Formalization of Set Theory in Lean

Juliana Brinker

May 11, 2026

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Philipp Hieronymi

Zweitgutachter: Dr. Tingxiang Zou
Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Introduction

This thesis presents a formalization in Lean [2] based on the set theory chapter of the lecture "Einführung in die Mathematische Logik" (from 2025) by Philipp Hieronymi [4]. The formalization follows the script closely and extends up to the theorem that the natural numbers are a set. The beginning of the formalization was done by Philipp Hieronymi until the theorem that the cartesian product of two sets is again a set. The next (not yet formalized) lecture chapter is about transfinite induction and ordinal number recursion.

The first chapters of the lecture introduce syntax, semantics, and proof calculi. The next part, which is of interest for this work, builds the logical foundations of mathematics based on set theory. Since this is done in a formal way, formal proofs (derivations over the sequent calculus) are used. The difficulty with these formal proofs is that they are neither easy to read nor easy to write. To improve the learning experience the proof assistant Lean could be used to do formal proofs interactively. The formalization, done within the scope of this thesis, provides an environment that allows to include Lean, an interactive theorem prover, into the lecture. This document can be used alongside the lecture "Einführung in die Mathematische Logik", to teach the basics of doing formal proofs in Lean.

This document is split into three parts. The first chapter will be an introduction to type theory to understand the foundations of Lean. The second chapter will be a tutorial for reading and working with the Lean formalization. The third chapter will document the Lean formalization, describes two example proofs in detail and compares a lecture proof with the corresponding Lean proof.

1 Type Theory

Lean statements are expressed in dependent type theory. This chapter first introduces the simply typed lambda calculus to build intuition for type-theoretic reasoning, and then discusses key concepts and properties of dependent type theory. The contents of this chapter are based on notes I took during a lecture on Type Theory given by Prof. J. H. Geuvers at Eindhoven University of Technology [3].

1.1 The Simply Typed Lambda Calculus

The simply typed λ-calculus assigns a type to every well-formed term and allows only type-correct applications.

1.1.1 Types and Terms

Simple types are generated from a fixed set of atomic types by the type constructor:

$$\sigma, \tau ::= \alpha \mid \sigma \to \tau$$

Here, α ranges over the atomic types. The type $\sigma \to \tau$ is the **type of functions** taking arguments of type σ and returning values of type τ. The connector $\to$ is right-associative, we write $\sigma \to \tau \to \rho$ for $\sigma \to (\tau \to \rho)$.

Terms are given by the grammar

$$M, N ::= x \mid MN \mid \lambda x^{\sigma}.M \tag{1}$$

The term MN denotes **application**, which is left-associative, so MNP means $(MN)P$. The term $\lambda x^{\sigma}.M$ denotes **abstraction**: a function with argument x of type σ and body M. Abstraction associates to the right, so $\lambda x^{\sigma}.\lambda y^{\tau}.\lambda z^{\rho}.M$ means $\lambda x^{\sigma}.(\lambda y^{\tau}.\lambda z^{\rho}.M)$. In the abstraction $\lambda x^{\sigma}.M$ the variable x is called **bound** in M. Variables not bound by any abstraction are called **free**. We write $FV(M)$ for the **free variables of** M and $BV(M)$ for the **bound variables in** M. For example, in $\lambda x^{\sigma}.yx$, the variable x is bound, while y is free.

1.1.2 Typing Judgments

Typing is defined relative to a context. A **context** Γ is a finite list of variable declarations $(x_1 : \sigma_1, \ldots, x_n : \sigma_n)$. We assume that variables occur at most once in a context. A **typing judgment** has the form $\Gamma \vdash M : \sigma$, this means under the assumptions in Γ the term M has type σ.

The **typing rules** of the simply typed λ-calculus are the following:

Variable Rule (var) A variable has the type assigned to it by the context:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

Function Application (app) If M is a function from σ to τ, and N is an argument of type σ, then the application MN has type τ:

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Lambda Abstraction (abs) If the term M has type τ under the additional assumption that x has type σ, then the abstraction $\lambda x^{\sigma}.M$ has type $\sigma \to \tau$:

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.M : \sigma \to \tau}$$

These three rules correspond to the three syntactic forms of terms (1), namely variables, applications, and abstractions. They ensure that only well-typed applications can be formed. In particular, a term MN is well typed, only if the type of M is a function type whose domain matches the type of N.

We can check a typing judgment by giving a derivation over the calculus defined by (var), (app) and (abs). In the following *flag-style proof* we only annotate *function application* and *lambda abstraction* and use the *variable rule* implicitly, referencing directly to the context.

Given no specific context, we can check the following term:type pair:

$$\lambda x^{\alpha \to ((\alpha \to \beta) \to \beta) \to \gamma}.\lambda y^{\alpha}.xy(\lambda z^{\alpha \to \beta}.zy) : (\alpha \to ((\alpha \to \beta) \to \beta) \to \gamma) \to \alpha \to \gamma$$

The corresponding flag-style proof is:

(1)	$\boxed{x : \alpha \to ((\alpha \to \beta) \to \beta) \to \gamma}$	
(2)	$\quad\boxed{y : \alpha}$	
(3)	$\quad xy : ((\alpha \to \beta) \to \beta) \to \gamma$	app (1) (2)
(4)	$\quad\boxed{z : \alpha \to \beta}$	
(5)	$\quad\quad zy : \beta$	app (4) (2)
(6)	$\quad\lambda z^{\alpha \to \beta}.zy : (\alpha \to \beta) \to \beta$	abs (4) (5)
(7)	$\quad xy(\lambda z^{\alpha \to \beta}.zy) : \gamma$	app (3) (6)

(8)	$\lambda y^{\alpha}.xy(\lambda z^{\alpha\to\beta}.zy) : \alpha \to \gamma$	abs (2) (7)
(9)	$\lambda x^{\alpha\to((\alpha\to\beta)\to\beta)\to\gamma}.\lambda y^{\alpha}.xy(\lambda z^{\alpha\to\beta}.zy) :$ $(\alpha \to ((\alpha \to \beta) \to \beta) \to \gamma) \to \alpha \to \gamma$	abs (1) (8)

The type check succeeds because the displayed derivation constructs the required typing judgment using only the rules (var), (app) and (abs).

Given the context $\Gamma = (x : \sigma,\ y : \sigma \to \tau)$, we cannot show $xy : \tau$, because x does not have a function type. But the term yx has type τ.

(1)	$\boxed{x : \sigma}$	
(2)	$\boxed{y : \sigma \to \tau}$	
(3)	$yx : \tau$	app (2) (1)
(4)	$xy : \tau$	!cannot be derived!

1.1.3 Substitution

Substitution replaces all free occurrences of a variable by a term. We write

$$M[x := N]$$

for the result of substituting N for all free occurrences of x in M. Substitution is always understood to be capture-avoiding, so free variables of N must not become bound after substitution. If necessary, bound variables are first renamed.

1.1.4 β-Reduction

The computational behavior of λ-terms is described by **β-reduction**, written as $\to_\beta$. A term that has the application of a function to an argument, e.g. $(\lambda x^{\sigma}.M)N$ is called a **β-redex**. The corresponding reduction step replaces the formal parameter x in the body M by the actual argument N:

$$(\lambda x^{\sigma}.M)N \to_\beta M[x := N].$$

Then the term $M[x := N]$ is called the **reduct**. A β-redex may be contracted anywhere inside a term. As with substitution in general, β-reduction is capture-avoiding. Reduction examples, if $N : \sigma$, are

$$(\lambda x^{\sigma}.x)N \to_\beta N \text{ and}$$
$$(\lambda x^{\sigma}.\lambda y^{\tau}.x)N \to_\beta \lambda y^{\tau}.N$$

If $y \in FV(N)$, the bound variable y must first be renamed.

The relation $\to_\beta$ denotes **one-step** β-reduction. Its reflexive, symmetric, and transitive closure is called **β-equivalence** and is written $=_\beta$. Thus, $M =_\beta N$ means that M and N can be connected by finitely many forward or backward β-reduction steps.

1.1.5 From Simple Types to Dependent Types

There are more aspects to the simply typed lambda calculus we will not cover here, like **α-equivalence**, written $\equiv_\alpha$, which is the relation identifying terms that differ only in the names of bound variables. The discussed topics will be sufficient for our purposes.

While the simply typed lambda calculus is a good start to understand the dynamics of a type theory, it is not sufficient for formalizing mathematics, as it is simply not expressive enough.

1.2 Dependent Type Theory

In dependent type theory, types are themselves represented by expressions of the calculus and classified by, so called, sorts. This additional expressiveness allows precise mathematical specifications to be encoded directly in types.

1.2.1 Syntax

The terms of dependent type theory are generated by

$$M, N, A, B ::= x \mid * \mid \Box \mid MN \mid \lambda x : A.M \mid \Pi x : A.B(x)$$

Here, x ranges over variables, the symbols $*$ and $\Box$ are called sorts, MN denotes application, $\lambda x : A.M$ denotes abstraction and $\Pi x : A.B(x)$ denotes a dependent product. Formally, it has to be $\Pi x : A.B$, but the notation $B(x)$ is used here to emphasize that B may contain the variable x.

We do not discuss the details of the derivation rules. But the judgment $\vdash * : \Box$ is an axiom of dependent type theory, which says that $*$ itself has sort $\Box$. We see $*$ as the **sort of ordinary types** and $\Box$ as a **higher sort** that classifies $*$.

We interpret M and N as terms and A and B as types. For $s \in \{*, \Box\}$ the expression $\Gamma \vdash A : s$ reads as A is well-formed and has sort s. And $\Gamma \vdash M : A$ reads as M has type A.

1.2.2 Dependent Products

The **dependent product**

$$\Pi x : A.B(x)$$

binds the variable x in $B(x)$, just as $\lambda x : A.M$ binds x in M. Again we write $B(x)$ instead of B to make clear, that B may depend on x. The difference is that $\lambda x : A.M$ forms a function *term*, whereas $\Pi x : A.B(x)$ forms the *type* of such dependent functions. A term of the dependent product type $\Pi x : A.B(x)$ is a function that takes an argument $x : A$ and returns a term of type $B(x)$. This cannot be expressed directly using an ordinary function type $A \to B$, because there the codomain B is fixed. When x does not occur in B, the dependent product $\Pi x : A.B$ behaves like the ordinary function type $A \to B$.

1.2.3 Types as Propositions

The dependent type theory is related to logic by the **Curry-Howard correspondence**. Under this interpretation, propositions are types and proofs are terms. A judgment

$$\Gamma \vdash M : A$$

can hence be read not only as: M is a term of type A under the assumptions Γ, but also as: M is a proof of the proposition A from the assumptions Γ.

Connecting this interpretation with dependent products gives us *universal quantification*. The type $\Pi x : A.P(x)$ can be read as $\forall x : A$ holds $P(x)$.
A proof/term of $A \to B$ is a function that transforms a proof/term of A into a proof/term of B. Similarly, a proof/term of $\Pi x : A.P(x)$ is a function which takes an arbitrary $x : A$ and returns a proof/term of $P(x)$. Hence it proves that $P(x)$ holds for every x of type A.

1.2.4 Basic Metatheoretic Properties

The following properties ensure that computation behaves well with respect to typing. The dependent type theory satisfies all of them.

Uniqueness of types: A term has at most one type, up to the definitional equality of the calculus, here represented by β-equivalence:

$$\text{if } \Gamma \vdash M : A \text{ and } \Gamma \vdash M : B, \text{ then } A =_\beta B$$

Subject reduction: Typing is preserved by computation:

$$\text{if } \Gamma \vdash M : A \text{ and } M \to_\beta N, \text{ then } \Gamma \vdash N : A$$

Strong normalization: Every reduction sequence starting from a well-typed term terminates:

$$\text{if } \Gamma \vdash M : A, \text{ then every sequence of } \beta\text{-reductions from } M \text{ is finite}$$

1.2.5 Decidability

Having a decidable problem means that the verification of whether a statement holds can be done algorithmically.

