b ‖ / |m|
```

```
            : by exact deg_estimate _ a b m m_is_deg pos_abs_m
        ... ≤ c / |m|
            : by refl
        ... ≤ c / (n : real)
            : by exact div_le_div_of_le_left c_geq_0 pos_n
                                              n_leq_abs_m,
  end,

  -- we conclude using the archimedean property of ℝ
  -- through zero_via_1_over_n
  show _,
       by exact zero_via_1_over_n (‖ a' ‖) c a_leq_c_over_n,
end
```

Representable functors only admit trivial finite functorial semi-norms In order to formalise and prove Corollary 4.1.5 on functorial semi-norms on representable functors, we first formalise the forgetful functor from `Ab` to `Type ...`; this corresponds to the forgetful functor Ab $\longrightarrow$ Set.

```
@[simp]
def forget_Ab := category_theory.forget Ab
```

Moreover, we make two preparations to unclutter the main proof:

For a representable functor to Set (or `Type ...`, respectively), every class is a push-forward of the universal class. This statement lifts also to functors to Ab (or `Ab`, respectively) that are representable when composed with the forgetful functor to Set (or `Type ...`, respectively).

```
lemma representable_rep
     {C : Type*} [C_is_cat : category_theory.category C]
     (F : C ⇒ Ab)
     (F_is_corep : category_theory.functor.corepresentable
                   (F ⋙ forget_Ab))
     {X : C}
     (a : F.obj X)
   : ∃ f : (F ⋙ forget_Ab).corepr_X ⟶ X,
     (F.map f) (F ⋙ forget_Ab).corepr_x = a
:=
begin
  let G := F ⋙ forget_Ab,
  let Y := G.corepr_X, -- "the" representing object
  let b := G.corepr_x, -- "the" universal element

  -- the underlying carriers of F X and G X coincide
  have eq_GX : G.obj X = (F.obj X).1, by refl,
  have eq_GY : G.obj Y = (F.obj Y).1, by refl,
```

```
  -- the morphism corresponding to a via representability ...
  let f := (G.corepr_w.app X).inv a,
  -- ... is the desired morphism, because ...
  use f,
  -- ... it maps the universal element to a
  show _, by
  calc F.map f b
         = G.map f b
         : by tauto
     ... = (G.corepr_w.app X).hom f
         : by exact (category_theory.functor.corepr_w_app_hom
                       G X f).symm
     ... = a
         : by simp,
end
```

Moreover, if a degree set between two classes is the set $\mathbb{Z}$ of all integers, then the target class clearly is weakly inflexible.

```
lemma deg_Z_weakly_flexible
     {C : Type*} [C_is_cat : category_theory.category C]
     (s : ffsn_setup C)
     {X : C}
     (a : s.F.obj X)
     {Y : C}
     (b : s.F.obj Y)
     (deg_ab_Z : deg s a b = { n : ℤ | tt })
   : is_weakly_flexible s a
:=
begin
  unfold is_weakly_flexible,
  use Y,
  use b,
  assume k : ℕ,

  use ↑k,
  finish,
end
```

Finally, we state and prove Corollary 4.1.5:

```
theorem representable_zero_fsn
     {C : Type*} [C_is_cat : category_theory.category C]
     (s : ffsn_setup C)
     (F_is_corep : category_theory.functor.corepresentable
                   (s.F ⋙ forget_Ab))
     {X : C}
```

```
    (a : s.F.obj X)
  : sn s a = 0
:=
begin
  let G := s.F ⋙ forget_Ab,

  -- we use the universal element for F (resp. G)
  -- to show that a is weakly flexible;
  let Y := category_theory.functor.corepr_X G,
           -- "the" representing object
  let b := category_theory.functor.corepr_x G,
           -- "the" universal element

  -- more precisely, we show that the degree set is all of ℤ
  have deg_ab_Z : deg s a b = { n : ℤ | tt }, from
  begin
    -- by definition, every degree is an integer
    have deg_sub_Z : deg s a b ⊆ { n : ℤ | tt }, from
    begin
      unfold deg,
      finish,
    end,

    -- conversely, universality shows
    -- that every integer can be realised as degree
    have Z_sub_deg : {n : ℤ | tt } ⊆ deg s a b, from
    begin
      assume n : ℤ,
      assume n_in_Z : n ∈ { m : ℤ | tt },

      simp only[deg, set.mem_set_of_eq],

      show ∃ (f : Y ⟶ X), (s.F.map f) b = n · a,
           by exact representable_rep s.F F_is_corep (n · a),
    end,

    show _,
         by exact set.subset.antisymm deg_sub_Z Z_sub_deg,
  end,

  -- we use the universal class as witness;
  -- because the degree set is all of ℤ,
  -- the weak flexiblity condition is clearly satisfied
  have a_is_weakly_flexible : is_weakly_flexible s a,
       by exact deg_Z_weakly_flexible s a b deg_ab_Z,
```

