# ProofLab: A Short Introduction to Formalising Mathematics in Lean

Seminar on Simplicial Topology
Wintersemester 2021/22

Clara Löh

Version of October 27, 2021
clara.loeh@mathematik.uni-regensburg.de
Fakultät für Mathematik, Universität Regensburg, 93040 Regensburg

# Contents

1 The Lean proof assistant 1

1.1 Proof assistants 2
1.2 Foundations 3
1.3 Proofs 4
1.4 First examples 5
1.4.1 Pen-and-paper 6
1.4.2 Lean 8
1.E Exercises 17

2 Further examples in Lean 19

2.1 Example: Induction 20
2.1.1 Pen-and-Paper 20
2.1.2 Lean 21
2.2 Example: Sums 23
2.3 Example: Commutators 24
2.3.1 Pen-and-Paper 25
2.3.2 Lean 26
2.E Exercises 28

A Source code A.1

Bibliography C.1

Index C.3

# 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 foundation; we quickly explain how one can formalise statements and proofs in this setup.

We then practice basic proof techniques in Lean by formalising simple examples of properties of maps.

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 introduction [1] and in the Lean documentation [5]. For dependent types, there is a step-by-step introduction available [3].

**Overview of this chapter.**

| | | |
|---|---|---|
| 1.1 | Proof assistants | 2 |
| 1.2 | Foundations | 3 |
| 1.3 | Proofs | 4 |
| 1.4 | First examples | 5 |
| 1.E | Exercises | 17 |

## 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 allow 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.

- 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 [11] 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 ignore this delicate point.

- Which proof assistants are in use?

  There are many proof assistants; currently, the most popular ones are Coq, Isabelle, Lean, .... In this course, we will use Lean [5, 1].

- 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. Moreover, Lean offers a convenient web interface that allows to experiment with Lean without a proper installation.

## 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 based on type theory 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 1.3).

**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, these subtle differences will not be relevant.

## 1.3 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 $A \Longrightarrow B$ is proved, then it follows that also $B$ is proved.

The *Curry–Howard isomorphism* 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 $A \longrightarrow B$, we obtain an element of type $B$.

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

In Lean the term `x : A` means that `x` has type `A`. A lemma of the form below thus says that under the hypothesis that `x` has type `A` ("satisfies `A`"), then `deep_lemma` of `x` has type $\varphi$ `x` (i.e., $\varphi$ `x` is "satisfied"). Usually, $\varphi$ `x` is a type that represents a concrete mathematical statement (e.g., "`x` is prime").

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

  ...

end
```

A proof of this lemma is nothing but an implementation of a function from the type `A` to another type (satisfying the constraints posed by $\varphi$) and this proof is enclosed in `:= begin ... end`.

The claim of a lemma causes that a *goal* (or several goals) have to be reached. During the proof these goals are manipulated; depending on the used deduction rules and intermediate claims, goals are resolved or new goals are added. The proof is complete once all goals are reached.

Parts of the state remain implicit and also the implicit order can play a role; in order to increase readability and robustness it is therefore recommended to make proofs more explicit than Lean would require.

| | |
|---|---|
| `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 |
| `assume` | introduces an identifier, as preparation for a proof of an all-statement or an implication |
| `use` | allows to prove an existence statement from an example |
| `rcases` | extracts, e.g., a witness from an existential term |
| `cases` | case distinction |
| `induction` | proof by induction (not only over natural numbers) |
| `by` | gives a justification; can interact with other statements or proof strategies via `apply`, `exact`, `simp`, `rw`, `refine`, `arith`, ... |
| `calc` | starts a calculation |
| `library_search`, `suggest`, `hint` | searches the libraries for ways to make progress in the proof |
| `sorry` | pretends to be a proof (useful for developing the overall structure of a proof) |
| `def` | definition |
| `unfold` | unfolds a definition |
| `let` | local definition |

Figure 1.1.: Basic Lean vocabulary

A selection of the Lean vocabulary is collected in Figure 1.1. More details can be found in the documentation [1, 5]. The individual steps in proofs usually consist of the elimination or introduction of logical constructs:

- 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).

## 1.4 First examples

We will now 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; in this section, the focus is on learning how to formulate statements and proofs in Lean.

**Interactive tool 1.4.1.** Try out Lean programs in a local Lean installation [6] or in the Lean web interface [8]! For more complex projects and a more efficient workflow, a local installation is highly recommended.

Many Lean proofs in current libraries or other Lean code will only be comprehensible when loading them into a Lean interpreter. Usually, we will try to make all relevant steps in Lean proofs explicit enough that they can be enjoyed and understood by humans.

## 1.4.1 Pen-and-paper

As a first step, we note down what we want to formulate 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 1.4.2** (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 1.4.3** (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 1.4.4** (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 1.4.5.** *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 1.4.6.** *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

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

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 now consider $y := f(x) \in Y$. Then, we obtain

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

Hence, $g$ is surjective. □

**Corollary 1.4.7.** *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 1.4.6 (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 1.4.6 (second part) shows that $f$ is surjective.

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

**Example 1.4.8.** 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 1.4.9.** 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.

## 1.4.2 Lean

We will now implement the material from Section 1.4.1 in Lean.

**Interactive tool 1.4.10.** The source code discussed in this section is available on the course homepage

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/maps.lean

You can also clone (or pull from) the course git repository; the project can then be properly initialised via `leanproject build` [9].

We start with general declarations and imports:

```
import tactic   -- standard proof tactics

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

The definitions of injectivity, surjectivity, and bijectivity are straightforward adaptions of their pen-and-paper counterparts (Definitions 1.4.2–1.4.4). The declarations before `:=` are the hypotheses of the definition. 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.

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

/- Surjectivity -/
def is_surjective
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
 := ∀ y : Y,
    ∃ x : X, f x = y

/- Bijectivity -/
def is_bijective
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
 := (is_injective X Y f) ∧ (is_surjective X Y 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).

As next step, we state and prove some basic inheritance properties for injective, surjective, bijective maps (Proposition 1.4.5–Corollary 1.4.7).

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

The first three lemmas correspond to Proposition 1.4.5. The first two parts (`bij_inj`, `bij_surj`) are proved by extracting the correct parts from the defining ∧-formula; these are elimination steps.

```
/- Simple inheritance properties
   of injective, surjective, bijective maps-/
lemma bij_inj
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_bijective: is_bijective X Y f)
  : is_injective X Y f
:=
-- we extract the correct part of the and-statement
-- in the definition of is_bijective
by {exact and.elim_left f_bijective}

lemma bij_surj
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_bijective: is_bijective X Y f)
  : is_surjective X Y f
:=
-- we extract the correct part of the and-statement
-- in the definition of is_bijective
by {exact and.elim_right f_bijective}
```

The keyword `exact` applies the corresponding arguments to resolve an open goal.

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 : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_surjective: is_surjective X Y f)
    (f_injective:  is_injective X Y f)
  : is_bijective X Y f
:=
-- we construct the and-statement
-- in the definition of is_bijective
-- in the correct order
by {exact and.intro f_injective f_surjective}
```

The lemmas `inj_comp_injfirst` and `inj_comp_surjfirst` are translations of Proposition 1.4.6; 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.

```
/- If a composition is injective,
   then the first map is injective -/
lemma inj_comp_injfirst
    (X : Type*)
    (Y : Type*)
    (Z : Type*)
    (f : X → Y)
    (g : Y → Z)
    (gf_injective : is_injective X Z (g ∘ f))
  : is_injective X Y f
:=
begin
  -- we prove the all-statement (double ∀)
  -- in the definition of is_injective
  assume x  : X,
  assume x' : X,
  -- we assume the hypothesis of the implication
  -- in the definition of is_injective
  assume f_xx' : f x = f x',

  -- and then show that this implies x = x',
  -- using injectivity of g ∘ f
  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 X Y 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). At this point, all goals are resolved and the proof is complete.

```
/- If a composition is surjective,
   then the last map is surjective -/
lemma surj_comp_surjsecond
    (X : Type*)
    (Y : Type*)
    (Z : Type*)
    (f : X → Y)
    (g : Y → Z)
    (gf_surjective : is_surjective X Z (g ∘ f))
  : is_surjective Y Z g
:=
begin
  -- we prove the all-statement
  -- in the definition of is_surjective
  assume z : Z,

  -- we use surjectivity of g ∘ f
  have ex_x : ∃ x : X, (g ∘ f) x = z,
       by {exact gf_surjective z},

  -- we extract such a preimage
  rcases ex_x with ⟨ x : X, gf_x_z⟩,
  -- and use it to define a g-preimage of z
  let y : Y := f x,

  -- we construct the existential statement
  -- in the definition of is_surjective,
  -- by using the example y
  use y,
  -- it remains to show that y indeed is a g-preimage of z
  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 1.4.7 is a direct translation of our pen-and-paper proof:

```
/- If the square of a self-map is bijective,
   then the self-map is bijective -/
lemma square_bij_bij
      (X : Type*)
      (f : X → X)
      (ff_bijective: is_bijective X X (f ∘ f))
    : is_bijective X X f
:=
begin
  -- the map f is injective
  have f_injective: is_injective X X f, from
  begin
    -- the composition f ∘ f is bijective, whence injective
    have ff_injective,
         by {exact bij_inj X X (f ∘ f) ff_bijective},
    -- thus, the first map (namely f) is injective
    show _,
         by {exact inj_comp_injfirst X X X f f ff_injective},
  end,
  -- the map f is surjective
  have f_surjective: is_surjective X X f, from
  begin
    -- the composition f ∘ f is bijective, wehnce surjective
    have ff_surjective,
         by {exact bij_surj _ _ (f ∘ f) ff_bijective},
    -- thus, the second map (namely f) is surjective
    show _,
         by {exact surj_comp_surjsecond _ _ _ f f ff_surjective
    },
  end,

  -- thus, f is bijective
  show is_bijective X X f,
       by {exact and.intro f_injective f_surjective},
end
```

Finally, we explain how to formalise Example 1.4.8 and Example 1.4.9 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 1.4.8, 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 1.4.8 and 1.4.9 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`.

```
/- The map {1,2,3} -> {1,2,3},
   1 -> 1, 2 -> 1, 3 -> 2
   is neither injective nor surjective -/
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 1.4.8, we show that this map `f` is neither injective nor surjective. We follow the outline given in Example 1.4.8; however, we will need to be more disciplined than in the pen-and-paper version (which contains many implicit steps).

```
lemma not_inj_f
    : ¬ is_injective A A 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,

  -- x and x' are witnesses for non-injectivity:
  have f_xx'_x_neg_x' : f x = f x' ∧ x ≠ x', from
  begin
    have f_xx' : f x  = f x', by {simp[f]},
    have x_neg_x' :  x ≠ x', by {finish},

    show _, by {exact and.intro f_xx' x_neg_x'},
  end,

  -- we move the negation to the innermost formula,
  -- use x and x' as examples for the existential quantifier,
  -- and then conclude via f_xx'_x_neg_x'
  show _, from
  begin
    unfold is_injective,
    push_neg,
    use x,
    use x',
    exact f_xx'_x_neg_x',
  end
end

lemma not_surj_f
     : ¬ is_surjective A A f
:=
begin
  -- we first move the negation through the all-quantifier
  refine not_forall_of_exists_not _,
  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 at *, exact A3_not_in_im}
  end
end
```

Similarly, we can also transform Example 1.4.9 into Lean:

```
/- The map {1,2} -> {1,2},
   1 -> 2, 2 -> 1
   is bijective -/
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 B B g
:=
begin
  -- we check injectivity and surjectivity
  -- by going through all the cases
  have inj_g : is_injective B B 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 B B 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 the Exercises (Section 1.E).

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.

## 1.E Exercises

**Exercise 1.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
   ```

**Exercise 1.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.

**Exercise 1.E.3** (formalising Example 1.4.8)**.** 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 an alternative proof of `lemma not_inj_f`.

**Exercise 1.E.4** (formalising Example 1.4.9)**.**

1. Show that the map $g$ from Example 1.4.9 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_gs`.

Hints. In case you need help: There is a basic source skeleton for these exercises available (p. A.10; also in the course git repository):

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/maps_exercise.lean

# 2

# Further examples in Lean

The Lean mathlib provides a wide range of proof tactics and mathematical libraries to simplify the task of formalising and proving mathematical statements.

We will first get acquainted with basic induction proofs, using geometric sums as example. Such proofs are very common; we will learn how to use mathlib to solve tasks of this type.