We distinguish between three decidability problems, the

Type-Checking Problem ($\Gamma \vdash M : A$) asks whether the given typing judgment is valid. Here both the term M and the proposed type A are given. In the dependent type theory, type checking is *decidable*, this means there is an algorithm which determines whether such a judgment is correct.

Under Curry-Howard, type checking is proof checking. If A is a proposition and M is a proof, then checking $\Gamma \vdash M : A$ means checking that M is indeed a valid proof for A.

Type-Inhabitation Problem ($\Gamma \vdash ? : A$) asks whether there exists a term of a given type. Equivalently, under Curry-Howard, it asks whether the proposition A has a proof. In general, this problem is *undecidable*.

Type-Synthesis Problem ($\Gamma \vdash M : ?$) asks what the type of a given term is. In the pure, explicitly annotated dependent type theory, type synthesis is *decidable*.

1.3 Connection to Lean

Lean uses the Curry-Howard correspondence: propositions are types, and proofs are terms. A theorem in Lean is therefore a declaration whose type is a proposition and whose value is a proof term.

Decidability of type checking ensures that proof verification can be carried out algorithmically. A user constructs the proof, often interactively with tactics, and Lean checks each resulting proof term. This is why decidable type checking is enough to make Lean a reliable proof checker even though fully automatic theorem proving is impossible in general.

Unlike the simple presentation of the dependent type theory with the two sorts $*$ and $\Box$, Lean uses a hierarchy of universes, including 'Prop', 'Type', 'Type 1', and so on. This hierarchy avoids certain paradoxes and allows types themselves to be organized by levels.

2 Introduction to Lean

This chapter is intended not only as technical preparation, but also as a tutorial for readers to understand or extend the Lean file. It covers the use of keywords as well as helpful strategies and overviews for proving with Lean.

Lean already has a large library of mathematics (also set theory), called mathlib. However, in this formalization we develop the required set-theoretic constructions from scratch in order to stay close to the lecture contents from "Einführung in die Mathematische Logik".

2.1 Basic Lean Commands

We start by explaining basic Lean commands used in the formalization. Here is an overview of them, they are explained in detail below.

```
axiom name : return_type

def name (parameter_list) := body

prefix : precedence_num (priority := high) "symbol" => some_def_name

infix : precedence_num (priority := high) "symbol" => some_def_name

notation "symbol" => some_def_name

theorem name (parameter_list) : return_type := by proof
```

We can use the `axiom` keyword and write `axiom name : return_type` to introduce any object called "name" of the type "return_type". Default types we use are the type universe (`Type`), which is used to introduce new types and the proposition universe (`Prop`). By writing `axiom U : Type` we introduce `U` as a type.

The `def` keyword is used to define an object, it is not necessary to give a type. We just write `def name := body`. We can also give explicit arguments or the return type by writing `def name (parameter_list) : return_type := body`.
For example `def Emptyset : U → Prop := (fun _ ↦ ⊥)` gives us the object `Emptyset`, which has the class type and can be interpreted as the function that maps any argument (of type `U`) to `⊥`.
Or by `def memSC (a : U) (P : U → Prop) := P a`, we define a function `memSC` that takes two explicit arguments, named `a` and `P` with fixed types `U` and `U → Prop`, and applies `P` to `a`. This works since `a : U` has the input type of `P`. We do not specify it here, but the return type of `memSC` is therefore `Prop`.

We can also define notation symbols to get a more readable and faster writeable setting. We have different keywords for that:

`prefix`: to define functions with one input like
`prefix:70 (priority := high) "𝒫" => PowerClass`

`infix`: to define functions with two inputs like
`infix:50 (priority := high) "∈" => mem`,
so we can use `y∈x` instead of `mem x y`

`notation`: for everything else, either
symbols without any arguments like `notation "∅" => Emptyset`,
or more complex notation like
`notation "{" x ", " y "}" => PairClass x y`, where we give the arguments explicitly by the placeholders `x` and `y`

The numbers after `prefix:` and `infix:` are the precedence of the defined notation, which describes how strong those symbols are binding (higher number binds stronger) so `x ∈ ⋃ X` is interpreted as `x ∈ (⋃ X)` and not `(x ∈ ⋃) X` which is nonsense (not a propositional formula). The addition `(priority := high)` makes this notation declaration preferred during notation resolution, but Lean still resolves overloaded notation using the expected types of the arguments. Then we write the symbol as a string and after the `=>` we can add any definition/function name, that we want to substitute for the symbol. In case of `notation` we write a combination of strings and placeholder variables, which we then feed into the corresponding definition.

It is possible to overload notation symbols and they can work in parallel if the substitutes operate on different types, Lean can figure out the correct mapping by the input types.
For example `infix:50 (priority := high) "∈" => memSC` and
`infix:50 (priority := high) "∈" => mem` use the same `"∈"` symbol, but `mem` takes two arguments of the type `U` and `memSC` takes one input of type `U` and one of type `U → Prop`. So when all arguments are given, it is clear which one we meant. We can always add round brackets to make sure Lean picks up exactly the binding we intended (e.g. end a quantifier scope early, or change default priorities in a specific case).

The keyword `theorem` is used to indicate that the following statement needs a proof. The basic form is `theorem name (parameter_list) : return_type := by proof`. An example is:

```
theorem ReflEquivOfClasses (P : U → Prop) : P ≃ P := by
...
```

First we give our theorem the name `ReflEquivOfClasses` and the argument `(P : U → Prop)` (it is possible to give multiple arguments by writing something like `(R : U → Prop) (a b : U)`). After the `:` follows, as the return type, the statement `P ≃ P` and after `:= by` the proof.

Note: A proof can be omitted by writing `sorry` instead. In this case we get a warning, but the statement is already usable (be careful to write something that is true!). This

can be practical to sketch out a proof structure or add a helper lemma for a longer proof. This way, we can choose the order in which we write our proofs. In the end, everything has to be proved.

Note: The `by` indicates that our proof has multiple lines (for a one line proof you can just write `:=` and then give the proof (this will be most interesting for the `have` tactic).

2.2 Guidelines and Helper Techniques

This subsection collects practical conventions and proof strategies as well as recurring techniques that are useful for working with this formalization.

2.2.1 Term-Type Interpretations in Lean

In type theory, every object is understood as a term together with its type. Depending on the use case, this term-type relation can be interpreted differently. Also naming comes in handy, a definition `A := B` introduces `A` as a name for `B`. Whenever `A` is used, Lean can unfold it to `B`, and the type of `A` is determined by the type of `B`. This applies both to ordinary definitions and to theorems. For instance:

```
theorem ReflEquivOfClasses (P : U → Prop) : P ≃ P := by ...
```

introduces `ReflEquivOfClasses` as the name of a proof of the proposition `P ≃ P`. Thus, theorem names can be used later as proof terms. Inside a proof, a hypothesis such as `h : y ∈ x` means that `h` is an identifier for a proof of the proposition `y ∈ x`. Similarly, if `x : B`, then `x` is a term of type `B`. In mathematical language, one may call `x` a variable of type `B`.

2.2.2 Naming Conventions

Definitions and theorems are named descriptively using CamelCase. The name should indicate the mathematical content. For example, a theorem showing that the successor of an ordinal is again an ordinal has a name such as `SuccOfOrdIsInOrd`.

Inside proofs, the following naming conventions are used:

- sets, that is, objects of type `U`, are named with lowercase letters
- classes, that is, objects of type `U → Prop`, are named with capital letters
- corresponding set-class pairs usually share the same letter

- if two sets correspond to the same class `P`, they may be named `p` and `p'`
- if `S` is a class, then `s` denotes a corresponding set, while `s1` may denote a corresponding set for its successor `S plus_one`
- hypothesis are named `h`, use `h1, h2, h3, ...` and `h01, h02, ...`
- hypothesis may also have descriptive names like `h0x` for `zero∈X` and `hs` for a set-class correspondence like `∀ (c : U), c∈s ↔ S c`

Note: Although Lean allows names to be reused after their scope ends, the formalization avoids this in order to make proofs easier to read.

2.2.3 Proving

skills needed when proving:

- know the underlying propositional formulas of definitions (what logical symbols are contained?)
- know how to manipulate and use logical symbols
- be able to identify sub-statements to improve proof structure

What to do when stuck in a proof: When a proof becomes difficult, it is often helpful to first formulate a weaker or more specialized statement, this can reveal the structure of the argument or show why the stronger statement does not hold. Typical simplifications include:

- proving the statement for a specific object, such as 0, before proving it for all objects
- proving the statement for sets before proving the statement for classes
- replacing a complicated goal by a helper theorem that captures the essential substatement

The last strategy is especially useful in formal proofs because Lean requires every intermediate step to be made explicit. A helper theorem can therefore serve both as a mathematical simplification and as a way to improve the readability of the final proof.

When to Introduce Helper Theorems: Helper theorems are used to keep proofs readable and reusable. They clarify which intermediate facts are actually used and make the structure of the argument more transparent. A separate helper theorem is useful when:

- the same argument is needed multiple times
- the statement expresses an abstract fact that does not depend on the specific proof in which it first appears
- the current proof becomes too long or difficult to read

2.2.4 Common Tactics

The following table summarizes the tactics that occur in the formalization.

Tactics	Use case
to solve the goal	
`rfl`	solves definitionally equal goal ($A = A$, $B \leftrightarrow B$)
`exact`	solves current goal by providing a term of the required type
`tauto`	solves a semantically true propositional goal (e.g. b≃b, V x, a≃SetToClass a, x∈a ↔ SetToClass a x) or finds it in the context
to work on the goal	
`constructor`	splits $\wedge$ or $\leftrightarrow$ goals into subgoals
`intro`	introduces objects from $\forall$ or $\rightarrow$
`by_contra`	starts a proof by contradiction by assuming negation of goal
`ext`	replaces an $=$ goal by a point wise ($\leftrightarrow$) goal
`left`	selects left side of an $\vee$ goal
`right`	selects right side of an $\vee$ goal
`use`	provides a witness for an $\exists$ goal
to work on the context	
`obtain`	deconstructs $\exists$ or $\wedge$ hypotheses
`have`	introduces intermediate statement (with a proof)
`rcases`	starts case distinction from $\vee$
`let`	defines a local abbreviation
`.1` or `.2`	accesses first/second component of constructive types
to work anywhere	
`apply`	uses an implication (from context or a theorem) whose conclusion matches current goal, new goals are its assumptions
`push_neg`	pushes negations through logical formulas (e.g. $\forall, \exists, \wedge, \vee$)
`unfold`	replaces a defined symbol by its definition
`rewrite`	rewrites goal or hypothesis using $=$ or $\leftrightarrow$ from context

Note: When a tactic creates multiple goals, such as `constructor` or `rcases`, focus dots `.` and indentation can be used to separate the corresponding proof branches. This makes the structure of the proof more explicit and improves readability.

2.2.5 Notation

The following table lists the notation used in the formalization. Some symbols are overloaded, so Lean resolves their meaning from the types of their arguments. We use Lean's predefined logical symbols $\top, \bot, \exists, \forall, \wedge, \vee, \neg, \to, \leftrightarrow$. These symbols therefore retain their standard Lean meanings and are not interpreted as definitions introduced in this formalization. All other symbols are defined within the formalization.