```
  -- thus, we can apply the theorem on vanishing
  -- of functorial semi-norms on weakly flexible classes
  show _,
       by exact weakly_flexible_zero_fsn s a
                  a_is_weakly_flexible,
end
```

4.2 Indirect Formalisation: Amenable Multiplicity

We give an example of an indirect formalisation of a recent approach in algebraic and geometric topology.

We will talk about CW-complexes, fundamental groups π_1, universal coverings, and (bounded) cohomology. But to understand the formalisation process, not much specific background knowledge is required: In Section 4.2.2, we will identify the core arguments. This backbone will be formalised in Section 4.2.3.

Mathematically, this technique corresponds to abstraction; implementation-wise, this technique is a form of prototyping. Similar approaches also have been used in other formalisation projects [51, 49].

Remark 4.2.1. For simplicity, we will ignore basepoints in the rest of this section; in our situation, this can be safely done because everything is sufficiently invariant/closed under conjugations. The detailed explanations would lead us too far astray.

4.2.1 Pen-and-Paper

Algebraic topology is concerned with invariants that measure the complexity of topological spaces. For example, one can ask for the minimal multiplicity that an open cover of a space needs to have if all members of the open cover are required to be "topologically simple." In particular, such multiplicity invariants appear in applied algebraic topology [62, 22, 56].

If one interprets "topologically simple" as the inclusion of the open sets into the ambient space being null-homotopic, this leads to the Lusternik–Schnirelmann multiplicity/category of topological spaces [20].

- Upper bounds for such multiplicity invariants can be given by exhibiting concrete "topologically simple" open covers of small multiplicity.
- In contrast, lower bounds require an understanding of *all* possible open covers by topologically simple subsets, which is a significantly harder task.

We will focus on amenable open covers, i.e., open covers whose members are constrained to an amenability condition on fundamental groups. Amenable open covers have been introduced by Gromov [25].

Definition 4.2.2 (amenable open cover)**.** Let X be a topological space. An *amenable open cover* of X is an open cover U of X such that each $V \in U$ is path-connected and such that the image of $\pi_1(V)$ in $\pi_1(X)$ under the map $\pi_1(V \hookrightarrow X)$ induced by the inclusion is an amenable group.

Examples of amenable groups [58, 45] include all finite groups and all Abelian groups. Moreover, the class of amenable groups is closed under subgroups, extensions, and directed unions. In particular, amenable subsets of spaces can be of a rather complicated topological shape. This means that finding meaningful lower bounds for the amenable multiplicity of spaces is difficult.

Gromov showed that the comparison map between bounded cohomology and ordinary (singular) cohomology provides lower bounds to the amenable multiplicity of topological spaces [25]. Another proof of this result via spectral sequences was given by Ivanov [31, 32]. *Bounded cohomology* $H_b^*(\,\cdot\,;\mathbb{R})$ of topological spaces is defined by taking cohomology of bounded singular cochains [25, 31]. Forgetting boundedness leads to a natural transformation $T\colon H_b^*(\,\cdot\,;\mathbb{R}) \Longrightarrow H^*(\,\cdot\,;\mathbb{R})$ from bounded cohomology to singular chomology; this natural transformation is called *comparison map*.

Theorem 4.2.3. *Let X be a path-connected CW-complex, let $n \in \mathbb{N}$, and let U be an amenable open cover of X with multiplicity* $\operatorname{mult} U \leq n$*. Then, the comparison map $T_X\colon H_b^n(X;\mathbb{R}) \longrightarrow H^n(X;\mathbb{R})$ is trivial.*

In other words: If the comparison map $H_b^n(X;\mathbb{R}) \longrightarrow H^n(X;\mathbb{R})$ is nontrivial, then X admits *no* amenable open cover with multiplicity at most n; i.e., in this case, the amenable muptlicity of X is at least $n+1$.

In the following, we give the main steps of an alternative proof of Theorem 4.2.3 via equivariant nerves and classifying spaces of families [48].

Lemma 4.2.4 (equivariant nerves [48, Section 4])**.** *Let X be a connected CW-complex and let U be an amenable open cover of X. Then the nerve N of the lifted cover of U (consisting of the connected components of the preimages under the universal covering map) on $\widetilde{X}$ has the following properties:*

1. *The nerve N admits a canonical $\pi_1(X)$-CW-structure whose isotropy groups all are amenable.*
2. *There exists a $\pi_1(X)$-equivariant nerve map $\widetilde{X} \longrightarrow N$.*

Proof of Theorem 4.2.3. We abbreviate $G := \pi_1(X)$. We pass to the equivariant setting by considering the universal covering $\widetilde{X}$ of X as a G-CW-complex (with respect to the lifted CW-structure and the deck transformation action). The universal covering map then induces natural isomorphisms

$$H^n(X;\mathbb{R}) \cong H^n_G(\widetilde{X};\mathbb{R}) \quad \text{und} \quad H^n_b(X;\mathbb{R}) \cong H^n_{G,b}(\widetilde{X};\mathbb{R})$$

between (bounded) cohomology of X and equivariant (bounded) cohomology of $\widetilde{X}$, which are compatible with the comparison maps. Thus, it remains to show that the comparison map $T_{G,\widetilde{X}} \colon H^n_{G,b}(\widetilde{X};\mathbb{R}) \longrightarrow H^n_G(\widetilde{X};\mathbb{R})$ is trivial.

The key is to consider classifying spaces of G: If F is a family of subgroups of G (i.e., a set of subgroups of G that is closed under conjugation and finite intersections and that contains the trivial subgroup), then there exists a G-CW-complex $E_F G$ with the following universal property [50]:

- All isotropy groups of $E_F G$ lie in F.
- For each G-CW-complex Y with isotropy in F there exists an up to G-homotopy unique G-map $Y \longrightarrow E_F G$.

We write EG for $E_{\{1\}}G$ (which is compatible with the classical notion of classifying spaces).

Let N be the G-CW-complex obtained from the nerve of the lifted cover of U and let $\nu \colon \widetilde{X} \longrightarrow N$ be a G-equivariant nerve map (Lemma 4.2.4).

Because N has isotropy in the family Am of amenable subgroups of G (Lemma 4.2.4), there exists a classifying map $c_N \colon N \longrightarrow E_{\mathsf{Am}}G$. Moreover, as $\widetilde{X}$ and EG are free G-spaces and the trivial group is amenable, we obtain classifying maps $c_{\widetilde{X}} \colon \widetilde{X} \longrightarrow EG$ and $c_{G,\mathsf{Am}} \colon EG \longrightarrow E_{\mathsf{Am}}G$.

The isotropy groups of $\widetilde{X}$ are amenable. Therefore, the universal property of $E_{\mathsf{Am}}G$ shows that the following diagram is commutative up to G-homotopy:

$$\begin{array}{ccc}
\widetilde{X} & = & \widetilde{X} \\
\Big\downarrow{\scriptstyle \nu} & & \Big\downarrow{\scriptstyle c_{\widetilde{X}}} \\
N & \xrightarrow[c_N]{} & EG \\
 & & \Big\downarrow{\scriptstyle c_{G,\mathsf{Am}}} \\
 & & E_{\mathsf{Am}}G
\end{array}$$

We now apply the functor $H^n_{G,b}(\,\cdot\,;\mathbb{R})$ to this diagram and extend the diagram with the comparison map (Figure 4.1).

This diagram commutes: The left rectangle commutes because $H^n_{G,b}(\,\cdot\,;\mathbb{R})$ is G-homotopy invariant; the right rectangle commutes by the naturality of the comparison map. We use the following observations:

- By the mapping theorem, $H^n_{G,b}(c_{\widetilde{X}}) \colon H^n_{G,b}(EG;\mathbb{R}) \longrightarrow H^n_{G,b}(\widetilde{X};\mathbb{R})$ is an isomorphism [25].

  Moreover, $H^n_{G,b}(c_{G,\mathsf{Am}}) \colon H^n_{G,b}(E_{\mathsf{Am}}G;\mathbb{R}) \longrightarrow H^n_{G,b}(EG;\mathbb{R})$ is an isomorphism [48, Proposition 5.2].

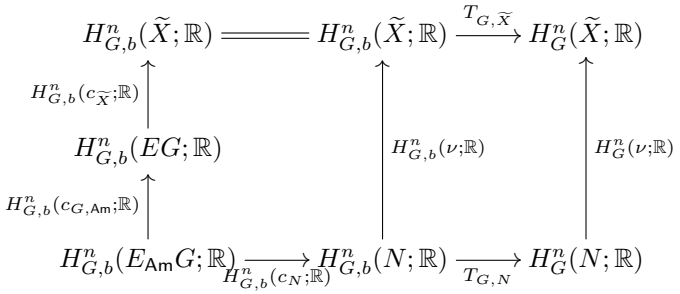


Figure 4.1: Applying (bounded) cohomology to the commutative diagram of G-CW-complexes

- Finally, $\dim N = \operatorname{mult} U - 1 \leq n - 1$, and so $H^n_G(N;\mathbb{R}) \cong 0$, because (equivariant) singular cohomology respects the dimension of CW-complexes.

Therefore, $T_{G,\widetilde{X}}$ factors over 0 and so is the trivial map. □

Another lower bound for the amenable multiplicity of aspherical spaces is given by L^2-Betti numbers [60, 61]. A benefit of the approach taken in Section 4.2.2 is that it will also encompass this result on L^2-Betti numbers.

4.2.2 Abstraction

The lower bound for amenable multiplicity presented in Section 4.2.1 involves several objects and notions from equivariant algebraic topology such as fundamental groups, the universal covering, G-CW-complexes, equivariant (bounded) cohomology, ... We will avoid implementing these notions concretely by identifying the core arguments and abstracting over the concrete setting. The abstract lower multiplicity bounds in this setting are formulated in Theorem 4.2.9 (for a single functor) and in Theorem 4.2.14 (for a pair of functors connected through a natural transformation). At the end of the section, we will indicate how to apply these statements.

Abstracted setup We begin by collecting the categories and properties needed for our arguments. We assume that we have the following (adding the standard interpretation in parentheses):

- For each group G, a category $\mathsf{CW}_{\mathsf{h}}^G$ (e.g., the homotopy category of G-CW-complexes);
- A map $\dim\colon \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^G) \longrightarrow \mathbb{N} \cup \{\infty\}$ (e.g., the dimension);

- For each group G, a notion of "families of subgroups of G" (e.g., sets of subgroups of G that are closed under conjugation, finite intersection, and that contain the trivial subgroup);
- For each group G, the trivial family 1 of subgroups of G (e.g., consisting only of the trivial subgroup);
- For each group G, each family F of subgroups of G, and each $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^{G})$, a predicate that determines whether X "has isotropy in F" or not (e.g., if all isotropy groups of X lie in F); objects with isotropy in the trivial family 1 have isotropy in every other family as well;
- For each group G and each family F of subgroups of G, a classifying space $E_F G$, i.e., an object in $\mathsf{CW}_{\mathsf{h}}^{G}$ with isotropy in F that is terminal in $\mathsf{CW}_{\mathsf{h}}^{G}$ among all objects with isotropy in F (e.g., classifying spaces for families of subgroups [50]);

  The universal maps given by such classifying spaces will be denoted by $c_{\dots}$ and called *classifying maps*;

  We abbreviate $EG := E_1 G$;
- A category $\mathsf{CW}_{\mathsf{h}}^{1}$ of (connected) CW-complexes and a map of the type $\pi_1 \colon \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^{1}) \longrightarrow \mathrm{Ob}(\mathsf{Group})$ (e.g., the fundamental group with respect to an implicit basepoint);