Moreover, as a first example of other structures provided by mathlib, we will look at a basic example in group theory.

**Overview of this chapter.**

2.1 Example: Induction 20
2.2 Example: Sums 23
2.3 Example: Commutators 24
2.E Exercises 28

## 2.1 Example: 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 $n \in \mathbb{N}$.

### 2.1.1 Pen-and-Paper

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

**Proposition 2.1.1.** *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. □

### 2.1.2 Lean

We will now implement the material from Section 2.1.1 in Lean.

**Interactive tool 2.1.2.** The source code discussed in this (and the next) section is available on the course homepage

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/induction.lean

You can also clone (or pull from) the course git repository.

We start with general declarations and imports; the imports finset and big_operators are only relevant for Section 2.2:

```
import tactic    -- standard proof tactics
open finset      -- for range operator
open_locale big_operators -- to enable Σ notation

open classical   -- we want to 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). 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).

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 2.1.1 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; 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 1.4.2).

```
begin
  -- we prove this claim by induction (over the natural number
    argument n);
  -- here, m is the variable used in the induction step
  -- and ind_hyp is the induction hypothesis used in the
    induction step
  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 geometrics sums at base 2:

```
#eval geometric_sum 0
#eval geometric_sum 5
```

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

## 2.2 Example: Sums

In Section 2.1.2, we considered geometric sums in Lean through an explicit inductive definition. As 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.

In this section, we will consider a simple example using sums as provided by the library big_operators, which is part of mathlib.

**Interactive tool 2.2.1.** The source code discussed in this (and the previous) section is available on the course homepage

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/induction.lean

You can also clone (or pull from) the course git repository.

The Lean mathlib can be found at:

https://leanprover-community.github.io/mathlib-overview.html

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

```
-- using the sum operator from mathlib (big_operators)
def one_sum
  : nat → nat
:= λ n : nat,
   Σ (i : nat) in range n, 1
```

The sum notation $\sum$ is provided by the library big_operators and `range n` (which corresponds to $\{0, \dots, n-1\}$) is provided by the library finset. Moreover, `\lambda` is a constructor for (unnamed) functions.

Of course, we have $\sum_{j=0}^{n} 1 = n$ for all $n \in \mathbb{N}$. This statement can be formalised and proved as in the case of geometric sums in Section 2.1.2:

```
lemma one_sum_eval
      (n : nat)
    : (one_sum n = n)
```

```
:=
begin
  -- we prove this claim by induction (over the natural number
    argument n)
  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) = Σ (i : nat) in range (m+1), 1
                       : by {simp[one_sum]}
                   ... = (Σ (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 [7] 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.3 to figure out what `sum_range_induction` exactly is about and how it is proved in mathlib.

## 2.3 Example: 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*

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

*Proof.* We compute that

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

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

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

as claimed. □

The previous proposition is important in the study of stable commutator length in groups [2].

## 2.3.2 Lean

We will now implement the material from Section 2.3.1 in Lean.

**Interactive tool 2.3.4.** The source code discussed in this section is available on the course homepage

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/commutator.lean

You can also clone (or pull from) the course git repository.

The Lean mathlib can be found at:

https://leanprover-community.github.io/mathlib-overview.html

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 want to work in classical logic
```

Definition 2.3.1 translates directly to Lean. In this definition, `[group G]` is also an argument/hypothesis (namely that `G` is a group), 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. 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 g h) = cmtr H (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`).

```
begin
  calc f (cmtr G 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[monoid_hom.map_inv],
                                 simp[monoid_hom.map_inv]}
                  ... = cmtr H (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 show 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
  calc g^3 = g^2 * g : by {exact pow_succ' g 2}
       ... = g * g * g : by {simp[pow_two]}
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]
      (b : G) [B : G]
      [A_def : A = a⁻¹]
      [B_def : B = b⁻¹]
    : (cmtr G a b)^3
    = cmtr G (a*b*A) (B*a*b*A^2) * cmtr G (B*a*b) (b^2)
:=
begin
  unfold cmtr,
  by {simp[pow_three,A_def,B_def],group},
end
```

## 2.E Exercises

**Exercise 2.E.1** (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.

**Exercise 2.E.2** (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.

**Exercise 2.E.3** (general sums and inductive computation). We consider the lemma `sum_range_induction` from Lean library 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 would you use it to show $\sum_{j=0}^{n} 1 = n$ for all $n \in \mathbb{N}$?
4. How is `sum_range_induction` proved in the Lean library?