Symbol	evaluates as	in Lean write
$\bot$		`\bot`
$\top$		`\top`
$\emptyset$	Emptyset	`\emptyset`, `\empty`
$\mathbb{N}$	NaturalNumbers	`\N`, `\nat`, `\bn`, `\Nat`, `\bbN`
$\neg$		`\neg`, `\not`, `\!`
$\wedge$		`\and`, `\an`, `\wedge`
$\vee$		`\v`, `\or`, `\vee`
$\exists$		`\exists`, `\ex`
$\forall$		`\forall`, `\all`
$\bigcup$	UnionClass	`\U`, `\Un`, `\Union`, `\bUnion`, `\bigcup`
$\bigcap$	IntersectionClass	`\I`, `\Inter`, `\bInter`, `\bigcap`
$\mathscr{P}$	PowerSet, PowerClass	`\MCP`
$\to$		`\r`, `\imp`, `\to`, `\->`, `\to`, `\r-`
$\leftrightarrow$		`\iff`, `\leftrightarrow`, `\<->`, `\lr`, `\lr-`
$\in$	mem, memSC, memCC	`\in`, `\member`, `\mem`
$\notin$	$\neg$ mem, $\neg$ memSC, $\neg$ memCC	`\notin`, `\nin`, `\inn`
$\simeq$	eqSC, eqCC	`\equiv`, `\simeq`, `\~-`
$\subseteq$	subclass	`\ss`, `\sub`, `\subset`, `\subseteq`, `\subseteqq`
$\cup$	BinaryUnionClass	`\union`, `\un`, `\cup`
$\cap$	BinaryIntersectionClass	`\i`, `\inter`, `\intersection`, `\cap`
$\times$	CartesianProduct	`\x`, `\times`, `\multiplication`
$\setminus$	ClassDifference	`\setminus`
$\{x, y\}$	PairClass x y	`{,}`
$\{x\}$	SingletonClass x	`{}`

3 Lecture Formalization

This chapter explains how the set-theoretic constructions from the lecture are represented in Lean. It gives the axioms and definitions in detail and states the formalized theorems, while the proofs themselves are contained in the accompanying Lean file. We begin with the basic representation of sets and classes, followed by two example proofs with detailed tactic explanations. We then introduce the ZF-axioms used in the lecture and develop standard constructions such as pairs, unions, powersets, relations, and functions. The

section concludes with the formalization of ordinals and the proof that the class of natural numbers is a set.

3.1 Prerequisites

```
import MIL.Common

noncomputable section

set_option quotPrecheck false
```

We start the file by writing `import MIL.Common` so we can use predefined keywords, symbols and tactics from MathematicsInLean [1]. We start with `noncomputable section`. The keyword `section` opens a local scope, while `noncomputable` allows later definitions in this section to use noncomputable constructions. This is okay here, since the file is intended for formalizing mathematics rather than for extracting executable code.
We also set `set_option quotPrecheck false`. This allows us to introduce overloaded notation that Lean resolves from the expected types of the arguments. In particular, we use the same symbols for several set/class variants of membership and equivalence. It is also possible to give input types as implicit arguments or to use some automated casting between sets and classes, but this is intentionally not done here.

Note: The formalization prioritizes correspondence with the lecture notation over automation and performance.

```
axiom U : Type

axiom mem : U → U → Prop

infix:50 (priority := high) "∈" => mem
infix:50 (priority := high) "∉" => ¬ mem
```

Using the `axiom` keyword we introduce a type `U` by writing `axiom U : Type`. In the formalization, elements of `U` are interpreted as sets. Thus, when `x : U`, we read `x` as a set. By writing `axiom mem : U → U → Prop`, we introduce a binary relation on `U`. We read `mem x y` as the proposition that the set `x` is an element of the set `y`. We also add notation symbols for this membership function by `infix:50 (priority := high)"∈" => mem` and for the negation by `infix:50 (priority := high) "∉" => ¬ mem`. Now we can start formalizing content from the lecture.

3.2 Basic Set-Theoretic Infrastructure

Note: In the following we will break up some lines of the Lean code to fit the pagewidth. Also we will not give proofs of many of the theorems here, but they are all included in the Lean file. We just state `theorem name input : proposition` instead of `theorem name input : proposition := by proof`.

3.2.1 Basic Definitions

In the lecture, all quantified variables range over sets. Classes are not objects of the first-order language itself, we say the symbol sequence

$$\{v_i : \varphi\}$$

where φ is an $\in$-formula, is called a **class**. In Lean, we represent sets by elements of an abstract type `U`. Thus, if `x : U`, then `x` is read as a set. Classes are represented by predicates on sets, that is, by terms of type `U → Prop`. If `P : U → Prop` and `x : U`, then `P x` is the proposition that `x` belongs to the class `P`. We start by building a bit of infrastructure to be able to check memberships and equivalences. Also we define some notation for easier use. We did something similar in the lecture and added some abbreviations for membership and equality.

Set in class membership: In the lecture we write

$$v_j \in \{v_i \ : \ \varphi\}$$

for the $\in$-formula $[\varphi]^{\frac{v_j}{v_i}}$, where v_j denotes a set and $\{v_i : \varphi\}$ denotes a class.

To check if a set is a member of a class, we can use the natural property of the `U → Prop` type and directly give that set as an input. Thus, the membership function uses the symbol `∈` and takes two arguments, a set `a` and a class `P`.

```
def memSC (a : U) (P : U → Prop) :=
P a

infix:50 (priority := high) "∈" => memSC
infix:60 (priority := high) "∉" => ¬ memSC
```

Set to class equality: In the lecture we write

$$v_j = \{v_i \ : \ \varphi\}$$

for the $\in$-formula $\forall v_k(v_k \in v_j \leftrightarrow [\varphi]^{\frac{v_k}{v_i}})$, where v_j denotes a set and $\{v_i : \varphi\}$ denotes a class.

Since sets and classes have different types in Lean, we cannot use Lean's ordinary equality symbol to compare them directly. Instead, we introduce extensional equivalence relations.

A set is equivalent to a class if it has exactly the elements described by that class. We define this the same way the $\in$-formula indicates, by saying that a set v_k is in the set v_j if and only if it is in the class $\{v_i : \varphi\}$. The equivalence function uses $\simeq$ in infix notation, with the set on the left and the class on the right.

```
def eqSC (a : U) (P : U → Prop) :=
∀ c : U, c ∈ a ↔ P c

infix:50 (priority := high) "≃" => eqSC
```

Class in class membership: In the lecture we write

$$\{v_j \ : \ \psi\} \in \{v_i \ : \ \varphi\}$$

for the $\in$-formula $\exists v_k([\varphi]\frac{v_k}{v_i} \wedge v_k = \{v_j \ : \ \psi\})$, where $\{v_j : \psi\}$ and $\{v_i : \varphi\}$ both denote classes.

When looking at the $\in$-formula, we can already see that we need a set v_k that is equivalent to the first class $\{v_j : \psi\}$ and member of the second class $\{v_i : \varphi\}$. For class in class membership we again use the $\in$ symbol with two classes as arguments.

```
def memCC  (P Q : U → Prop) :=
∃ a : U, Q a ∧ ∀ c : U , c ∈ a ↔ P c

infix:50 (priority := high) "∈" => memCC
infix:60 (priority := high) "∉" => ¬ memCC
```

Class to class equality: In the lecture we write

$$\{v_j \ : \ \psi\} = \{v_i \ : \ \varphi\}$$

for the $\in$-formula $\forall v_k([\psi]\frac{v_k}{v_j} \leftrightarrow [\varphi]\frac{v_k}{v_i})$, where $\{v_j : \psi\}$ and $\{v_i : \varphi\}$ both denote classes.

As the $\in$-formula already hints, we again cannot use equality. This time since classes are represented by predicates of type `U → Prop`, so we can only compare their outputs for every input, the output type is `Prop` so we use logical bi-implication to do this. We say two classes are equivalent if they contain exactly the same sets. We use $\simeq$ in infix notation, with two classes as inputs.

```
def eqCC (P Q : U → Prop) :=
∀ c : U , P c ↔ Q c

infix:50 (priority := high) "≃" => eqCC
```

Subclass: In the lecture we write

$$X \subseteq Y$$

for $\forall v_k \in X\ v_k \in Y$, where X and Y both denote classes.

For two classes `P` and `Q`, we say `P` is a subclass of `Q`, if for all sets `a` in `P`, `a` is also in `Q`. We use `⊆` in infix notation.

```
def subclass (P Q : U → Prop):=
∀ a : U, P a → Q a

infix:50 (priority := high) " ⊆ " => subclass
```

Our first specific class in the lecture is the **emptyset** which is defined by $\emptyset := \{v_0 \ : \ \bot\}$, we translate this into a function that takes any argument of the type `U` (so any set) and returns `⊥`. So it is the class that contains no set. We also add notation to write `∅` instead of `Emptyset`.

```
def Emptyset : U → Prop := (fun _ ↦ ⊥)

notation "∅" => Emptyset
```

In the lecture, the **universe** is introduced as $V := \{v_0 : \top\}$. It is formalized as a function, that maps any element of type `U` to `⊤`. So it is the class that contains all sets.

```
def V : U → Prop := (fun _ ↦ ⊤)
```

In the lecture we say for an $\in$-formula φ the symbol sequence $\{v_i : \varphi\}$ is

a **set**, if $\text{ZF} \vdash \{v_i \ : \ \varphi\} \in V$

a **proper class**, if $\text{ZF} \vdash \{v_i \ : \ \varphi\} \notin V$

Using a class `P` that plays the role of $\{v_i \ : \ \varphi\}$, we can formalize it as intended.

```
def IsSet (P : U → Prop) := P ∈ V
def ProperClass (P : U → Prop) := ¬ P ∈ V
```

Here the `∈` symbol is interpreted as `memCC` since both `P` and `V` are of type `U → Prop`.

Every set can be interpreted as a class, to formalize this in Lean we write a function `SetToClass`. It takes sets as an input and adds an object of the class type, that contains the same elements as that set.

```
def SetToClass (a : U) := (fun c : U ↦ c ∈ a)
```

3.2.2 Basic Theorems

In this chapter we will cover the first theorems, we can prove without the use of any ZF-axioms.

Let us have a detailed look at a proof in Lean. This is done at the example of extensionality for classes (two classes P and Q are equivalent if and only if P is a subclass of Q and Q is a subclass of P).

```
theorem ExtForClasses (P Q : U → Prop) :
                P ≃ Q ↔ (P ⊆ Q ∧ Q ⊆ P) := by
  constructor
  . intro h
    constructor
    . intro c
      exact (h c).1
    . intro c
      exact (h c).2
  . intro h
    intro c
    exact ⟨(h.1) c, (h.2) c⟩
```

We start the proof with a context that has all arguments as objects, here two classes `P` and `Q` both of type `U → Prop`. We need to prove the given bi-implication.

`constructor` changes the goal into two subgoals, each of them one direction of `↔`. The active goal is now `P ≃ Q → (P ⊆ Q ∧ Q ⊆ P)`.

`intro h` adds `h : P ≃ Q` to the context. `h` is the name of the hypothesis and called the **identifier** of `P ≃ Q` which itself was the precondition of the implication we need to show. Equivalence of two classes, was defined as `∀ c : U , P c ↔ Q c`. And the goal changed to `P ⊆ Q ∧ Q ⊆ P`.

`constructor` changes the active goal into two subgoals, both sides of the `∧` need to be shown. We now have three goals with `P ⊆ Q` as the active one. Having a look at the subclass definition shows that this notation means `∀ a : U, P a → Q a`.

`intro c` adds `c : U` to the context. We take an arbitrary set and named it `c`, we want to use it with `h`. The goal is now `P c → Q c`.

`exact (h c).1` closes the active goal. By using `c` at `h` we know that `P c ↔ Q c`. As we saw in the first proof step the `↔` is a constructive type and we can use `.1` and `.2` to access each of the implications. We can enter the next proof branch (without focus dots this happens automatically). The next goal is `Q ⊆ P`. The context 'resets' to the state it was in when that goal was created.

`intro c` adds `c : U` to the context. This will be used similar to the old one. The goal is now `Q c → P c`.

`exact (h c).2` closes the active goal. By using `c` at `h` we know that `P c ↔ Q c`. This time we need the back implication and use `.2`. The active goal changes to the implication `P ⊆ Q ∧ Q ⊆ P → P ≃ Q`, the context again 'resets' and only has `P` and `Q` both of type `U → Prop`.

`intro h` adds `h : P ⊆ Q ∧ Q ⊆ P` to the context, which is the conjunction of two subset statements, namely `∀ a : U, P a → Q a` and `∀ a : U, Q a → P a`. The goal changes to `P ≃ Q` which was defined as `∀ c : U, P c ↔ Q c`.