- For each $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^{1})$ an object $\widetilde{X} \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^{\pi_1(X)})$ that has isotropy in 1 (e.g., the universal covering of X);
- A notion of open F-covers of objects X in $\mathsf{CW}_{\mathsf{h}}^{1}$ with isotropy in a given family F of subgroups of $\pi_1(X)$; (e.g., open covers by path-connected subsets such that the images on π_1 (up to conjugation) lie in the given family);
- A notion of multiplicity of such open covers (e.g., the geometric multiplicity);
- A notion of equivariant nerve: If $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^{1})$ and if U is an open F-cover of X, then the equivariant nerve is an object N of $\mathsf{CW}_{\mathsf{h}}^{\pi_1(X)}$
  - with isotropy in F,
  - with dimension equal to the multiplicity of U minus 1,
  - and with a nerve map, i.e., a morphism $\nu \colon \widetilde{X} \longrightarrow N$ in $\mathsf{CW}_{\mathsf{h}}^{\pi_1(X)}$.

  (e.g., the nerve of the lifted cover on $\widetilde{X}$; Lemma 4.2.4);

In the following, we will work with functors of the following type. For simplicity, we formulate everything for covariant functors. The corresponding statements for the contravariant case can then be obtained by considering functors to the opposite category of C. As we are interested in vanishing results, our target category is assumed to have a zero object.

Setup 4.2.5. Let G be a group, let C be a category with a zero object, and let $H\colon \mathsf{CW}_{\mathsf{h}}^G \longrightarrow C$ be a (covariant) functor. Moreover, let $n \in \mathbb{N}$.

In the proof of Theorem 4.2.3, we used several properties of (equivariant) bounded cohomology and (equivariant) cohomology. These properties can be abstracted as follows:

Definition 4.2.6 (H-aspherical space)**.** In the situation of Setup 4.2.5, an object $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ is *H-aspherical* if $\pi_1(X) = G$ and $H(c_{\widetilde{X}})\colon H(\widetilde{X}) \longrightarrow H(EG)$ is an isomorphism.

Definition 4.2.7 (admissible family)**.** In the situation of Setup 4.2.5, a family F of subgroups of G is *strongly H-admissible* if $H(c_{G,F})\colon H(EG) \longrightarrow H(E_FG)$ is an isomorphism.

Definition 4.2.8 (dim-vanishing)**.** In the situation of Setup 4.2.5, we say that H is *dim-vanishing (for n)* if the following holds: For all $Y \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^G)$ with $\dim Y + 1 \leq n$, we have

$$H(Y) \cong_C 0.$$

Abstracted statements and proofs Then, a first version of the lower multiplicity bound looks as follows:

Theorem 4.2.9 (lower multiplicity bound)**.** *In the situation of Setup 4.2.5, let H be dim-vanishing for n, let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ be H-aspherical, let F be a strongly H-admissible family of subgroups of G, and let U be an open F-cover of X with multiplicity at most n. Then*

$$H(\widetilde{X}) \cong_C 0.$$

To improve modularity, we organise the main arguments of the proof into separate statements:

Lemma 4.2.10 (the key commutative diagram)**.** *Let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$, let $G := \pi_1(X)$, let F be a family of subgroups of G, and let U be an open F-cover of X. Let N be the equivariant nerve of U. Then the following diagram is commutative in $\mathsf{CW}_{\mathsf{h}}^G$:*

$$\begin{array}{ccc}
\widetilde{X} & = & \widetilde{X} \\
\Big\downarrow{\scriptstyle \nu} & & \Big\downarrow{\scriptstyle c_{\widetilde{X}}} \\
 & & EG \\
 & & \Big\downarrow{\scriptstyle c_{G,F}} \\
N & \xrightarrow[c_N]{} & E_FG
\end{array}$$

Proof. The three classifying maps indicated in the diagram all exist because N has isotropy in F, because $\widetilde{X}$ has trivial isotropy and EG has trivial isotropy (whence also isotropy in F).

The universal covering $\widetilde{X}$ has trivial isotropy and thus isotropy in the family F. Because the classifying space $E_F G$ is terminal in $\mathsf{CW}_{\mathsf{h}}^G$ among objects with isotropy in F, both routes from $\widetilde{X}$ to $E_F G$ through the diagram result in the same morphism (namely the classifying map $\widetilde{X} \longrightarrow E_F G$). □

Proposition 4.2.11 (factorisation of an admissible functor over the nerve)**.** *In the situation of Setup 4.2.5, let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ be H-aspherical, let F be a strongly H-admissible family of subgroups of G, and let U be an open F-cover of X. Let N be the equivariant nerve of U. Then $\mathrm{id}_{H(X)}$ factors over $H(N)$ in the category C.*

Proof. Applying the functor H to the commutative diagram in $\mathsf{CW}_{\mathsf{h}}^G$ from Lemma 4.2.10, leads to the following commutative diagram in the target category C:

$$\begin{array}{ccc} H(\widetilde{X}) & = & H(\widetilde{X}) \\ \Big\downarrow{\scriptstyle H(\nu)} & & \Big\downarrow{\scriptstyle \cong_C\ H(c_{\widetilde{X}})} \\ & & H(EG) \\ & & \Big\downarrow{\scriptstyle \cong_C\ H(c_{G,F})} \\ H(N) & \xrightarrow[H(c_N)]{} & H(E_F G) \end{array}$$

Because X is H-aspherical and the family F is strongly H-admissible, the right vertical morphisms are isomorphism. This gives the desired factorisation of $\mathrm{id}_{H(X)}$ over $H(N)$. □

Lemma 4.2.12 (dim-vanishing functors on small nerves)**.** *In the situation of Setup 4.2.5, let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ satisfy $\pi_1(X) = G$, let F be a family of subgroups of G, let U be an F-cover of X with $\operatorname{mult} U \leq n$, and let H be dim-vanshing for n. Then the equivariant nerve N of U satisfies*

$$H(N) \cong_C 0.$$

Proof. We have

$$\dim N \leq \operatorname{mult} U - 1 \leq n - 1 < n.$$

Because H is dim-vanishing for n, we obtain $H(N) \cong_C 0$. □

Proof of Theorem 4.2.9. As H is dim-vanishing for n, we have $H(N) \cong_C 0$ (Lemma 4.2.12).

In view of Proposition 4.2.11, the identity morphism of $H(X)$ factors over $H(N) \cong_C 0$. Therefore, $H(X) \cong_C 0$. □

The lower-bound aspect of Theorem 4.2.9 is more visible when translated into the following form:

Corollary 4.2.13 (lower multiplicity bound, contrapositive version)**.** *In the situation of Setup 4.2.5, let H be dim-vanishing for n, let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ be H-aspherical, let F be a strongly H-admissible family of subgroups of G, and let U be an open F-cover of X. If $H(\widetilde{X}) \ncong_C 0$, then*

$$\operatorname{mult} U > n.$$

Proof. This lower bound follows from Theorem 4.2.9 by contraposition. □

The range of applicability of Theorem 4.2.9 can be improved by splitting the hypotheses on the functor over two functors: One that satisfies the asphericity and admissibility condition and one that satisfies the dim-vanishing condition. In fact, this is the version that we are aiming for.

Theorem 4.2.14 (lower multiplicity bound, refined version)**.** *In the situation of Setup 4.2.5, let $X \in \mathrm{Ob}(\mathsf{CW}_{\mathsf{h}}^1)$ be H-aspherical, let F be a strongly H-admissible family of subgroups of G, and let U be an open F-cover of X of multiplicity at most n. Moreover, let $K\colon \mathsf{CW}_{\mathsf{h}}^G \longrightarrow C$ be a functor that is dim-vanishing for n and let $T\colon H \Longrightarrow K$ be a natural transformation. Then, $T_X\colon H(\widetilde{X}) \longrightarrow K(\widetilde{X})$ is the zero morphism.*

Proof. Applying H to the commutative diagram from Lemma 4.2.10 and extending by the natural transformation T, we obtain the following commutative diagram in C:

$$\begin{array}{ccccc}
K(\widetilde{X}) & \xrightarrow{T_{\widetilde{X}}} & H(\widetilde{X}) & = & H(\widetilde{X}) \\
 & & & & \cong_C \downarrow H(c_{\widetilde{X}}) \\
\downarrow K(\nu) & & \downarrow H(\nu) & & H(EG) \\
 & & & & \cong_C \downarrow H(c_{G,F}) \\
K(N) & \xrightarrow[T_N]{} & H(N) & \xrightarrow[H(c_N)]{} & H(E_F G)
\end{array}$$

As K is dim-vanishing for n, we obtain $K(N) \cong_C 0$ (Lemma 4.2.12). Therefore, the diagram shows that $T_{\widetilde{X}}$ factors over zero and thus is the zero morphism. □

Clearly, Theorem 4.2.9 can be obtained as a special case of Theorem 4.2.14, using the identity transformation $H \Longrightarrow H$. More generally, one could distribute the three properties asphericity, admissibility, and dim-vanishing over three functors. As we have no applications for the fully generalised setting, we did not add this triple version.

For example, Theorem 4.2.9 and Theorem 4.2.14 can be instantiated to the following special cases [48, 43], where $\mathsf{CW}_{\mathsf{h}}^G$, π_1, etc. carry the usual topological meaning:

Bounded cohomology We obtain the result of Theorem 4.2.3 by applying Theorem 4.2.14 to the following setting:

- Let C be the opposite category of the category of $\mathbb{R}$-vector spaces.
- Let $H := H^n_{G,b}(\,\cdot\,;\mathbb{R})\colon \mathsf{CW}^G_{\mathsf{h}} \longrightarrow C$.
- Let F be the family of amenable subgroups of G (or, more generally, a family consisting of uniformly boundedly acyclic subgroups of G).
- Let X be a connected CW-complex.
- Let $K := H^n_G(\,\cdot\,;\mathbb{R})\colon \mathsf{CW}^G_{\mathsf{h}} \longrightarrow C$ and let $T\colon H \Longrightarrow K$ be the comparison map from (equivariant) bounded cohomology to (equivariant) cohomology.

Then the hypotheses of Theorem 4.2.14 are indeed satisfied [48, 43].

L^2-Betti numbers We obtain Sauer's vanishing result for L^2-Betti numbers by applying Theorem 4.2.9 to the following setting:

- Let C be the category of NG-modules localised at dimension isomorphisms, where NG denotes the group von Neumann algebra of G.
- Let $H := H^G_n(\,\cdot\,;NG)\colon \mathsf{CW}^G_{\mathsf{h}} \longrightarrow C$ be the corresponding version of L^2-homology.
- Let F be the family of amenable subgroups of G (or, more generally, a family consisting of L^2-acyclic subgroups of G).
- Let X be a connected aspherical CW-complex.

Then, the hypotheses of Theorem 4.2.9 are indeed satisfied [43, Section 7].

The abstraction provided by Theorem 4.2.9 also applies to the relative setting (by not taking the literal obvious topological interpretations of $\mathsf{CW}^G_{\mathsf{h}}$, etc., but instead going into a relative equivariant setup) [43] and might be applicable to other interesting cases in the future.

4.2.3 Lean

We implement the backbone from Section 4.2.2.

Source code 4.2.15. This is `open_covers.lean` of the git repo [47].

Instead of giving concrete formalisations of the category of equivariant CW-complexes, etc., we only introduce corresponding placeholders in the form of constants and axioms and connect them through the relevant properties.

Caveat 4.2.16. It is tempting to introduce missing pieces with `sorry`. This works well for `Prop`-valued pieces (by propositional extensionality).

However, constants/axioms/assumptions of other types should *not* be prototyped with `sorry`, because any two `sorry`-terms of the same type will be considered equal by Lean:

```
def x : nat := sorry
def y : nat := sorry

lemma x_is_y
    : x = y
:=
begin
  refl
end
```

In particular, this means that when later replacing `sorry` by a concrete definition, the corresponding statements might not be provable anymore; this would break the prototyping process.

Therefore, it is more advisable to use `constant` or `axiom` instead when postulating the existence of objects etc. Of course, one still has to be careful not to introduce inconsistent assumptions!

We import groups and basic category theory; moreover, as many parts will be `noncomputable`, we declare the whole theory as `noncomputable`.

```
import tactic -- standard proof tactics
import algebra.category.Group.basic -- the category of groups
import group_theory.subgroup.basic  -- subgroups
import category_theory.category.basic -- basic category theory
import category_theory.functor -- functors
import category_theory.limits.shapes.zero -- categories with
    zero objects/morphisms
import data.nat.enat -- extended naturals

open classical -- we work in classical logic
open category_theory

noncomputable theory
```

Our setup requires us to speak about constructions that assign a category to every group and also a group to every "CW-complex". Therefore, we need to limit the universe levels of these constructions. Mostly, we will only deal with the single universe level `u`.

```
universes u v
```

Classifying spaces For a given group G, we introduce a name for the homotopy category of G-CW-complexes. Moreover, each G-CW-complex has a

dimension in the extended natural numbers. None of this is given an actual implementation; we only postulate the existence of such things.

```
constant CWh (G : Type u) [group G] : Type u
constant dim (G : Type u) [group G] : CWh G → enat

@[instance]
axiom CWh_cat (G : Type u) [group G] : (category.{u u} (CWh G))
```

Moreover, we postulate a function that assigns to each group the corresponding type of all families of subgroups. If one were to give an actual implementation of this concept, one could realise `family` as a record (parametrised over groups), having fields for the corresponding set of subgroups, and for the properties of being conjugation-closed, finite intersection-closed, and for containing the trivial subgroup. Moreover, we introduce a constant for the trivial family, consisting only of the trivial subgroup.

```
constant family (G: Type u) [group G] : Type u
constant one (G : Type u) [group G] : family G
```

Families of subgroups can arise as isotropy groups of equivariant CW-complexes. Thus, we postulate a predicate that checks whether the given equivariant CW-complex has isotropy in the given family. If an equivariant CW-complex has trivial isotropy, then it has isotropy in every other family.

```
constant has_isotropy_in
    {G : Type u} [group G]
    (F : family G)
    (X : CWh G)
  : Prop

axiom isotropy_one_implies_all
    {G : Type u} [group G]
    (F : family G)
    (X : CWh G)
    (X_is_free : has_isotropy_in (one G) X)
  : has_isotropy_in F X
```

We can now formalise the notion of a classifying space of a family of subgroups of a group G as being terminal among all G-CW-complexes with isotropy in the given family. The universal property of terminality is expressed by the fact that the type `X ⟶ space` of morphisms (envisioned as G-homotopy classes of G-maps) has a unique member.

```
structure classifying_space
    (G : Type u) [group G]
    (F : family G)
:= mk :: (space : CWh G)
         (isotropy : has_isotropy_in F space)
```

```
        (universal_property : ∀ X : CWh G,
                              has_isotropy_in F X
                              → unique (X ⟶ space))
```

Postulating existence of such classifying spaces is done by postulating a function constructing such classifying spaces.

```
constant E
    (G : Type u) [group G]
    (F : family G)
  : classifying_space G F
```

We introduce notation for the classifying maps (as the corresponding unique morphism granted by the universal property).

```
def cm
    (G : Type u) [group G]
    (F : family G)
    (X : CWh G)
    (X_has_isotropy_in_F : has_isotropy_in F X)
  : X ⟶ (E G F).space
:= ((E G F).universal_property X X_has_isotropy_in_F).
    to_inhabited.default
```

In addition, we set up abbreviations for the case of the trivial family of subgroups.