**Exercise 2.E.4** (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?

Hints. In case you need help: There is a basic source skeleton for these exercises available (p. A.18; also in the course git repository):

http://www.mathematik.uni-r.de/loeh/teaching/prooflab_ws2122/induction_exercise.lean

# A

# Source code

- maps.lean: An implementation of basic properties of maps in Lean; p. A.2.
- maps_exercise.lean: Skeleton for the exercises in Section 1.E; p. A.10
- induction.lean: Simple induction proofs for sums; p. A.13.
- induction_exercise.lean: Skeleton for the exercises in Section 2.E; p. A.18
- commutator.lean: Basics on commutators in groups; p. A.16.

## maps.lean

```
/- Clara L"oh 2021 -/

import tactic   -- standard proof tactics

open classical  -- we want to work in classical logic

/-
# Injective, surjective, bijective maps
-/

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

/- Surjectivity -/
def is_surjective
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
 := ∀ y : Y,
    ∃ x : X, f x = y

/- Bijectivity -/
def is_bijective
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
 := (is_injective X Y f) ∧ (is_surjective X Y f)

/- Simple inheritance properties
   of injective, surjective, bijective maps-/
lemma bij_inj
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_bijective: is_bijective X Y f)
```

```
  : is_injective X Y f
:=
-- we extract the correct part of the and-statement
-- in the definition of is_bijective
by {exact and.elim_left f_bijective}
-- alternatively: by {exact f_bijective.1}

lemma bij_surj
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_bijective: is_bijective X Y f)
  : is_surjective X Y f
:=
-- we extract the correct part of the and-statement
-- in the definition of is_bijective
by {exact and.elim_right f_bijective}
-- alternatively: by {exact f_bijective.2}

lemma surj_inj_bij
    (X : Type*)
    (Y : Type*)
    (f : X → Y)
    (f_surjective: is_surjective X Y f)
    (f_injective:  is_injective X Y f)
  : is_bijective X Y f
:=
-- we construct the and-statement
-- in the definition of is_bijective
-- in the correct order
by {exact and.intro f_injective f_surjective}

/- If a composition is injective,
   then the first map is injective -/
lemma inj_comp_injfirst
    (X : Type*)
    (Y : Type*)
    (Z : Type*)
    (f : X → Y)
    (g : Y → Z)
    (gf_injective : is_injective X Z (g ∘ f))
  : is_injective X Y f
:=
begin
  -- we prove the all-statement (double ∀)
  -- in the definition of is_injective
  assume x  : X,
  assume x' : X,
  -- we assume the hypothesis of the implication
  -- in the definition of is_injective
  assume f_xx' : f x = f x',

  -- and then show that this implies x = x',
  -- using injectivity of g ∘ f
  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

/- If a composition is surjective,
   then the last map is surjective -/
lemma surj_comp_surjsecond
    (X : Type*)
    (Y : Type*)
    (Z : Type*)
    (f : X → Y)
    (g : Y → Z)
    (gf_surjective : is_surjective X Z (g ∘ f))
  : is_surjective Y Z g
:=
begin
  -- we prove the all-statement
  -- in the definition of is_surjective
  assume z : Z,

  -- we use surjectivity of g ∘ f
  have ex_x : ∃ x : X, (g ∘ f) x = z,
       by {exact gf_surjective z},

  -- we extract such a preimage
  rcases ex_x with ⟨ x : X, gf_x_z⟩,
  -- and use it to define a g-preimage of z
  let y : Y := f x,

  -- we construct the existential statement
  -- in the definition of is_surjective,
  -- by using the example y
  use y,
  -- it remains to show that y indeed is a g-preimage of z
  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