`intro c` adds `c : U` to the context, which we want to use inside `h`. The goal changes to `P c ↔ Q c`.

Note: we can also write `intro h c` which does both in one line. When writing, it is more intuitive to just use one intro, as you can see the goal change. But when you want to have more visible goal changes when clicking and reading through a finished file, more lines are preferred.

`exact ⟨(h.1) c, (h.2) c⟩` closes the active (and last) goal. By using `⟨ , ⟩` we can give both elements that are needed to build the goal, which has a constructive type. Using both parts of `h` and feeding `c` into them we can solve the goal.

Using the same tactics in similar ways we can also show symmetry of class to class equivalence

```
theorem SimEquivOfClasses (P Q : U → Prop) : P ≃ Q ↔ Q ≃ P
```

and reflexivity of class to class equivalence.

```
theorem ReflEquivOfClasses (P : U → Prop) : P ≃ P
```

We need some new tactics to show that if two classes are equivalent and one is a set, then the other is also a set.

```
theorem EquivOfSetIsSet (P Q : U → Prop) :
                (P ∈ V ∧ P ≃ Q) → Q ∈ V := by
  intro h1
  obtain ⟨p, hp⟩ := h1.1
  use p
  constructor
  . tauto
  . intro y
    constructor
    . intro h2
      apply (h1.2 y).1
      apply (hp.2 y).1
      exact h2
    . intro h3
      apply (hp.2 y).2
      apply (h1.2 y).2
      exact h3
```

We start the proof with a context that has two classes `P` and `Q` both of type `U → Prop`. We need to prove the given implication.

`intro h1` adds `h1 : P∈V ∧ P≃Q` to the context. The goal changes to `Q ∈ V`.

`obtain ⟨p, hp⟩ := h1.1` splits `P∈V` and therefore adds a set `p : U` to the context that corresponds to `P` and adds `h1 : V p ∧ ∀ (c : U), c∈p ↔ P c` to the context, which says that `p` is a set and `p≃P`.

`use p` The formula `Q ∈ V` means `∃ a : U, V a ∧ ∀ c : U , c ∈ a ↔ Q c`. So we need to give an element with type `U`, that is equivalent to the class `Q`. Since one of our assumptions is `P≃Q`, we use `p` as a witness for this. The goal changes to `V p ∧ ∀ (c : U), c∈p ↔ Q c`.

`constructor` splits the goal into `V p` and `∀ (c : U), c∈p ↔ Q c`.

`tauto` solves `V p`. The definition of `V` was `def V : U → Prop := (fun _ ↦ ⊤)` and since `p` has type `U`, `V p` evaluates as `⊤`.

`intro y` specifies a set `y : U`, we need to show `y∈p ↔ Q y`.

`constructor` splits the `↔` into `y∈p → Q y` and `Q y → y∈p`.

`intro h2` adds `h2 : y∈p` to the context, while the goal is now `Q y`.

`apply (h1.2 y).1` uses `P≃Q`, given by `h1.2`, to apply the implication `P y → Q y` to the goal, `P y ↔ Q y` The goal changes to the assumption of the implication `P y`.

`apply (hp.2 y).1` uses `p≃P`, given by `hp.2`, to apply `y∈p → P y`. The goal changes to `y∈p`, which we already have in the context.

`exact h2` solves this.

`intro h3` adds `h3 : Q y` to the context, while the goal is now `y∈p`.

`apply (hp.2 y).2` uses `p≃P`, by applying `P y → y∈p` to the goal, which changes to `P y`.

`apply (h1.2 y).2` uses `P≃Q`, by applying `Q y → P y` to the goal, which changes to `Q y`, which is already given by `h3`.

`exact h3` solves this.

Using those tactics, we can also show that every element of a class is a set.

```
theorem ElementofClassIsSet (P Q : U → Prop) :
    (P∈Q) → (P∈V)
```

3.3 ZF-Axioms

Here is an overview of the axioms that were used in the lecture with the corresponding formalization. When not specified otherwise the variable names translate as: v_0 - `a`, v_1 - `b`, v_2 - `c` and v_3 - `d`.

Axiom of the existence: There is a set that contains no other set.

> (Ex) $\exists v_0 \neg \exists v_1 v_1 \in v_0$

Where v_0 plays the role of the emptyset and v_1 is universally quantified due to $\neg\exists$. When formalizing we push this negation into the formula for readability and easier usage.

```
axiom ax_emptyset : ∃ a : U, (∀ b : U, ¬ b∈a)
```

Axiom of extensionality: Every set is defined through its elements.

> (Ext) $\forall v_0 \forall v_1 (\forall v_2 (v_2 \in v_0 \leftrightarrow v_2 \in v_1) \rightarrow v_0 = v_1)$

Where v_0 and v_1 are arbitrary sets we want to compare and v_2 is the variable for their elements.

```
axiom ax_ext : ∀ a b : U, (∀ c : U, c∈a ↔ c∈b) → a=b
```

Axiom of pairing: For two given sets there exists a set that contains exactly those as elements.

(Paar) $\forall v_0 \forall v_1 \exists v_2 \forall v_3 (v_3 \in v_2 \leftrightarrow (v_3 = v_0 \vee v_3 = v_1))$

Where v_0 and v_1 are arbitrary sets we want to pair, v_2 will be the pair-set and v_3 quantifies over the elements of v_2.

```
axiom ax_pair : ∀ a b : U, ∃ c : U, ∀ d : U,
    d∈c ↔ (d=a ∨ d=b)
```

Axiom of union: For a given set there exists its union, a set that contains exactly all elements of all elements of the given set.

($\bigcup$-Ax) $\forall v_0 \exists v_1 \forall v_2 (v_2 \in v_1 \leftrightarrow \exists v_3 (v_3 \in v_0 \wedge v_2 \in v_3))$

Where v_0 is an arbitrary set we want to build the union of, v_1 will be the union and v_2 quantifies over the elements of the union-set. $\exists v_3 (v_3 \in v_0 \wedge v_2 \in v_3)$ says v_2 is an element of an element of the original set.

```
axiom ax_union : ∀ a : U, ∃ b : U, ∀ c : U,
    (∃ d : U, (c∈d ∧ d∈a)) ↔ c∈b
```

Axiom of specification: We can cut down a given set, such that all remaining elements satisfy a given property.

(Aus) Let φ be a $\in$-formula, where the variables v_{n+1} and v_{n+2} do not occur,

$$\forall v_1 \cdots \forall v_n \, \forall v_{n+1} \exists v_{n+2} \forall v_0 (v_0 \in v_{n+2} \leftrightarrow (v_0 \in v_{n+1} \wedge \varphi))$$

Where $v_0, \ldots, v_n$ are variables that may occur in φ and $v_1, \ldots, v_n$ can be arbitrary, v_{n+1} is an arbitrary set that we want to cut down, v_{n+2} will be the specified set and v_0 quantifies over its elements.

Variable names: v_{n+1} - `a`, v_{n+2} - `b`, v_0 - `c` and φ - `P`.

```
axiom ax_spec (P : U → Prop) : ∀ a : U, ∃ b : U, ∀ c : U,
    c∈b ↔ c∈a ∧ P c
```

We formalize the specification by using a class `P`, so we do not need the variables $v_1, \ldots, v_n$, and we can input v_0 - `c` by using the properties of the type of `P : U → Prop` and interpreting `P` as a function.

Axiom of powerset: For any given set there exists its powerset, a set that contains exactly all subsets of the original set as elements.

$$\text{(Pot) } \forall v_0 \exists v_1 \forall v_2 (v_2 \in v_1 \leftrightarrow \underbrace{\forall v_3 (v_3 \in v_2 \rightarrow v_3 \in v_0)}_{v_2 \subseteq v_0})$$

Where v_0 is an arbitrary set we want to build the powerset of, v_1 will be the powerset and v_2 quantifies over the subsets. The sequence $\forall v_3(v_3 \in v_2 \rightarrow v_3 \in v_0)$ means $v_2 \subseteq v_0$.

```
axiom ax_power : ∀ a : U, ∃ b : U, ∀ c : U,
    c∈b ↔ (∀ d : U, d∈c → d∈a)
```

Axiom of infinity: There exists a set with infinitely many elements.

$$\text{(Inf) } \exists v_0 (\underbrace{\exists v_1 (v_1 \in v_0 \wedge \forall v_2 \neg v_2 \in v_1)}_{\emptyset \in v_0} \wedge$$
$$\forall v_1 \exists v_2 (v_1 \in v_0 \rightarrow (v_2 \in v_0 \wedge \underbrace{\forall v_3 (v_3 \in v_2 \leftrightarrow (v_3 \in v_1 \vee v_3 = v_1))}_{v_2 = v_1 + 1})))$$

Where v_0 will be the set with infinitely many elements. Induction is used here and the variables v_1 and v_2 are used independently in both cases.

Induction start: The first v_1 and v_2 are used in $(\exists v_1 (v_1 \in v_0 \wedge \forall v_2 \neg v_2 \in v_1)$ to say that the emptyset (v_1) is an element of v_0.
Induction step: Now we want to 'generate' infinitely many elements. The second v_1 is an arbitrary element of which we know that it is in v_0 and v_2 will be its successor. $\forall v_3 (v_3 \in v_2 \leftrightarrow (v_3 \in v_1 \vee v_3 = v_1))$ is the successor property.

Variable names: For the first part v_0 - `a`, v_1 - `b` and v_2 - `c`. For the second part v_1 - `d`, v_2 - `e` and v_3 - `f`.

```
axiom ax_inf : ∃ a : U, ((∃ b : U, b∈a ∧ ∀ c : U, ¬ c∈b)
    ∧ (∀ d : U, ∃ e : U, d∈a → (e∈a
        ∧ (∀ f : U, f∈e ↔ (f∈d ∨ f=d)))))
```

Axiom of replacement: The function image of a set is also a set.

(Ers) Let φ be a $\in$-formula, where the variables v_{n+2} and v_{n+3} do not occur,

$$\forall v_2 \dots \forall v_{n+1} \underbrace{(\forall v_0 \forall v_{n+2} \forall v_{n+3} (([\varphi]\frac{v_{n+2}}{v_1} \wedge [\varphi]\frac{v_{n+3}}{v_1}) \to v_{n+2} = v_{n+3})}_{\varphi \text{ defines a function}}$$
$$\to \forall v_{n+2} \exists v_{n+3} \underbrace{\forall v_1 (v_1 \in v_{n+3} \leftrightarrow \exists v_0 (v_0 \in v_{n+2} \wedge \varphi)))}_{v_{n+3} \text{ is image of } v_{n+2}}$$

Where v_0 is the variable for elements of the definition space and v_1 is the variable for the image space. So when $[\varphi]\frac{a\ b}{v_0\ v_1}$ evaluates as true, b is the image of a under the function defined by φ. $v_2, \dots, v_{n+1}$ may occur in φ and are other parameters to define the function.

In $(\forall v_0 \forall v_{n+2} \forall v_{n+3} (([\varphi]\frac{v_{n+2}}{v_1} \wedge [\varphi]\frac{v_{n+3}}{v_1}) \to v_{n+2} = v_{n+3})$ the variable v_0 is an arbitrary element of the definition space and is automatically fed to φ and the variables v_{n+2} and v_{n+3} are two initially different images of v_0, but we require them to be equal, for φ to define a function.

In $\forall v_{n+2} \exists v_{n+3} \forall v_1 (v_1 \in v_{n+3} \leftrightarrow \exists v_0 (v_0 \in v_{n+2} \wedge \varphi))$ we take an arbitrary set v_{n+2} and v_{n+3} will be its image set under φ.