```
def E1
    (G : Type u) [group G]
  : classifying_space G (one G)
:= E G (one G)

lemma E1_isotropy
    (G : Type u) [group G]
    (F : family G)
  : has_isotropy_in F (E1 G).space
:=
begin
  exact isotropy_one_implies_all F
          (E G (one G)).space
          (E G (one G)).isotropy
end

def cm1_E
    (G : Type u) [group G]
    (F : family G)
  : (E G (one G)).space ⟶ (E G F).space
:= cm G F (E G (one G)).space
          (E1_isotropy G F)
```

Restricted covers and their equivariant nerves We postulate the existence of the category of CW-complexes (connected, with an implicit basepoint) and continuous maps and that we can assign a fundamental group to each such object (with respect to the implicit basepoint). As we do not want to speak of base-point preserving maps, we will not assume that the fundamental group is a functor.

```
constant CW1 : Type u

@[instance]
axiom CW1_cat : category_theory.category.{u u} CW1

constant pi_1 (X : CW1.{u}) : Type u

@[instance]
axiom fundamental_group (X : CW1.{u}) : group (pi_1 X)
```

Moreover, we postulate the existence of universal coverings, but we only restrict to the relevant features for us: The universal covering of an object `X` in `CW1` is an object of `CWh (pi_1 X)` with trivial isotropy (whence the isotropy lies in any subgroup family of `pi_1 X`). In particular, we obtain a corresponding classifying map.

```
constant universal_covering (X : CW1) : CWh (pi_1 X)

axiom universal_covering_is_free (X : CW1)
  : has_isotropy_in (one (pi_1 X)) (universal_covering X)

lemma universal_covering_isotropy
    (X : CW1)
    (F : family (pi_1 X))
  : has_isotropy_in F (universal_covering X)
:=
begin
  exact isotropy_one_implies_all
          F
          (universal_covering X)
          (universal_covering_is_free X),
end

def cm1_universal_covering
    (X : CW1)
  : universal_covering X ⟶ (E1 (pi_1 X)).space
:=
cm (pi_1 X) (one (pi_1 X))
   (universal_covering X)
   (universal_covering_is_free X)
```

We postulate a function that assigns to each CW-complex the type of all open covers with restricted isotropy – envisioned as covers by path-connected open subsets such that the images of the fundamental groups are "conjugate to" elements of the given family (the latter could be done in our strange implicit-basepoint setting!). Again, in a concrete implementation, this could be realised as a suitable record type.

```
constant open_cover
    (X : CW1)
    (F : family (pi_1 X))
  : Type u
```

Such open covers are supposed to have a multiplicity (in the extended natural numbers) and a nerve of the corresponding lifted cover (whose dimension can be expressed in terms of the multiplicity) as well as a nerve map. The essential connection between restricted open covers and the nerves of their lifted covers is encoded in the isotropy condition in `nerve_isotropy`.

```
constant open_cover
    (X : CW1)
    (F : family (pi_1 X))
  : Type u

constant multiplicity
    {X : CW1}
    {F : family (pi_1 X)}
    (U : open_cover X F)
  : enat

constant nerve
    {X : CW1}
    {F : family (pi_1 X)}
    (U : open_cover X F)
  : CWh (pi_1 X)

constant nerve_map
    {X : CW1}
    {F : family (pi_1 X)}
    (U : open_cover X F)
  : (universal_covering X) ⟶ (nerve U)

axiom nerve_dim
    {X : CW1}
    {F : family (pi_1 X)}
    (U : open_cover X F)
  : dim (pi_1 X) (nerve U) + 1 = multiplicity U
```

```
axiom nerve_isotropy
    {X : CW1}
    {F : family (pi_1 X)}
    (U : open_cover X F)
  : has_isotropy_in F (nerve U)
```

The lower bound strategy We are now prepared to implement the actual lower bound strategy. In the pen-and-paper version, we could organise many arguments neatly in terms of commutative diagrams. In the implementation, we will replace these diagrams by the corresponding equations.

As first step, we formalise the key commutative diagram on the level of G-CW-complexes (Lemma 4.2.10):

```
lemma commutative_diagram
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
  : (nerve_map U)                  -- nerve map: ucov X ⟶ nerve
    ≫
    (cm (pi_1 X) F (nerve U)       -- nerve ⟶ E_F G
        (nerve_isotropy U))
  = (cm1_universal_covering X)     -- ucov X ⟶ EG
    ≫
    (cm1_E (pi_1 X) F)             -- EG ⟶ E_F G
:=
begin
  -- notation for the relevant classifying space
  let EFG := E (pi_1 X) F,
  -- the universal property of E_F G with domain ucov X:
  -- there exists a unique morphism ucov X ⟶ E_F G
  have unique_X_EFG :
       unique (universal_covering X ⟶ EFG.space),
       by exact EFG.universal_property (universal_covering X)
                  (universal_covering_isotropy X F),

  -- the two maps are morphisms ucov X ⟶ E_F G
  let f1 : universal_covering X ⟶ EFG.space
         := (nerve_map U)
            ≫
            (cm (pi_1 X) F (nerve U)
                (nerve_isotropy U)),
  let f2 : universal_covering X ⟶ EFG.space
         := (cm1_universal_covering X)
            ≫
            (cm1_E (pi_1 X) F),
```

```
  -- therefore, the universal property of E_F G
  -- proves the claim
  calc f1 = unique_X_EFG.to_inhabited.default
          : by exact @unique.eq_default _ unique_X_EFG f1
      ... = f2
          : by exact @unique.default_eq _ unique_X_EFG f2,
end
```

The lower bound strategy consists of applying functors to this commutative diagram. Therefore, we first formalise the axiomatisation of such functors. The target category is a category with a zero object.

```
constant C : Type u

@[instance]
constant C_is_cat : category.{u u} C

@[instance]
constant C_has_zero
: category_theory.limits.has_zero_object.{u u} C

-- allows us to use 0 for the zero object C_has_zero.zero
instance : has_zero C
:= category_theory.limits.has_zero_object.has_zero
-- and to use 0 for the zero morphisms
instance : category_theory.limits.has_zero_morphisms C
:= category_theory.limits.has_zero_object.
    zero_morphisms_of_zero_object
```

In this setting, we can simply translate the definitions from Section 4.2.2:

```
def is_aspherical_for
    (X : CW1)
    (H : CWh (pi_1 X) ⇒ C)
:= is_iso (H.map (cm1_universal_covering X))

def is_admissible_family_for
    {G : Type u} [group G]
    (F : family G)
    (H : CWh G ⇒ C)
:= is_iso (H.map (cm1_E G F))

def dim_vanishing
    {G : Type u} [group G]
    (H : CWh G ⇒ C)
    (n : nat)
:= ∀ Y : CWh G, dim G Y + 1 ≤ n → (H.obj Y ≅ 0)
```

```
structure admissible_functor
    (X : CW1)
    (F : family (pi_1 X))
:= mk :: (functor  : (CWh (pi_1 X)) ⇒ C )
         (ucov_iso : is_aspherical_for X functor)
                     -- iso for ucov X → EG
         (family_admissible : is_admissible_family_for
                              F functor)
                              -- iso for EG → EFG
```

As next step, we apply admissible functors to the commutative diagram, obtaining the factorisation over the value on the equivariant nerve (Proposition 4.2.11).