/- If the square of a self-map is bijective,
   then the self-map is bijective -/
lemma square_bij_bij
      (X : Type*)
      (f : X → X)
      (ff_bijective: is_bijective X X (f ∘ f))
    : is_bijective X X f
:=
begin
  -- the map f is injective
  have f_injective: is_injective X X f, from
  begin
    -- the composition f ∘ f is bijective, whence injective
    have ff_injective,
         by {exact bij_inj X X (f ∘ f) ff_bijective},
    -- thus, the first map (namely f) is injective
    show _,
         by {exact inj_comp_injfirst X X X f f ff_injective},
  end,

  -- the map f is surjective
  have f_surjective: is_surjective X X f, from
  begin
    -- the composition f ∘ f is bijective, wehnce surjective
    have ff_surjective,
         by {exact bij_surj _ _ (f ∘ f) ff_bijective},
    -- thus, the second map (namely f) is surjective
    show _,
         by {exact surj_comp_surjsecond _ _ _ f f ff_surjective
    },
  end,

  -- thus, f is bijective
  show is_bijective X X f,
       by {exact and.intro f_injective f_surjective},
      -- alternatively: by {exact surj_inj_bij X X f
    f_surjective f_injective}
end

/- Some simple examples -/

/- The map {1,2,3} -> {1,2,3},
   1 -> 1, 2 -> 1, 3 -> 2
   is neither injective nor surjective -/
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

lemma not_inj_f
    : ¬ is_injective A A 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,

  -- x and x' are witnesses for non-injectivity:
  have f_xx'_x_neg_x' : f x = f x' ∧ x ≠ x', from
  begin
    have f_xx' : f x  = f x', by {simp[f]},
    have x_neg_x' :  x ≠ x', by {finish},

    show _, by {exact and.intro f_xx' x_neg_x'},
  end,

  -- we move the negation to the innermost formula,
  -- use x and x' as examples for the existential quantifier,
  -- and then conclude via f_xx'_x_neg_x'
  show _, from
  begin
    unfold is_injective,
    push_neg,
    use x,
    use x’,
    exact f_xx’_x_neg_x’,
  end
end

lemma not_surj_f
     : ¬ is_surjective A A f
:=
begin
  -- we first move the negation through the all-quantifier
  refine not_forall_of_exists_not _,
  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 at *, exact A3_not_in_im}
  end
end

/- The map {1,2} -> {1,2},
   1 -> 2, 2 -> 1
   is bijective -/
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 B B g
:=
begin
  -- we check injectivity and surjectivity
  -- by going through all the cases
  have inj_g : is_injective B B 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 B B 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

/- And (parts of) the same thing,
   but with (types from) sets instead of sum types -/
def set_123 : set ℕ
:= {1,2,3}

lemma one_in_123 : 1 ∈ set_123
:= by {fconstructor,linarith}

lemma two_in_123 : 2 ∈ set_123
:= by {apply or.inr, finish}

lemma three_in_123 : 3 ∈ set_123
:= by {apply or.inr, finish}

def map_const1
  : set_123 → set_123 --({1,2,3} : set ℕ) → ({1,2,3} : set ℕ
    )
:= λ ⟨x, _⟩, ⟨1, one_in_123⟩

def f’
  : set_123 → set_123
:= λ ⟨x, x_in_123⟩,
   if x = 3 then ⟨ 2, two_in_123 ⟩
            else ⟨ 1, one_in_123 ⟩

lemma not_inj_f'
    : ¬ is_injective set_123 set_123 f'
:=
begin
  -- idea: f' 1 = f' 2, even though 1 ≠ 2
  let x1 : ↥set_123 := ⟨ 1, one_in_123 ⟩,
  let x2 : ↥set_123 := ⟨ 2, two_in_123 ⟩,

  -- we move the negation through the first all-quantifier
  refine (not_forall.mpr _),
  -- we use A_1 as first input
  use x1,
  -- we move the negation through the second all-quantifier
  refine (not_forall.mpr _),
  -- we use A_2 as second input
  use x2,
  -- we use the definition of f'
  by {finish},
end
```