Variable names: For the first part v_0 - `a`, v_{n+2} - `b` and v_{n+3} - `c`. For the second part v_{n+2} - `d` and v_{n+3} - `e`, v_1 - `b` and v_0 - `a`.

```
axiom ax_repl : ∀ p : U → U → Prop,
    ((∀ a : U, ∀ b : U, ∀ c : U, ((p a b ∧ p a c) → b=c))
    → ∀ d : U, ∃ e : U,
        ∀ b : U, (b∈e ↔ ∃ a : U, (a∈d ∧ p a b)))
```

We formalize the function with `p : U → U → Prop`. If `p` gets as a first argument a set from its definition space and as second the corresponding image set, it evaluates as true, else as false. So again we do not need the variables $v_2, \dots, v_{n+1}$. In the $\in$-formula specific choices of v_0 and v_1 are automatically put into φ, in the formalization, we need to feed them in explicitly by writing e.g. `p a b`.

Axiom of regularity: For a given property, if there exists a set that satisfies this property, then there exists a set that satisfies the property and all its elements do not satisfy the property.

(Fund) Let φ be a $\in$-formula, where the variable v_{n+1} does not occur,

$$\forall v_1 \dots \forall v_n (\exists v_0 \varphi \to \exists v_0 (\varphi \wedge \forall v_{n+1} (v_{n+1} \in v_0 \to \neg[\varphi]\frac{v_{n+1}}{v_0})))$$

Where v_0 is the variable for sets that satisfy the property defined by φ. The variables $v_1, \ldots, v_n$ may occur in φ. The variable v_{n+1} quantifies over the elements of v_0.

Variable names: For the first part v_0 - `a` and for the second part v_0 - `b`, v_{n+1} - `c`.

```
axiom ax_reg : ∀ P : U → Prop, ((∃ a : U, P a)
    → (∃ b : U, (P b ∧ ∀ c : U, (c∈b → ¬ (P c)))))
```

We use `P : U → Prop` to define a property and can therefore omit the variables $v_1, \ldots, v_n$. To substitute v_0 - `b` or v_{n+1} - `c` we use `P` as a function and write e.g. `P b`.

By using the axioms `ax_emptyset` and `ax_spec` we can show: The emptyset is indeed a set

```
theorem EmptysetIsASet : ∅ ∈ V
```

and the universe is a proper class.

```
theorem VIsProperClass : ¬ V ∈ V
```

3.4 Basic Constructions in Set Theory

Notation: $X, Y, X_1, X_2, A, F, R,$ denote classes and $v_0, v_1, v_2, v_i, v_j, v_k, v_\ell$ denote sets. In Lean we use `A, F, P, Q, R, S, X, Z` for classes and `a, b, c, ...` for sets.

3.4.1 Quantifiers

When using quantifiers in the lecture, we have a class A, a variable v_i, that does not occur in A and a $\in$-formula φ, we write

$$\exists v_i \in A\ \varphi \text{ for } \exists v_i(v_i \in A \wedge \varphi)$$
$$\forall v_i \in A\ \varphi \text{ for } \forall v_i(v_i \in A \to \varphi)$$

For the class A we can use a class `A : U → Prop` and for the set variable v_i we can use a set `c : U`. For the $\in$-formula we have to introduce a new class `P : U → Prop`, by defining it as a function that maps any input `x` to a corresponding $\in$-formula where `x` plays the role of v_i. Then writing `P c` will have the same effect as writing φ on paper. We use the default quantifiers given by Lean, so for the classes `A : U → Prop` and `P : U → Prop` and a set `c : U` we would write `∃ c, A c ∧ P c` and `∀ c, A c → P c` or, with an explicit input type, `∃ c : U, A c ∧ P c` and `∀ c : U, A c → P c`.

3.4.2 Singleton & Pair

In the lecture, we introduce the concept of giving a class explicitly by writing out its elements. To do this we define notation for a class that contains exactly n-many given classes. In the formalization we do not have the concept of n-many, we can only pick a specific number and write a formalization for it. So only the cases for $n = 1, 2$ are done. In the lecture we write

$$\{X_1\} \text{ for } \{v_k \ : \ v_k = X_1\}$$

For the formalization we use equivalence and not equality, since we work with a class here.

```
def SingletonClass (P : U → Prop) := (fun c : U ↦ c ≃ P)

notation "{" x "}" => SingletonClass x
```

The singleton class `{P}` of a given class `P` is defined as a function that takes sets and returns true for every set `c` that is equivalent to `P`. The added notation for the singleton class allows us to write `{P}` from now on.

Warning: Since membership in a class is always witnessed by a set representative, a proper class has no representative set. Hence the singleton class of a proper class is extensionally equivalent to the empty class. So if `P ∈ V`, then `P ∈ {P}` is true. In contrast, when `¬ P ∈ V`, then `{P} ≃ ∅` and `P ∈ {P}` is false. We can prove that the singleton class of a proper class, it is equivalent to the empty class:

```
theorem ProperClassSingletonEqEmptyset (P : U → Prop) :
    ¬ P ∈ V → ({P} ≃ ∅)
```

For $n = 2$ we have the notation

$$\{X_1, X_2\} \text{ for } \{v_k \ : \ (v_k = X_1 \vee v_k = X_2)\}$$

in the lecture, which translates in a similar manner.

```
def PairClass (P Q : U → Prop) :=
    (fun c : U ↦ (c ≃ P) ∨ (c ≃ Q))

notation "{" x ", " y "}" => PairClass x y
```

We can now prove some properties of the singleton class and the pair class. The pair class is symmetric, since it is defined via `∨`, which is symmetric itself.

```
theorem PairClassSymmetric (P Q : U → Prop) :
    {P,Q} ≃ {Q,P}
```

If two classes are equivalent, their corresponding singleton classes are also equivalent.

```
theorem SingletonEqIfClassesEq (P Q : U → Prop) :
    (P ≃ Q) → ({P} ≃ {Q})
```

The converse does not hold for arbitrary classes, since the singelton class of a proper class is equivalent to the emptyset (see `ProperClassSingletonEqEmptyset`). However, it does hold when both classes are sets. We use here `∈ V` as the determining property of being a set, so that we can use objects of the class type.

```
theorem SetsEqIfSingletonsEq (P Q : U → Prop) :
    P ∈ V → ({P} ≃ {Q}) → (P ≃ Q)
```

When we build the pair class of two sets, then the corresponding pair class is also a set.

```
theorem PairClassIsSetForSets (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → {P,Q} ∈ V
```

The same holds for the singleton class. The singleton class of a set, is itself a set. To prove this we use `PairClassIsSetForSets` and the following helper theorem.

When building the pair class of a class `P` with itself, it is the same as taking the singleton class of `P`.

```
theorem SingletonIsPair (P : U → Prop) :
    {P,P} ≃ {P}
```

```
theorem SingletonClassIsSetForSets (P : U → Prop) :
    P ∈ V → {P} ∈ V
```

3.4.3 Ordered Pair Class

Having the singleton class and the pair class, we can now define an ordered pair. Let X and Y denote classes, in the lecture we write

$$(X, Y) \text{ for } \{\{X\}, \{X, Y\}\}$$

Then (X, Y) is called the **ordered pair** of X and Y. This translates simply to:

```
def OrderedPairClass (P Q : U → Prop) := {{P},{P,Q}}
```

Warning: As for singleton classes, this definition behaves as expected only when the classes involved are sets. If one of the arguments is a proper class, the ordered-pair construction no longer corresponds to the intended ordered pair object.

We also want to be able to use an ordered pair on sets with a set type, so we also add such a class.

```
def OrderedSetPairClass (a b : U) :=
    OrderedPairClass (SetToClass a) (SetToClass b)
```

We can show, that the ordered pair of two sets is a set itself.

```
theorem OrderedPairClassIsSetForSets (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → OrderedPairClass P Q ∈ V
```

3.4.4 Unions

To build the union of two classes X and Y, where v_k does not occur in neither X nor Y, we wrote

$$X \cup Y \text{ for } \{v_k \ : \ (v_k \in X \vee v_k \in Y)\}$$

Since a class is represented as a predicate, membership in the class is expressed by function application, so we write `P c` for $c \in P$.

```
def BinaryUnionClass (P Q : U → Prop) :=
    (fun c : U ↦ P c ∨ Q c)

infix:50 (priority := high) "∪" => BinaryUnionClass
```

The added infix notation allows us to write `P ∪ Q`.

We can also take the union of a single class. Let X denote a class and v_i, v_j variables, that do not occur in X. We wrote

$$\bigcup X \text{ for } \{v_k \ : \ \exists v_i \in X \ v_k \in v_i\}$$

Then $\bigcup X$ is called the **union** of X. This simply translates to:

```
def UnionClass (P : U → Prop) :=
    (fun c : U ↦ ∃ a : U, P a ∧ c ∈ a)

prefix:60 (priority := high) "⋃" => UnionClass
```

And the added prefix notation allows us to write ⋃ P.

The union of a set is also a set.

```
theorem UnionClassIsSetForSets (P : U → Prop) :
    P ∈ V → (⋃ P) ∈ V
```

When given two sets P and Q (in class notation), the union of P and Q is equivalent to the union of the class that has exactly P and Q as elements.

```
theorem BinaryUnionAsUnion (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → (P ∪ Q) ≃ ⋃ {P,Q}
```

The union of two sets is also a set.

```
theorem BinaryUnionClassIsSetForSets (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → (P ∪ Q) ∈ V
```

3.4.5 Power

In the lecture we write

$$\mathcal{P}(X) \text{ for } \{v_k \ : \ v_k \subseteq X\}$$

where X denotes a class and v_k a variable that does not occur in X. We call $\mathcal{P}(X)$ the **powerclass** of X.

```
def PowerClass (P : U → Prop) :=
    (fun c : U ↦ ∀ b : U, (b ∈ c → P b))

prefix:70 (priority := high) "𝒫" => PowerClass
```

It is formalized as a function, that takes any set c : U and evaluates as true, if c is a subset of the input class P : U → Prop, we do not use the subclass symbol (⊆), since we only defined that for two classes, and c is a set. We just write the definition out. We also add notation that allows us to write 𝒫 P.

We can show that the power class of a set is also a set.

```
theorem PowerClassIsSetForSets (P : U → Prop) :
    P ∈ V → 𝒫 P ∈ V
```

We can show some additional properties of the powerclass that we later use to prove `CartesianProductIsSetForSets`.

If the singleton class of a set `P` is an element of the power class of a class `Q`, then `P` is an element of `Q`.

```
theorem SingletonOfPowerClass (P Q : U → Prop) :
    P ∈ V → {P} ∈ 𝒫 Q → P ∈ Q
```

Similarly, if the pair class of a set `P` and a class `R` is an element of the power class of a class `Q`, then `P` is an element of `Q`.

```
theorem PairOfPowerClass (P Q R : U → Prop) :
    P ∈ V → {P,R} ∈ 𝒫 Q → P ∈ Q := by
```

If one class `P` is an element of a class `Q`, then the singleton class of `P` is an element of the power class of `Q`.

```
theorem SingletonClassIsInPowerSetForSets (P Q : U → Prop) :
    (P ∈ Q) → {P} ∈ 𝒫 Q
```

Similarly, if two classes `P` and `Q` are both element of a class `R`, then the pair class of `P` and `Q` is an element of the power class of `R`.

```
theorem PairClassIsInPowerSetForSets (P Q R : U → Prop) :
    (P ∈ R ∧ Q ∈ R) → {P,Q} ∈ 𝒫 R
```

3.4.6 Cartesian Product

We also defined the cartesian product of two classes. Let X and Y denote classes and v_k, v_i, v_j variables that do not occur in X nor Y. In the lecture we write

$$X \times Y \text{ for } \{v_k \ : \ \exists v_i \exists v_j ((v_i \in X \wedge v_j \in Y) \wedge v_k = (v_i, v_j)\}$$

This translates to:

```
def CartesianProduct (P Q : U → Prop) :=
    (fun c : U ↦ ∃ a b : U, P a ∧ Q b ∧
        c ≃ OrderedSetPairClass a b)

infix:80 (priority := high) " × " => CartesianProduct
```

In the beginning we can omit one ∃ quantifier symbol since one is enough to catch both `a` and `b`. And we have to switch from equality to equivalence, since the ordered pair class is a class. The added notation allows us to write `P × Q`.