```
lemma admissible_functors_factor_over_nerve
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
    (H : admissible_functor X F)
  : ∃ g : H.functor.obj (nerve U)
            ⟶ H.functor.obj (universal_covering X),
          (H.functor.map (nerve_map U)) ≫ g
        = 𝟙 (H.functor.obj (universal_covering X))
:=
begin
  let G := pi_1 X,
  let HX : C := H.functor.obj (universal_covering X),
  let EG  := E1 (pi_1 X),
  let EFG := E (pi_1 X) F,
  let HN : C := H.functor.obj (nerve U),

  -- the commutative diagram:
  -- both ways of getting from ucov X to E_FG coincide
  let f11 : universal_covering X ⟶ (nerve U)
          := nerve_map U,
  let f12 : nerve U ⟶ EFG.space
          := cm (pi_1 X) F (nerve U) (nerve_isotropy U),

  let f21 : universal_covering X ⟶ EG.space
          := cm1_universal_covering X,
  let f22 : EG.space ⟶ EFG.space
          := cm1_E G F,

  have comm_diag : f11 ≫ f12 = f21 ≫ f22,
       by exact commutative_diagram X F U,
```

```
-- using the commutative diagram,
-- we show that id HX factors over HN;
-- More precisely:
   -- id_HX =  H f11
   --       ≫ H f12 ≫ inv (H (f21 ≫ f22))
-- the first part of the factorisation:
let f : HX ⟶ HN
      := H.functor.map f11,

-- the second part of the factorisation:
-- preparation:
-- admissibility of H with respect to F
-- shows that H (f21 ≫ f22) is an iso
have H_f2_is_iso : is_iso (H.functor.map (f21 ≫ f22)), from
begin
  -- H (f21 ≫ f22) = H f21 ≫ H f22 is an iso,
  -- because H f21 and H f22 are isos
  have H_f2_comp : H.functor.map (f21 ≫ f22)
       = (H.functor.map f21) ≫ (H.functor.map f22),
       by simp,

  have : is_iso (H.functor.map f21 ≫ H.functor.map f22),
       by exact @is_iso.comp_is_iso C C_is_cat _ _ _
                    (H.functor.map f21) (H.functor.map f22)
                    H.ucov_iso H.family_admissible,

  show _, by {simp only[H_f2_comp], assumption},
end,

let i_H_f2 := @inv C C_is_cat _ _
                   (H.functor.map (f21 ≫ f22)) H_f2_is_iso,
let g : HN ⟶ HX
      := H.functor.map f12 ≫ i_H_f2,

-- the computation that id HX indeed factors over HN:
have id_HX_factors_over_HN : f ≫ g  = 𝟙 HX, from
calc f ≫ g = H.functor.map f11
             ≫ H.functor.map f12 ≫ i_H_f2
             : by simp -- by definition
       ... = H.functor.map (f11 ≫ f12)
             ≫ i_H_f2
             : by simp -- nz ≫ zn cancels and functoriality
       ... = H.functor.map (f21 ≫ f22)
             ≫ i_H_f2
```

```
            : by simp[comm_diag]
              -- finally, we use the commutative diagram
        ... = 𝟙 HX
            : by exact @is_iso.hom_inv_id C C_is_cat _ _
                       (H.functor.map (f21 ≫ f22))
                       H_f2_is_iso,

  -- thus, f and g witness the claimed factorisation
  show _,
       by {use g, exact id_HX_factors_over_HN},
end
```

In the presence of the dim-vanishing property, the value on the equivariant nerve is zero if the multiplicity of the given open cover is small (Lemma 4.2.12):

```
lemma dim_vanishing_functors_vanish_on_small_nerves
    (n : nat)
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
    (mult_U_leq_n : multiplicity U ≤ n)
    (H : CWh (pi_1 X) ⇒ C)
    (H_dim_vanishing : dim_vanishing H n)
  : H.obj (nerve U) ≅ 0
:=
begin
  -- simplified notation:
  let G := pi_1 X,
  let HN : C := H.obj (nerve U),

  -- the dimension of the nerve is smaller than n
  have dim_N_leq_n : dim G (nerve U) + 1 ≤ n, by
  calc dim G (nerve U) + 1 = multiplicity U
                           : by exact nerve_dim U
                       ... ≤ n
                           : by exact mult_U_leq_n,
  -- thus, we apply the dim-vanishing property
  exact H_dim_vanishing (nerve U) dim_N_leq_n,
end
```

As final preparation before the main proof, we record the fact that if the identity of an object factors over zero, then the object is zero.

```
lemma id_factors_over_zero_iso_zero
    (X : C)
    (f : X ⟶ 0)
```

```
    (g : 0 ⟶ X)
    (id_factors : 𝟙 X = f ≫ g)
  : X ≅ 0
:=
begin
  -- we use f and g as isomorphisms
  exact { hom := f,
          inv := g,
          hom_inv_id' := _,
          inv_hom_id' := _},
  -- f ≫ g = id by assumption:
  exact id_factors.symm,
  -- g ≫ f = id by the universal property of the zero object
  ext1,
end
```

After this preparation, we can assemble the first version of the lower bound strategy (Theorem 4.2.9):

```
theorem restricted_cover_lower_bound
    (n : nat)
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
    (mult_U_leq_n : multiplicity U ≤ n)
    (H : admissible_functor X F)
    (H_dim_vanishing : dim_vanishing H.functor n)
  : H.functor.obj (universal_covering X) ≅ 0
:=
begin
  -- we first show that H(nerve) ≅ 0,
  -- using the dim-vanishing property
  let HN : C := H.functor.obj (nerve U),

  have HN_is_zero : HN ≅ 0,
       by exact dim_vanishing_functors_vanish_on_small_nerves
                  n X F U mult_U_leq_n
                  H.functor H_dim_vanishing,

  -- the resulting isomorphisms to and from the zero object
  let nz : HN ⟶ 0 := HN_is_zero.hom,
  let zn : 0 ⟶ HN := HN_is_zero.inv,

  -- second, the preparation on classifying spaces shows that
  -- the identity on H(ucov X) factors over H(nerve)
  let HX := H.functor.obj (universal_covering X),
```

```
  let f  := H.functor.map (nerve_map U),

  -- we choose such a factorisation
  choose g fg_idHX
         using admissible_functors_factor_over_nerve X F U H,

  let fz : HX ⟶ 0 := f ≫ nz,
  let zg : 0 ⟶ HX := zn ≫ g,

  -- thus, the identity on H(ucov X) factors over 0
  have id_HX_factors_over_0 : fz ≫ zg = 𝟙 HX, from
  calc fz ≫ zg = f ≫ nz ≫ zn ≫ g
                 : by simp
           ... = f ≫ g
                 : by simp -- nz ≫ zn cancels
           ... = 𝟙 HX
                 : by exact fg_idHX,

  -- hence, H(ucov X) must be zero
  show HX ≅ 0,
       by exact id_factors_over_zero_iso_zero HX fz zg
                 id_HX_factors_over_0.symm,
end
```

Similarly, also the contrapositive version (Corollary 4.2.13) admits a straightforward translation: We first note down basics on non-isomorphic objects and then prove the contrapositive version by contradiction; proofs by contradiction are initated by `by_contradiction`.

```
def not_isomorphic
    {C : Type v}
    [category.{u v} C]
    (X : C)
    (Y : C)
:= ¬ ∃ f : X ⟶ Y, is_iso f

lemma iso_implies_not_not_isomorphic
    {C : Type v}
    [category.{u v} C]
    (X : C)
    (Y : C)
    (X_iso_Y : X ≅ Y)
   : ¬ not_isomorphic X Y
:=
begin
  -- X ≅ Y shows that there exists an isomorphism X ⟶ Y
```

```
  have ex_iso : ∃ f : X ⟶ Y, is_iso f, from
  begin
    let f := X_iso_Y.hom,
    have f_is_iso : is_iso f,
         by exact category_theory.is_iso.of_iso X_iso_Y,
    use f,
    show _, by exact f_is_iso,
  end,

  -- thus, there does not not exist an iso
  unfold not_isomorphic,
  exact not_not.mpr ex_iso,
end

theorem restricted_cover_lower_bound_cp
    (n : nat)
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
    (H : admissible_functor X F)
    (H_dim_vanishing : dim_vanishing H.functor n)
    (H_non_zero : not_isomorphic
                    (H.functor.obj (universal_covering X))
                    0)
  : multiplicity U > n
:=
begin
  -- we apply the theorem and convert the contraposition
  by_contradiction negation,
  have mult_leq_n : multiplicity U ≤ n,
       by exact not_lt.mp negation,
  have HX_zero : H.functor.obj (universal_covering X) ≅ 0,
       by exact restricted_cover_lower_bound n X F U mult_leq_n
                  H H_dim_vanishing,
  have not_HX_non_zero :
       ¬ not_isomorphic (H.functor.obj (universal_covering X))
                        0,
       by exact iso_implies_not_not_isomorphic _ _ HX_zero,

  -- not_HX_non_zero contradicts HX_non_zero; thus we are done
  finish,
end
```

The lower bound strategy; refined version with two functors To model the version of the lower bound strategy in Theorem 4.2.14 we make use

of the fact that records in Lean are extensible: We extend the structure of `admissible_functor` by a second functor and a corresponding natural transformation. The formalisation of Theorem 4.2.14 and its proof is a straightforward translation of the pen-and-paper version.