## maps_exercise.lean

```
/- Clara L"oh 2021 -/

import tactic

import maps

open classical


/- Exercise 1 -/

-- the identity map
def id_map
    (X : Type*)
  : X → X
:= λ x, x

lemma id_bijective
    (X : Type*)
  : is_bijective X X (id_map X)
:=
begin
  let f : X → X := id_map X,

  -- injectivity
  have id_inj : is_injective X X f, from
  begin
    sorry,
  end,

  -- surjectivity
  have id_surj : is_surjective X X f, from
  begin
    sorry,
  end,

  show _,
       by {exact surj_inj_bij X X f id_surj id_inj},
end
```

```
/- Exercise 2 -/

lemma comp_inj_is_inj
      (X : Type*)
      (Y : Type*)
      (Z : Type*)
      (f : X → Y)
      (g : Y → Z)
      (f_injective : is_injective X Y f)
      (g_injective : is_injective Y Z g)
    : is_injective X Z (g ∘ f)
:=
begin

  sorry,

end

/- Exercise 3 -/

lemma inj_from_examples
      (X : Type*)
      (Y : Type*)
      (f : X → Y)
      (x : X)
      (x' : X)
      (x_neq_x' : x ≠ x')
      (f_xx' : f x = f x')
    : ¬ is_injective X  Y f
:=
begin

  sorry,

end

-- redoing the first example
lemma not_inj_f_alt
    : ¬ is_injective A A f
:=
begin

  sorry,

end
```

```
/- Exercise 4 -/

-- redoing the second example
lemma gg_id
    : g ∘ g = id_map B
:=
begin

  sorry,

end

lemma bij_g_alt
    : is_bijective B B g
:=
begin

  sorry,

end
```

## induction.lean

```
/- Clara L"oh 2021 -/

import tactic   -- standard proof tactics
open finset     -- for range operator
open_locale big_operators -- to enable Σ notation

open classical  -- we want to work in classical logic

/-
# A simple induction proof
-/

-- we define geometric sums (at base 2) ...
def geometric_sum
  : nat → nat
| 0            := 1
| (nat.succ n) := geometric_sum n + 2^(n+1)

-- ... and show how they can be computed
lemma geometric_sum_eval
      (n : nat)
    : geometric_sum n = 2^(n+1) - 1
:=
begin
  -- we prove this claim by induction (over the natural number
    argument n);
  -- here, m is the variable used in the induction step
  -- and ind_hyp is the induction hypothesis used in the
    induction step
  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

-- computing some examples:
#eval geometric_sum 0
#eval geometric_sum 5


/-
# Using the Σ notation
-/

-- and another simple sum,
-- using the sum operator from mathlib (big_operators)
def one_sum
  : nat → nat
:= λ n : nat,
   Σ (i : nat) in range n, 1

lemma one_sum_eval
      (n : nat)
    : (one_sum n = n)
:=
begin
  -- we prove this claim by induction (over the natural number
    argument n)
  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) = Σ (i : nat) in range (m+1), 1   : by {
    simp[one_sum]}
```

```
                    ... = (Σ (i : nat) in range m, 1) + 1 : by {
    simp}
                    ... = m + 1                          : by {
    simp[ind_hyp]},
  end
  /-
  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
```

## commutator.lean

```
/- Clara L"oh 2021 -/

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

open classical  -- we want to work in classical logic

/-
# Commutators in groups
-/

-- we define the commutator of two given group elements
def cmtr
    (G : Type*) [group G]
    (g : G)
    (h : G)
:= g * h *  g^{-1} * h^{-1}

-- images of commutators under homomorphisms are commutators
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 g h) = cmtr H (f g) (f h)
:=
begin
  -- this is a straightforward computation,
  -- using that f is a homomorphism
  calc f (cmtr G 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[monoid_hom.map_inv],
    simp[monoid_hom.map_inv]}
                ... = cmtr H (f g) (f h)
                    : by {simp[cmtr]},
end

-- as a preparation for triple powers of commutators,
-- we establish that g^3 = g * g * g:
lemma pow_three
      (G : Type*) [group G]
      (g : G)
    : (g^3 = g * g * g)
:=
begin
  calc g^3 = g^2 * g : by {exact pow_succ' g 2}
       ... = g * g * g : by {simp[pow_two]}
end

-- triple powers of commutators are products of _two_
    commutators
lemma cmtr_pow_three
      (G : Type*) [group G]
      (a : G) [A : G]
      (b : G) [B : G]
      [A_def : A = a⁻¹]
      [B_def : B = b⁻¹]
    : (cmtr G a b)^3 = cmtr G (a*b*A) (B*a*b*A^2) * cmtr G (B*a
    *b) (b^2)
:=
begin
  -- with the help of pow_three,
  -- the group tactic can perform the computation
  unfold cmtr,
  by {simp[pow_three,A_def,B_def],group},
end
```