We can show the cartesian product of two sets is also a set. To do this we need one more helper theorem:

The cartesian product of two classes `P` and `Q` is a subclass of the power class of the powerclass of the union of `P` and `Q`.

```
theorem CartesianProductSubclassOfPowerClass2 (P Q : U → Prop) :
    (P × Q) ⊆ (𝒫 𝒫(P∪Q))
```

```
theorem CartesianProductIsSetForSets (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → (P × Q) ∈ V
```

3.4.7 Relations & Functions

Now we can formalize the concept of relations. Given a class R, in the lecture we write

$$\text{Rel}(R) \text{ for the } \in\text{-formula } R \subseteq V \times V$$

which means that every set in R is the ordered pair of two other sets. We call R a **binary relation** if $\vdash \text{Rel}(R)$. To check if a given class is a relation, we check if all its elements are equivalent to an ordered pair.

```
def IsRelation (R : U → Prop) : Prop := R ⊆ V × V
```

We also write

$$v_i R v_j \text{ for } (v_i, v_j) \in R$$

To check if two sets `a : U` and `b : U` are in relation with respect to a class `R : U → Prop`, we check if `R` is a relation, and if the ordered pair of `a` and `b` is in `R`.

```
def InRelation (R : U → Prop) (a b : U) : Prop :=
    (IsRelation R ∧ (OrderedSetPairClass a b) ∈ R)
```

In the lecture we write

$$\text{Dom}(R) \text{ for } \{v_i \ : \exists v_j \ v_i R v_j\}$$

to get the domain of a relation R. Given a class `R : U → Prop`, this returns a class, that contains all sets `a : U` for that there exists a set `b : U` such that `a` is in relation with `b` with respect to `R`.

```
def RelationDomain (R : U → Prop) : U → Prop :=
    (fun a : U ↦ ∃ b : U, InRelation R a b)
```

We write

$$\text{Bild}(R) \text{ for } \{v_j \ : \exists v_i \ v_i R v_j\}$$

to get the image of a relation R. Given a class `R : U → Prop`, this returns a class, that contains all sets `b : U` for that exists a set `a : U` such that `a` is in relation with `b` with respect to `R`.

```
def RelationImage (R : U → Prop) : U → Prop :=
    (fun b : U ↦ ∃ a : U, InRelation R a b)
```

We can also take the image of a class. Let A be another class. In the lecture we write

$$R[A] \text{ for } \{v_j \ : \exists v_i \in A \ v_i R v_j\}$$

This translates in a similar manner.

```
def RelationClassImage (R A : U → Prop) : U → Prop :=
    (fun b : U ↦ ∃ a : U, A a ∧ InRelation R a b)
```

By using relations we can finally define what a function is. We write

$$\text{Fun}(F) \text{ for the } \in\text{-formula } \text{Rel}(F) \wedge \forall v_i \forall v_j \forall v_k ((v_i F v_j \wedge v_i F v_k) \to v_j = v_k)$$

This means F is a **function**. This translates as:

```
def IsFunction (F : U → Prop) : Prop :=
    (IsRelation F ∧ ∀ a : U, ∀ b : U, ∀ c : U,
        (InRelation F a b ∧ InRelation F a c) → b = c)
```

First we need to check, that `F` is indeed a relation. Then we can check if each input has at most one output.

In the lecture we write

$$F(v_i) \text{ for } \{v_k \ : \ \forall v_j(v_i F v_j \to v_k \in v_j)\}$$

This means, if F is a function, then $F(v_i)$ is the function value of F at v_i.

```
def FunctionValue (F : U → Prop) (a : U) : U → Prop :=
    (fun c : U ↦ IsFunction F ∧
        (∀ b : U, (InRelation F a b → c ∈ b)))
```

First we check, that the given class `F` is indeed a function. Then we can put the $\in$-formula.

We can prove, that the function image of a set is again a set.

```
theorem FunctionImageOfSetIsSet (F Q : U → Prop) :
    IsFunction F → Q ∈ V → (RelationClassImage F Q) ∈ V
```

3.5 Ordinal Numbers

3.5.1 Universe

First we add some math notation for class difference.

```
def ClassDifference (P Q : U → Prop) :=
    (fun a : U ↦ ¬ Q a ∧ P a)

infix:80 (priority := high) "\\" => ClassDifference
```

Since the backslash is an escape character in Lean strings, the string literal `"\\"` denotes a single visible backslash. This allows us to write `P\Q`. We need the difference of two classes to later proof the induction principle and also in the proof of the following theorem. It states, that a class that contains all its subclasses as elements is the universe.

```
theorem ClassIsUniverseIfSubclassesAreElements (Q : U → Prop):
    (∀ e : U, (SetToClass e ⊆ Q → e ∈ Q)) → Q ≃ V
```

3.5.2 Intersection

We also add notation for intersection of classes and the intersection of one class.

```
def BinaryIntersectionClass (P Q : U → Prop) :=
    (fun c : U ↦ P c ∧ Q c)

infix:55 (priority := high) "∩" => BinaryIntersectionClass
```

```
def IntersectionClass (P : U → Prop) :=
    (fun c : U ↦ ∀ b : U, b∈P → c∈b)

prefix:65 (priority := high) "⋂" => IntersectionClass
```

Which allows us to write `P∩Q` and `⋂P`

We can now show, that there are no circles of classes. In the lecture we have

$$\text{ZF} \vdash \neg\exists v_1 \dots \exists v_n (v_1 \in v_2 \wedge (v_2 \in v_3 \wedge \dots \wedge v_n \in v_1) \dots)$$

We again show this only for $n = 1, 2$. For $n = 1$ we quickly get:

```
theorem NotExist1SetCircle (P : U → Prop) :
    ¬ P ∈ P
```

And for $n = 2$, we first show some helper theorems:

If we have two sets (of class type) with `Q ∈ P`, then we also have corresponding sets with `q ∈ p`.

```
theorem SetInSetForClassInSetClass (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ P) →
        ∃ q : U, ∃ p : U, (((q ≃ Q) ∧ (p ≃ P)) ∧ q ∈ p)
```

The assumption `P ∈ V` gives us that `P` is a set, and the property that every element of a class is a set, gives us that `Q` is a set.

If we have two sets that are equivalent to the same class, those sets are equal.

```
theorem TransOverClassSetEq (P : U → Prop) (q r : U) :
    ((r ≃ P) ∧ (q ≃ P)) → r = q
```

```
theorem NotExist2SetCircle (P Q : U → Prop) :
    ¬ (P ∈ Q ∧ Q ∈ P)
```

3.5.3 Successor & Zero

The ordinals will be defined inductively, so we need a successor function.

Note: To not clash with predefined symbols, we will not use symbols such as $0, 1$ and $+$ but use `"zero"` and `"plus_one"` instead.

In the lecture we write

$$X + 1 \text{ for } X \cup \{X\}$$

where X denotes a class. It can be formalized in the same way.

```
def SuccOf (P : U → Prop) := (fun b : U ↦ b ∈ (P ∪ {P}))

notation:65 (priority := high) x "plus_one" => SuccOf x
```

Warning: The definition of the successor uses the singleton class, so this again works only properly for sets. The successor of a class `P` will be equivalent to the class itself, since `{P} = ∅`. We can prove:

```
theorem SuccOfProperClass (P : U → Prop) :
    ¬ P ∈ V → P ≃ P plus_one
```

In the lecture we write

$$0 \text{ for } \emptyset$$

We just add another notation for `Emptyset`. Of course we can define more number names by applying the successor function to `zero`.

```
notation "zero" => Emptyset
notation "one" => zero plus_one
notation "two" => one plus_one
```

```
theorem ZeroPlusOne : zero plus_one = {zero}
```

```
theorem OnePlusOne : one plus_one = {zero, {zero}}
```

We show that a set and its successor are not equivalent. To do this, we introduce two new helper theorems:

The successor of a set is also a set.

```
theorem SuccOfSetIsSet (P : U → Prop) :
    P ∈ V → P plus_one ∈ V
```

A set is an element of its successor.

```
theorem SetIsElementOfSucc (P : U → Prop) :
    P ∈ V → P ∈ P plus_one
```

```
theorem NotSetEqSucc (P : U → Prop) :
    P ∈ V → ¬ P ≃ P plus_one
```

If the successors of two sets are equivalent, then the sets are also equivalent.

```
theorem SetsEqIfSuccsEq (P Q : U → Prop) :
    (P ∈ V ∧ Q ∈ V) → (P plus_one ≃ Q plus_one → P ≃ Q)
```

3.5.4 Ordinals

We represent ordinal numbers as transitive sets whose elements are themselves transitive. Let A denote a class and v_i, v_j denote two variables, that do not occur in A. In the lecture we say A is **transitive** (or in short notation: Trans(A)), if

$$\text{ZF} \vdash \forall v_i \in A \, \forall v_j \, (v_j \in v_i \to v_j \in A)$$

We formalize it as given and add notation, which allows us to write `Trans P`.

```
def isTransitive (P : U → Prop) :=
    (∀ a : U, P a → ∀ b : U, (b ∈ a → P b))

notation "Trans" => isTransitive
```

Now we can define the ordinal numbers. In the lecture we write

$$\text{Ord} \text{ for } \{v_i \ : \ \text{Trans}(v_i) \wedge \forall v_j (v_j \in v_i \to \text{Trans}(v_j)\}$$

and Ord is called the **class of ordinal numbers**.

```
def OrdinalNumberClass : U → Prop :=
    (fun a ↦ IsTransitive (SetToClass a) ∧
        (∀ b : U, (b ∈ a → IsTransitive (SetToClass b))))

notation "Ord" => OrdinalNumberClass
```

Since `isTransitive` is defined on classes, we cast the sets `a` and `b` to classes. We also add notation that allows us to write `Ord a` for a set `a` or `P ∈ Ord` for a class `P`.

We can prove that 0 is an ordinal number

```
theorem ZeroInOrd : zero ∈ Ord
```

and that the successor of an ordinal number, is also an ordinal number. To do this we add some helper theorems:

Every ordinal number is transitive.

```
theorem ClassTransIfInOrd (P : U → Prop) :
    P ∈ Ord → isTransitive P
```

If a class is transitive, its successor is also transitive.

```
theorem SuccTransIfClassTrans (P : U → Prop) :
    isTransitive P → isTransitive (SuccOf P)
```

If a set `p` is equivalent to a class `P`, then its corresponding class, derived by the set to class cast `SetToCLass p` is also equivalent to `P`.

```
theorem SetClassSetToClassEquiv (P : U → Prop) (p : U) :
    p ≃ P → (SetToClass p) ≃ P
```

Every ordinal number is a set.

```
theorem OrdNumIsSet (P : U → Prop) :
    P∈Ord → P∈V
```

```
theorem SuccOfOrdIsInOrd (P : U → Prop) :
    P ∈ Ord → (SuccOf P) ∈ Ord
```

The class of ordinal numbers `Ord` is transitive.

```
theorem OrdIsTrans : isTransitive Ord
```

The class of ordinal numbers is a proper class. We also introduce a new helper theorem:

If a class `Q` is equivalent to a transitive class `P`, then `Q` is transitive.

```
theorem EquivOfTransIsTrans (P Q : U → Prop) :
    P ≃ Q → isTransitive P → isTransitive Q
```

```
theorem OrdIsProperClass : ProperClass Ord
```

3.5.5 Linear Order

We can prove that the natural order of the ordinal numbers is transitive, anti reflexive and total.

```
theorem OrdHasTrans (a b c : U) :
    (a∈Ord ∧ (b∈Ord ∧ c∈Ord)) → (a∈b ∧ b∈c) → a∈c
```

```
theorem OrdHasAntiRefl (a : U) :
    a∈Ord → ¬a∈a
```

```
theorem OrdHasTotal (a b : U) :
    (a∈Ord ∧ b∈Ord) → a∈b ∨ a=b ∨ b∈a
```

In the lecture we write

$$< \text{ for the class } \in \cap (\text{Ord} \times \text{Ord}).$$

So $a < b$ means that $a \in b$ and $(a, b) \in \text{Ord} \times \text{Ord}$. We call $<$ the **natural linear order of ordinal numbers**.

```
def InLinearOrderOfOrd (P Q : U → Prop) : Prop :=
    (P∈Ord ∧ Q∈Ord ∧ P∈Q)

infix:50 (priority := high) "<" => InLinearOrderOfOrd
```

To check if two classes `P : U → Prop` and `Q : U → Prop` are in linear order of Ord, we first check, that indeed, `P` and `Q` are ordinals, and then, if `P` is an element of `Q`. This works,

since the ordinals have the property that every ordinal number contains all other previous ordinals. With the added notation we can write `P < Q`.