```
structure admissible_functor_2
    (X : CW1)
    (F : family (pi_1 X))
extends admissible_functor X F
:= mk :: (functor2  : (CWh (pi_1 X)) ⇒ C )
         (nat_trafo : functor2 ⟶ functor)

theorem restricted_cover_lower_bound_2
    (n : nat)
    (X : CW1)
    (F : family (pi_1 X))
    (U : open_cover X F)
    (mult_U_leq_n : multiplicity U ≤ n)
    (H : admissible_functor_2 X F)
    (K_dim_vanishing : dim_vanishing H.functor2 n)
  : H.nat_trafo.app (universal_covering X) = 0
:=
begin
  -- simplified notation:
  let G  := pi_1 X,
  -- the first functor
  let HX := H.functor.obj (universal_covering X),
  let H' : admissible_functor X F
         := admissible_functor.mk
              H.functor H.ucov_iso H.family_admissible,
  -- the second functor
  let K  : CWh G ⇒ C
         := H.functor2,
  let KN := K.obj (nerve U),
  let KX := K.obj (universal_covering X),
  -- the natural transformation
  let T  := H.nat_trafo,

  -- we combine
  -- * the factorisation of id on H(ucov X) over H(nerve)
  -- * with the natural transformation
  -- * and the vanishing of K(nerve)

  -- we first show that K(nerve) ≅ 0,
  -- using the dim-vanishing property
```

```
  have KN_is_zero : KN ≅ 0,
       by exact dim_vanishing_functors_vanish_on_small_nerves
                  n X F U mult_U_leq_n K K_dim_vanishing,

  -- the resulting isomorphisms to and from the zero object
  let nz : KN ⟶ 0 := KN_is_zero.hom,
  let zn : 0 ⟶ KN := KN_is_zero.inv,

  -- second, the preparation on classifying spaces shows that
  -- H(ucov X) factors over H(nerve map)
  let f := H.functor.map (nerve_map U),
  choose g fg_idHX
         using admissible_functors_factor_over_nerve X F U H',

  -- we now combine both aspects
  -- and show that T(ucov X) factors over 0
  let TX := T.app (universal_covering X),
  let TN := T.app (nerve U),
  let Kn := K.map (nerve_map U),