## induction_exercise.lean

```
/- Clara L"oh 2021 -/

import tactic -- standard proof tactics
import algebra.group.basic -- basic group theory
open finset   -- for range operator
open_locale big_operators -- to enable Σ notation


open classical -- we want to work in classical logic


/- Exercise 1 -/

-- the sum of the natural numbers 0,...,n
def first_nat_sum
  : nat → nat
| 0            := -- add definition
| (nat.succ n) := -- add definition

-- ... and its value in closed form
lemma first_nat_sum_eval
      (n : nat)
    : first_nat_sum n = n * (n+1)
:=
begin
  -- we prove this claim by induction (over the natural number
    arugment n)
  induction n with m ind_hyp,

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

  -- induction step: m -> m+1 with induction hypothesis ind_hyp
  case nat.succ :
  begin

    sorry,

  end
end
```

```
-- alternatively: using Σ (optional)

/-

def first_nat_sum'
  : nat → nat
:= λ n, 2 * Σ (i : nat) in range (n+1), -- complete definition

lemma first_nat_sum'_eval
      (n : nat)
    : first_nat_sum' n = n * (n+1)
:=
begin
  have rw_sum : first_nat_sum' = λ n : nat, Σ (i : nat) in
    range (n+1), 2 * i, from
  begin

    sorry,

    -- unfold, ext1, and library_search might help
  end,

  -- induction proof, using rw_sum
  -- sum_range_succ might help
  induction n with m ind_hyp,

end

-/

/- Exercise 2 -/

lemma powers_of_conjugates
      (G : Type*) [group G]
      (a : G)
      (b : G)
      (n : nat)
    : -- complete statement
:=
begin
  -- we prove this claim by induction (over the natural number
    arugment n)

  sorry,
end

lemma commuting_powers
      (G : Type*) [group G]
      (a : G)
      (b : G)
      -- complete the hypotheses
      (n : nat)
    : -- complete the statemnt
:=
begin

   sorry,

end
```

# Bibliography

Please note that this bibliography is still growing; thus, labels might change!

[1] J. Avigad, L. de Moura, S. Kong. *Theorem Proving in Lean*, Release 3.23.0, https://leanprover.github.io/theorem_proving_in_lean/, 2021. Cited on page: 1, 3, 5

[2] D. Calegari. *scl*, MSJ Memoirs, vol 20, Mathematical Society of Japan, 2009. Cited on page: 25

[3] D.P. Friedman, D.T. Christiansen. *The Little Typer*, MIT Press, 2018. Cited on page: 1

[4] G. Gonthier. Formal proof–the four-color theorem, *Notices Amer. Math. Soc.*, 55(11), S. 1382–1393, 2008.
Implementierung in Coq: https://github.com/math-comp/fourcolor
Cited on page:

[5] Lean community. Learning Lean,
https://leanprover-community.github.io/learn.html Cited on page: 1, 3, 5

[6] Lean community. Get started with Lean,
https://leanprover-community.github.io/get_started.html Cited on page: 6

[7] Lean community. mathlib,
https://leanprover-community.github.io/mathlib-overview.html Cited on page: 24

[8] Lean community. Lean web editor, https://leanprover-community.github.io/lean-web-editor/ Cited on page: 6

[9] Lean community. Using leanproject, https://leanprover-community.github.io/leanproject.html Cited on page: 8

[10] R.M. Smullyan, M. Fitting. *Set theory and the continuum problem*, überarbeitete Auflage, Dover, 2010. Cited on page:

[11] The Xena project, https://xenaproject.wordpress.com/ Cited on page: 2

# Index

**B**

bijective, 6

**C**

commutator, 24, 25
Curry–Howard isomorphism, 4

**F**

foundation, 3

**G**

geometric sum, 20

**I**

induction, 20
injective, 6

**L**

Lean, 1
  commutator, 26
  examples, 8
  induction, 21
  proofs, 4
  sum, 23
  vocabulary, 4

**M**

map
  bijective, 6
  injective, 6
  surjective, 6
modus ponens, 4

**P**

Peano axioms, 21
proof, 4
proof assistant, 1, 2

**S**

source code, A.1
sum, 23
surjective, 6