We can prove, that every ordinal number has lower order than its successor.

```
theorem SuccIsGreaterElement (P : U → Prop) :
    P ∈ Ord → (P < (SuccOf P))
```

And that the successor of a ordinal number is the next bigger ordinal with respect to the natural order. We introduce a new helper theorem:

If two classes are equivalent to the same set, those classes are equivalent.

```
theorem TransOverSetClassEquiv (P Q : U → Prop) (q : U) :
    ((q ≃ Q) ∧ (q ≃ P)) → P ≃ Q
```

```
theorem SuccIsNextGreaterElement (P Q : U → Prop) :
    P ∈ Ord ∧ Q ∈ Ord → ((Q < (SuccOf P)) → ((Q ≃ P) ∨ Q < P))
```

3.5.6 Succ & Lim

We now define the successor ordinals and limit ordinals. Let v_i, v_j, v_k be three variables, that do not occur in Ord. In the lecture we write

$$\text{Succ for } \{v_i \ : \ (v_i \in \text{Ord} \ \wedge \exists v_j \in \text{Ord} \ \ v_i = v_j + 1)\}$$

We call Succ the **class of successor ordinal numbers**. It is the class that contains all ordinal numbers that we can get by applying the successor function to some other ordinal number.

```
def Succ : U → Prop :=
    (fun a : U ↦ a ∈ Ord ∧
        (∃ P : U → Prop, P ∈ Ord ∧ a ≃ (P plus_one)))
```

In the lecture we write

$$\text{Lim for } \{v_i \ : \ (v_i \in \text{Ord} \ \wedge (\neg v_i = 0 \wedge v_i \notin \text{Succ}))\}$$

and call Lim the **class of limit ordinal numbers**. It is the class that contains all ordinal numbers that are not 0 nor a successor ordinal.

```
def Lim : U → Prop :=
    (fun a : U ↦ a ∈ Ord ∧ (¬ a ≃ zero ∧ ¬ (Succ a)))
```

3.5.7 Natural Numbers

Finally we can define the natural numbers. In the lecture we write

$$\mathbb{N} \text{ for } \{v_i \ : \ (v_i \in \text{Ord} \ \wedge \forall v_j (v_j \in v_i + 1 \to v_j \notin \text{Lim}))\}$$

We call $\mathbb{N}$ the **set of natural numbers**, although we still have to show, that $\mathbb{N}$ is indeed a set.

```
def NaturalNumbers : U → Prop :=
    (fun a : U ↦ a ∈ Ord ∧ (∀ Q : U → Prop,
        (Q ∈ ((SetToClass a) plus_one) → ¬ (Q ∈ Lim))))

notation "ℕ" => NaturalNumbers
```

To show that 0 is a natural number, we introduce some helper theorems:

A class equivalent to an ordinal number is transitive.

```
theorem EquivOfOrdNumIsTrans (P Q : U → Prop) :
    P ≃ Q → P ∈ Ord → isTransitive Q
```

If a set `p` is equivalent to an ordinal number `P`, then `p` is also an ordinal number.

```
theorem EquivOfOrdNumIsOrdNum (P : U → Prop) (p : U) :
    P ∈ Ord ∧ p≃P → Ord p
```

Two sets `a` and `b` are equal, if `a` is equivalent to the class of `b`.

```
theorem SetsEqIfSetSetToClassEquiv (a b : U) :
    a ≃ (SetToClass b) → a=b
```

```
theorem ZeroInℕ : zero ∈ ℕ
```

We also prove: the natural numbers are closed under the `plus_one` operation.

```
theorem ℕIsClosedUnderPlusOne (P : U → Prop) :
    P ∈ ℕ → (P plus_one) ∈ ℕ
```

To prove the induction principle, we add the following helper theorems:

If we have two classes with `Q ⊆ P`, which are not equivalent, then there exists an element in `P` that is no element of `Q`.

```
theorem ExistsElementInSetDifferenceWithSubset (P Q : U → Prop) :
    (Q ⊆ P ∧ ¬ Q ≃ P) → ∃ a : U, (P\Q) a
```

If a class P is transitive and we have $S \in R \in P$, then we have $S \in P$.

```
theorem TransMemCC (P R S : U → Prop) :
    isTransitive P → S ∈ R → R ∈ P → S ∈ P
```

If we have two equivalent classes `P` and `R`, then every element of `R` is also an element of `P`.

```
theorem MemOfClassThenMemOfSomeEquiv (P R S : U → Prop) :
    (S ∈ R ∧ R ≃ P) → S ∈ P
```

If we have two ordinal numbers with `Q ∈ P`, the same relation holds for their successors.

```
theorem MemOfOrdRemainsUnderPlusOne (P Q : U → Prop) :
    P ∈ Ord ∧ Q ∈ Ord ∧ Q ∈ P → Q plus_one ∈ P plus_one
```

The natural numbers are transitive.

```
theorem ℕIsTrans : isTransitive ℕ
```

```
theorem InductionPrinciple (P : U → Prop) :
    (P ⊆ ℕ ∧ (zero ∈ P ∧ ∀ Q : U → Prop,
       Q ∈ P → (Q plus_one) ∈ P)) → P ≃ ℕ
```

We can prove the existence of an infinite set (with class type).

```
theorem ExistsInfiniteSet :
    ∃ P : U → Prop, (P ∈ V ∧ zero ∈ P ∧
        ∀ Q : U → Prop, Q ∈ P → (Q plus_one) ∈ P)
```

Now we can show that the natural numbers are indeed a set:

```
theorem ℕIsSet : ℕ ∈ V := by ...
```

In the lecture, the proof is:

Corollary. $\mathrm{ZF} \vdash \mathbb{N} \in V$. *Informal,* $\mathbb{N}$ *is a set.*

Proof. According to (Inf), there exists a set X, such that $0 \in X$ and $Y + 1 \in X$ for all $Y \in X$ holds. Define Z as $X \cap \mathbb{N}$. According to (Aus), Z is a set. By definition of X and Lemma:

Lemma. $\mathrm{ZF} \vdash 0 \in \mathbb{N} \wedge \forall v_\ell \in \mathbb{N}\ v_\ell + 1 \in \mathbb{N}$. *Informal,* $\mathbb{N}$ *contains* 0 *and is closed under* $+1$.

Proof. It is clear, that $0 \in \mathbb{N}$. Now let $X \in \mathbb{N}$. Then every $Y \in X + 1$ is not a limit ordinal number. Since $X + 1$ is obviously not a limit ordinal number and

$$(X + 1) + 1 = (X + 1) \cup \{X + 1\} = X \cup \{X, X + 1\},$$

every Element of $(X + 1) + 1$ is also not a limit ordinal number. Therefore is $X + 1 \in \mathbb{N}$. □

we have, $0 \in Z$ and $Y + 1 \in Z$ for all $Y \in Z$. According to the theorem:

Theorem (Induction Principle)**.** *Let* A *be a classterm. Then*

$$\mathrm{ZF} \vdash ((A \subseteq \mathbb{N} \wedge (0 \in A \wedge \forall v_\ell \in A\ v_\ell + 1 \in A)) \to A = \mathbb{N})$$

Proof. Suppose, that $A \subseteq \mathbb{N}$, $0 \in A$ and $X + 1 \in A$ holds for all $X \in A$, but $A \neq \mathbb{N}$. According to (Fund), there exists $X \in \mathbb{N} \setminus A$, such that $Y \in A$ holds for all $Y \in X$. Now we will show, that X is a limit ordinal number. Since $0 \in A$, can X not be the emptyset. Assume, X would be a successor ordinal number. Then there would be an $Y \in \mathrm{Ord}$, with $X = Y + 1$. Since $Y \in X$, we know $Y \in A$ due to the minimality of X. But then it also applies, that $Y + 1 \in A$ and therefore $X \in A$. This contradicts $X \in \mathbb{N} \setminus A$. □

we have, $Z = \mathbb{N}$. Therefore, $\mathbb{N}$ is a set. □

The formalized proof follows the lecture argument. By the axiom of infinity, there is a set X containing 0 and closed under the successor function. We define Z as $X \cap \mathbb{N}$. By specification, Z is a set. Since both X and $\mathbb{N}$ contain 0 and are closed under the successor function, so is Z. The induction principle then gives $Z \simeq \mathbb{N}$. Since Z is represented by a

set, it follows that $\mathbb{N}$ is a set.

We walk through the formalized proof step by step. We start by using some axioms.

```
have ⟨X, ⟨⟨x, hx⟩, h0x, h1x⟩⟩ := ExistsInfiniteSet
let Z := X ∩ ℕ
have ⟨z, hz⟩ : Z ∈ V := by
  have ⟨z, h01⟩ := ax_spec ℕ x
  use z
  constructor
  . tauto
  . intro w
    unfold Z
    constructor
    . intro h02
      have h03 := (h01 w).1 h02
      constructor
      . apply (hx.2 w).1
        exact h03.1
      . exact h03.2
    . intro h04
      apply (h01 w).2
      constructor
      . apply (hx.2 w).2
        exact h04.1
      . exact h04.2
...
```

In order to get a set with a class type `P : U → Prop` from `ax_inf`, instead of `p : U`, we use the helper function

```
theorem ExistsInfiniteSet :
    ∃ P : U → Prop, (P ∈ V ∧ zero ∈ P ∧
        ∀ Q : U → Prop, Q ∈ P → (Q plus_one) ∈ P)
```

which itself uses `ax_inf` and `ax_ext`. Where `X : U → Prop` represents the infinite set X as a class. By `⟨x, hx⟩` we split `X ∈ V` into `x : U`, which is an equivalent set to `X`, and `hx : V x ∧ ∀ (c : U), c∈x ↔ X c`. The two properties, given by the axiom of infinity, are captured by `h0x : zero∈X` and `h1x : ∀ (Q : U → Prop), Q∈X → Q plus_one∈X`. The next step is to define Z as $X \cap \mathbb{N}$. To do this we write `let Z := X ∩ ℕ`. What remains is to use `ax_spec` to show that `Z` is a set, `Z ∈ V`. We use a `have` tactic to do this sub-proof and write `have ⟨z, hz⟩ : Z ∈ V := by ...` to immediately unpack the

statement and receive a corresponding set `z : U` of `Z : U → Prop`. To prove `Z ∈ V`, we need to give a set, that is equivalent to `Z`, we use the axiom of specialization that takes a class `(P : U → Prop)` and returns `∀ a : U, ∃ b : U, ∀ c : U, c∈b ↔ c∈a ∧ P c`. We give `ℕ` as well as `x` for `a` and can unpack the `∃` quantifier to receive a set that contains exactly all sets that are elements of `ℕ` and `x`, hence the intersection of those.

Next, the proof stated: By definition of X and the Lemma ... we have $0 \in Z$ and $Y + 1 \in Z$ for all $Y \in Z$.

```
...
have h0n := ZeroInℕ
have h0z : zero ∈ Z := by
  obtain ⟨e, h01⟩ := h0x
  obtain ⟨e', h02⟩ := h0n
  have h03 : e = e' :=
        TransOverClassSetEq zero e' e ⟨h01.2, h02.2⟩
  use e
  constructor
  . constructor
    . exact h01.1
    . rewrite [h03]
      exact h02.1
  . exact h01.2
...
```

To show that `zero ∈ Z`, we use that `zero ∈ ℕ`, which we proved in the helper theorem `ZeroInℕ` (unlike the comment in the proof of the lemma, that this is clear, it took quite some proof lines, even though we quickly see, that by definition of `ℕ`, this is true). Since `Z` is an intersection, we also need `zero ∈ X`, and since `zero` is a class, we obtain two initially independent sets, called `e` and `e'`, which are the emptysets, so we also show that those are the same in `h03`.