  let fz : KX ⟶ 0 := Kn ≫ nz,
  let zg : 0 ⟶ HX := zn ≫ TN ≫ g,

  show _, by
  calc TX = TX ≫ 𝟙 HX
          : by simp
      ... = TX ≫ f ≫ g
          : by {congr, rw fg_idHX}
      ... = Kn ≫ TN ≫ g
          : by simp -- natural transformation
      ... = Kn ≫ nz ≫ zn ≫ TN ≫ g
          : by simp -- nz ≫ zn cancels
      ... = fz ≫ zg
          : by simp
      ... = (0 : KX ⟶ 0) ≫ (0 : 0 ⟶ HX)
          : by {congr, ext, ext}
      ... = 0
          : by simp,
end
```

This concludes the formalisation of the material from Section 4.2.2.

One could now start filling in concrete implementations of the items that we abstracted over – such as equivariant CW-complexes, singular cohomology, bounded cohomology, the family of amenable subgroups, ... But this would be a different story.

4.E Exercises

Exercise 4.E.1 (dependency graph)**.**

1. Pick your favourite textbook or lecture notes.
2. Can you formalise the very first definition of the book? Or are there missing pieces? How could the missing pieces be replaced by suitable postulates/abstractions?
3. Pick your favourite chapter.
4. Draw a dependency graph for the definitions, results, proofs, and examples of this chapter.

Exercise 4.E.2 (fundamental theorems)**.** Pick a key theorem such as the fundamental theorem of algebra, the Brouwer fixed point theorem, ... and carry out the following steps:

1. Formalise the statement of the theorem.
2. Pen-and-paper: Pick one proof of the theorem and structure it into its major steps and ideas.
3. Formalise this proof skeleton, using postulates/abstractions whenever necessary or convenient. Use `sorry` only in appropriate situations!
4. Formalise a statement/proof that uses this theorem.

Exercise 4.E.3 (the Riemann hypothesis)**.** Make an attempt at formalising the statement of the Riemann hypothesis. Which steps will you keep in the form of postulates/axioms? Which steps can you fill in with a concrete formalisation?

Exercise 4.E.4 (formalisation challenge)**.**

1. Pick your favourite research paper.
2. Pick your favourite result/proof from this paper.
3. Which description matches this result/proof best?
   - All background material is already available in Lean.
   - The result is already formulated on an abstract level.
   - The result could be formalised through a suitable abstraction step.
4. Make a concrete plan for formalising (a suitable abstraction of) this result/proof.
5. Carry out the plan!

Bibliography

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*, Pan Books, 1979. Cited on page: 12

[2] A. Asperti, H. Geuvers, R. Natarajan. Social processes, program verification and all that, *Math. Structures Comput. Sci.*, 19(5), 877–896, 2009. Cited on page: 6

[3] J. Avigad. Mathlib naming conventions.
https://leanprover-community.github.io/contribute/naming.html
Cited on page: 87

[4] J. Avigad. Library Style Guidelines.
https://leanprover-community.github.io/contribute/style.html
Cited on page: 87

[5] J. Avigad, G. Ebner, S. Ullrich. The Lean Reference Manual, Release 3.3.0, 2018.
https://leanprover.github.io/reference/lexical_structure.html
Cited on page: 7, 13

[6] J. Avigad, L. de Moura, S. Kong. *Theorem Proving in Lean*, Release 3.23.0, 2021.
https://leanprover.github.io/theorem_proving_in_lean/
Cited on page: 5, 7, 9, 13, 88

[7] J. Avigad, L. de Moura, S. Kong, S. Ullrich. *Theorem Proving in Lean 4*, 2022.
https://leanprover.github.io/theorem_proving_in_lean4/
Cited on page: 4

[8] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, J. Limperg. *The Hitchhiker's Guide to Logical Verification*, 2021 Standard Edition, 2021.
https://github.com/blanchette/logical_verification_2021/raw/main/hitchhikers_guide.pdf
Cited on page: 5, 7, 13

[9] H. Barendregt, W. Dekkers, R. Statman. *Lambda calculus with types*, with contributions from F. Alessi, M. Bezem, F. Cardone, M. Coppo, M. Dezani-Ciancaglini, G. Dowek, S. Ghilezan, F. Honsell, M. Moortgat, P. Severi and P. Urzyczyn. Perspectives in Logic. Association for Symbolic Logic, Cambridge University Press, 2013. Cited on page: 10

[10] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, An EATCS Series, Springer, 2004. Cited on page: 7

[11] N. Bourbaki. *General topology. Chapters 1–4*, translated from the French, reprint of the 1989 English translation, Elements of Mathematics, Springer, 1998. Cited on page: 42

[12] K. Buzzard, J. Commelin, P. Massot. Lean perfectoid spaces.
https://leanprover-community.github.io/lean-perfectoid-spaces/
Cited on page: 6, 7

[13] D. Calegari. *scl*, MSJ Memoirs, vol 20, Mathematical Society of Japan, 2009. Cited on page: 36

[14] M. Carneiro. The type theory of Lean, 2019.
https://github.com/digama0/lean-type-theory/releases
Cited on page: 8, 88

[15] L. Chicli, L. Pottier, C. Simpson. Mathematical quotients and quotient types. *Types for proofs and programs*, 95–107, Lecture Notes in Computer Science, 2646, Springer, 2003. Cited on page: 89

[16] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, MIT Press, 2014. Cited on page: 9

[17] C. Cohen. Pragmatic quotient types in Coq, *Interactive Theorem Proving*, 213–228, Lecture Notes in Computer Science, 7998, Springer, 2013. Cited on page: 89

[18] Coq community. Mathematical Components.
https://math-comp.github.io/ Cited on page: 6

[19] T. Coquand, G. Huet. The calculus of constructions, Technical Report RR-0530, INRIA, 1986.
https://hal.inria.fr/inria-00076024
Cited on page: 7

[20] O. Cornea, G. Lupton, J. Oprea, D. Tanré. *Lusternik-Schnirelmann category*, Mathematical Surveys and Monographs, 103, American Mathematical Society, 2003. Cited on page: 110

[21] H. Edelsbrunner, J. L. Harer. *Computational topology. An introduction*, American Mathematical Society, 2010. Cited on page: 50

[22] M. Farber. *Invitation to topological robotics*, Zürich Lectures in Advanced Mathematics, European Mathematical Society, 2008. Cited on page: 110

[23] D. P. Friedman, D. T. Christiansen. *The Little Typer*, MIT Press, 2018. Cited on page: 5

[24] R. Ghrist. *Elementary Applied Topology*, ed. 1.0, Createspace, 2014.
https://www2.math.upenn.edu/~ghrist/notes.html
Cited on page: 50

[25] M. Gromov. Volume and bounded cohomology, *Publ. Math. IHES*, 56, 5–99, 1982. Cited on page: 96, 98, 111, 112

[26] M. Gromov. *Metric structures for Riemannian and non-Riemannian spaces*. With appendices by M. Katz, P. Pansu, and S. Semmes, translated by S. M. Bates, Progress in Mathematics, 152, Birkhäuser, 1999. Cited on page: 96

[27] T. C. Hales. Developments in formal proofs, *Astérisque*, 367–368, Exp. no. 1086, 2015. Cited on page: 6

[28] J. Harrison. Verification: industrial applications. In: *Proof technology and computation*, 161–205, *NATO Sci. Ser. III Comput. Syst. Sci.*, 200, IOS, 2006. Cited on page: 6

[29] M. Herlihy, D. Kozlov, S. Rajsbaum. *Distributed computing through combinatorial topology*, Elsevier/Morgan Kaufmann, 2014. Cited on page: 50

[30] Isabelle community. The Archive of Formal Proofs,
https://www.isa-afp.org/
Cited on page: 6

[31] N. V. Ivanov. Foundations of the theory of bounded cohomology, *J. Soviet Math.*, 37, 1090–1114, 1987. Cited on page: 111

[32] N. V. Ivanov. Leray theorems in bounded cohomology theory, preprint, arXiv:2012.08038 [math.AT], 2020. Cited on page: 111

[33] Lean community. Learning Lean,
https://leanprover-community.github.io/learn.html
Cited on page: 5, 7, 13

[34] Lean community. Get started with Lean,
https://leanprover-community.github.io/get_started.html
Cited on page: 16, 22

[35] Lean community. Lean web editor,
https://leanprover-community.github.io/lean-web-editor/
Cited on page: 7, 22

[36] Lean community. Using leanproject,
https://leanprover-community.github.io/leanproject.html
Cited on page: 22

[37] Lean community. mathlib,
https://leanprover-community.github.io/mathlib-overview.html
Cited on page: 6, 9, 32, 33

[38] Lean community. mathlib4,
https://github.com/leanprover-community/mathlib4
Cited on page: 4

[39] Lean community. Lean 4,
https://github.com/leanprover/lean4
Cited on page: 4

[40] Lean community. Papers about Lean,
https://leanprover-community.github.io/papers.html Cited on page: 6

[41] Lean community. Documentation style,
https://leanprover-community.github.io/contribute/doc.html
Cited on page: 87

[42] Lean community. How to contribute to mathlib,
https://leanprover-community.github.io/contribute/index.html
Cited on page: 87

[43] K. Li, C. Löh, M. Moraschini. Bounded acylicity and relative simplicial volume, preprint, arXiv:2202.05606 [math.AT], 2022. Cited on page: 117, 118

[44] C. Löh. Finite functorial semi-norms and representability, *Int. Math. Res. Not.*, 2016(2), 3616–3638, 2016. Cited on page: 96, 97, 98

[45] C. Löh. *Geometric Group Theory. An Introduction*, Universitext, Springer, 2018. Cited on page: 111

[46] C. Löh. The example from Figure 1.1 in the Lean web editor: hyperlink to https://leanprover-community.github.io/lean-web-editor..., containing the code at https://loeh.app.uni-regensburg.de/mapa/intro_example.lean Cited on page: 11

[47] C. Löh. git repository with all Lean source files covered in these notes and templates/solutions to selected exercises, https://gitlab.com/polywuisch/mapa_notes update! Cited on page: 22, 30, 36, 39, 42, 44, 45, 46, 52, 58, 65, 73, 78, 83, 90, 98, 118

[48] C. Löh, R. Sauer. Bounded cohomology of amenable covers via classifying spaces, *Enseign. Math.*, 66(1/2), 151–172, 2020. Cited on page: 111, 112, 117, 118

[49] C. Löh, M. Uschold. L^2-Betti numbers and computability of reals, preprint, arXiv:2202.03159 [math.GR], 2022. Cited on page: 110

[50] W. Lück. Survey on classifying spaces for families of subgroups, *Infinite groups: geometric, combinatorial and dynamical aspects*, 269–322, *Progress in Mathematics*, 248, Birkhäuser, 2005. Cited on page: 112, 114

[51] P. Massot. The sphere eversion project, https://leanprover-community.github.io/sphere-eversion/blueprint/ Cited on page: 6, 7, 110

[52] C. McBride. A polynomial testing principle, preprint. https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf Cited on page: 1

[53] S. Mimram. *Program = Proof*, https://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf Cited on page: 10

[54] L. Moura, S. Ullrich. The Lean 4 Theorem Prover and Programming Language. In: A. Platzer, G. Sutcliffe (eds). *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, 12699, Springer, 2021. Cited on page: 4

[55] J. R. Munkres. *Elements of algebraic topology*, Addison-Wesley Publishing Company, 1984. Cited on page: 50

[56] J. Oprea. Applications of Lusternik-Schnirelmann category and its generalizations, *J. Geom. Symmetry Phys.*, 36, 59–97, 2014. Cited on page: 110

[57] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Logic*, 2020.
https://isabelle.in.tum.de/doc/tutorial.pdf Cited on page: 7

[58] A. L. T. Paterson. *Amenability*, Mathematical Surveys and Monographs, 29, AMS, 1988. Cited on page: 111

[59] B. Pierce (ed.). *Advanced Topics in Types and Programming Languages*, MIT Press, 2004. Cited on page: 9

[60] R. Sauer. Amenable covers, volume and L^2-Betti numbers of aspherical manifolds, *J. Reine Angew. Math.*, 636, 47–92, 2009. Cited on page: 113

[61] R. Sauer. Volume and homology growth of aspherical manifolds, *Geom. Topol.*, 20, 1035–1059, 2016. Cited on page: 113

[62] S. Smale. On the topology of algorithms. I, *J. Complexity*, 3(2), 81–89, 1987. Cited on page: 110

[63] P. Wadler, S. Blott. How to make ad-hoc polymorphism less ad hoc. *16'th Symposium on Principles of Programming Languages*, ACM Press, 1989. Cited on page: 9

[64] F. Wiedijk. Formalizing 100 Theorems.
https://www.cs.ru.nl/~freek/100/ Cited on page: 6

[65] The Xena project, https://xenaproject.wordpress.com/
Cited on page: 6, 7

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z