To formalize the statement that for all $Y \in Z$, $Y + 1 \in Z$ holds, we write
`∀ (R : U → Prop), R∈Z → R plus_one∈Z`.

```
...
have h1n := ℕIsClosedUnderPlusOne
have h1z : ∀ (R : U → Prop), R∈Z → R plus_one∈Z := by
  intro S ⟨s, ⟨⟨h01, h02⟩, hs⟩⟩
  have h03 : S ∈ X := by
    use s
  have ⟨s1, h04⟩ := h1x S h03
  use s1
  constructor
  . have h05: S ∈ ℕ := by
      use s
    have ⟨s1', h06⟩ := h1n S h05
    have h07 : s1 = s1' :=
        TransOverClassSetEq (S plus_one) s1' s1 ⟨h04.2, h06.2⟩
    constructor
    . exact h04.1
    . rewrite [h07]
      exact h06.1
  . exact h04.2
...
```

We write `intro S ⟨s, ⟨⟨h01, h02⟩, hs⟩` , this adds a class `S : U → Prop` and destructs `S∈Z` by adding a corresponding set `s : U` of `S` (see `hs : ∀ (c : U), c∈s ↔ S c`), and splits the intersection information of `Z s` into `h01 : X s` and `h02 : ℕ s`. It remains to show that `S plus_one∈Z`, by `h1n` we know that ℕ is closed under `plus_one` and due to the properties of `ax_inf X` is also closed under `plus_one`, this property can be found at `h1x : ∀ (Q : U → Prop), Q∈X → Q plus_one∈X`.

The last step is to apply the induction principle

```
theorem InductionPrinciple (P : U → Prop) :
    (P ⊆ ℕ ∧ (zero ∈ P ∧ ∀ Q : U → Prop,
        Q ∈ P → (Q plus_one) ∈ P)) → P ≃ ℕ
```

to show that `Z ≃ ℕ` and to use the corresponding set `z : U` to show, that ℕ is a set.

```
...
have h3 : Z ⊆ ℕ := by
  intro y h01
  exact h01.2
have h4 : Z ≃ ℕ := InductionPrinciple Z ⟨h3, h0z, h1z⟩
use z
constructor
. tauto
. intro w
  rewrite [hz.2 w]
  exact h4 w
```

This completes the proof.

Note: Tracing the dependencies of this proof shows that it uses most of the theorems from the ordinal-number section of the Lean formalization. It also shows that most of the formalized ZF-axioms are used, although many occur only indirectly through intermediate results. The only unused axioms are `ax_power` and `ax_repl`. The full dependency overview is included in the appendix.

Conclusion

The goal of this thesis was to develop a framework to enable incorporating Lean proofs when teaching set theory. This was done by formalizing part of the set theory chapter of the lecture "Einführung in die Mathematische Logik" in Lean and documenting the formalization in a way that makes it usable alongside the lecture. The formalization follows the lecture closely and develops the set-theoretic framework up to the theorem that the natural numbers are a set.

Since Lean is based on dependent type theory, the first part of the thesis introduced the relevant type-theoretic background. Starting from the simply typed lambda calculus, then moving on to motivate why dependent types are expressive enough to represent mathematical statements and proofs. The Curry-Howard correspondence plays an important role. It explains a central idea used in Lean: propositions are types, and proofs are terms. This means that proofs can be checked by the system.
The second part introduced basic Lean concepts and proof techniques. It explains how to formalize axioms, definitions, theorems and notation. It also gives guidelines and provides overviews to help readers write their own proofs or extend the file.
The third part of the thesis documented the formalization itself. Sets were represented as elements of an abstract type, while classes were represented as predicates on sets. On this basis, the ZF-axioms used in the lecture were introduced, and usual set-theoretic constructions such as the empty set, singleton and pair classes, ordered pairs, unions, powersets, relations, and functions were formalized. Afterwards, ordinal numbers, successor ordinals, limit ordinals, and finally the class of natural numbers were formalized. Along these definitions, corresponding theorems from the lecture were formalized as well as some interesting statements.

The formalization shows that the informal arguments from the lecture can be translated into Lean in a way that remains close to the original presentation. At the same time many details that are usually left implicit in written mathematical proofs, were made explicit through the exactness of Lean expressions.

Future work could be to continue the formalization. Remaining topics for the ordinal numbers are "transfinite induction and recursion on ordinal numbers" and "transfinite recursion and ordinal arithmetic". The other remaining chapters in the set theory part of the lecture are "cardinal numbers" and "the axiom of choice". It would also be interesting to develop additional examples and exercises, so that the Lean formalization can be used directly in teaching. In this way, Lean could help students interactively construct and check formal proofs, making the proof-theoretic foundations of set theory more accessible.

Overall, the thesis demonstrates that Lean can serve as a bridge between the formal proof system presented in the lecture and an interactive environment in which these proofs can be written, checked, and explored.

German Summary

Diese Arbeit umfasst eine Formalisierung in Lean auf Grundlage des mengentheoretischen Kapitels der Vorlesung "Einführung in die Mathematische Logik" (Stand 2025) von Philipp Hieronymi. Die Formalisierung umfasst die Inhalte des Kapitels "Mengentheorie" bis einschließlich des Satzes, dass die Klasse der natürlichen Zahlen eine Menge ist.

Ausgangspunkt ist die Beobachtung, dass die in der Vorlesung verwendeten formalen Beweise als Ableitungen über dem Sequenzkalkül zwar präzise, aufgrund ihrer Natur, aber für komplexere Aussagen schwer nachvollziehbar und aufwendig zu erstellen sind. Der interaktive Beweisassistent Lean bietet die Möglichkeit, solche formalen Beweise schrittweise zu entwickeln und maschinell überprüfen zu lassen. Ziel dieser Arbeit war es daher, eine Lean-Umgebung zu erstellen, die den mengentheoretischen Teil der Vorlesung möglichst genau widerspiegelt.

Da Lean auf abhängiger Typentheorie basiert, beginnt die Arbeit mit einer Einführung in typentheoretische Grundlagen. Zuerst wird der einfach typisierte Lambda-Kalkül vorgestellt, um ein Verständnis für Terme und Typen zu entwickeln. Danach werden zentrale Ideen der abhängigen Typentheorie erklärt, insbesondere die Curry-Howard-Korrespondenz, nach der Propositionen als Typen und Beweise als Terme verstanden werden.
Im zweiten Teil werden grundlegende Lean-Konzepte eingeführt, welche für das Verständnis der Formalisierung benötigt werden. Dazu gehören Axiome, Definitionen, Notationsdeklarationen, Theoreme und typische Beweistaktiken. Zusätzlich werden Konventionen und Strategien beschrieben, die beim Lesen und Schreiben von Lean-Beweisen hilfreich sind.
Der dritte Teil dokumentiert die eigentliche Formalisierung. Mengen werden durch Elemente eines abstrakten Typs repräsentiert, Klassen durch Prädikate über diesem Typ. Auf dieser Grundlage werden die verwendeten ZF-Axiome formalisiert. Danach werden grundlegende Konstruktionen wie die leere Menge, Singleton- und Paar-Klassen, geordnete Paare, Vereinigungen, Potenzklassen, Relationen und Funktionen entwickelt. Anschließend werden Ordinalzahlen, Nachfolger- und Limesordinalzahlen sowie die Klasse der natürlichen Zahlen formalisiert. Der abschließende Beweis, dass die natürlichen Zahlen eine Menge bilden, wird detailliert dargestellt und mit dem entsprechenden Beweis aus der Vorlesung verglichen.

Die Formalisierung zeigt, dass die informellen und formalen Argumente der Vorlesung in Lean nachvollzogen werden können, ohne die Nähe zur ursprünglichen mathematischen Darstellung zu verlieren. Gleichzeitig erzwingt Lean eine hohe Präzision: Viele Zwischenschritte, die in gewöhnlichen mathematischen Texten implizit bleiben, müssen explizit angegeben werden. Dadurch kann die Formalisierung als Ergänzung zur Vorlesung als Grundlage für eine spätere Einbindung interaktiver Beweise in die Lehre dienen.

Appendix

Backtracing all prerequisites of the theorem that $\mathbb{N}$ is a set, leads to the following dependencies:

1. ExistsInfiniteSet
   (a) ax.inf
   (b) TransOverClassSetEq – ax.ext
2. ax.spec
3. ZeroInN
   (a) ax.emptyset
   (b) ZeroInOrd – ax.emptyset
   (c) EquivOfOrdNumIsOrdNum
      - EquivOfOrdNumIsTrans .
   (d) TransOverClassSetEq – ax.ext
   (e) SetsEqIfSetSetToClassEquiv – ax.ext
4. $\mathbb{N}$IsClosedUnderPlusOne
   (a) SuccOfOrdIsInOrd
      i. OrdNumIsSet .
      ii. ClassTransIfInOrd .
      iii. SuccTransIfClassTrans .
      iv. SuccOfSetIsSet
         - SingletonClassIsSetForSets
           – PairClassIsSetForSets – ax.pair – ax.ext
           – SingletonIsPair .
         - BinaryUnionClassIsSetForSets
           – UnionClassIsSetForSets – ax.union
           – PairClassIsSetForSets – ax.pair – ax.ext
           – EquivOfSetIsSet .
           – SimEquivOfClasses .
           – BinaryUnionAsUnion .
      v. SetClassSetToClassEquiv .
      vi. TransOverClassSetEq – ax.ext
   (b) EquivOfOrdNumIsOrdNum
      - EquivOfOrdNumIsTrans .
   (c) SetClassSetToClassEquiv .
   (d) SimEquivOfClasses .
   (e) TransOverSetClassEquiv .
   (f) EquivOfOrdNumIsOrdNum
      - EquivOfOrdNumIsTrans .

(g) TransOverClassSetEq – ax.ext

5. TransOverClassSetEq – ax.ext
6. InductionPrinciple
   (a) ExistsElementInSetDifferenceWithSubset .
   (b) ax.reg
   (c) TransOverClassSetEq – ax.ext
   (d) $\mathbb{N}$IsTrans
      i. OrdIsTrans .
      ii. SuccOfOrdIsInOrd (see above)
      iii. MemOfOrdRemainsUnderPlusOne
         A. SuccIsGreaterElement
            - SuccOfOrdIsInOrd (see above)
            - OrdNumIsSet .
            - SetIsElementOfSucc .
         B. SuccOfOrdIsInOrd (see above)
         C. ClassTransIfInOrd .
         D. TransMemCC .
         E. OrdHasTotal – ax.ext – ax.reg
         F. SuccIsNextGreaterElement
            - TransOverSetClassEquiv .
         G. MemOfClassThenMemOfSomeEquiv .
         H. NotExist1SetCircle – ax.reg
         I. NotExist2SetCircle
            - ElementofClassIsSet .
            - ax.pair
            - PairClassExists – ax.pair – ax.ext
            - ax.reg
            - SetInSetForClassInSetClass .
            - TransOverClassSetEq – ax.ext
            - SetInSetForClassInSetClass .
         J. MemOfClassThenMemOfSomeEquiv .
         K. SuccIsNextGreaterElement
            - TransOverSetClassEquiv .
   (e) TransOverClassSetEq – ax.ext
   (f) TransOverClassSetEq – ax.ext

References

[1] Jeremy Avigad and Patrick Massot. *Mathematics in Lean*. 2025.

[2] Leonardo De Moura et al. "The Lean theorem prover (system description)". In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388.

[3] Herman Geuvers. *Proving with Computer Assistance*. Lecture notes, Eindhoven University of Technology. 2023.

[4] Philipp Hieronymi. *Einführung in die Mathematische Logik*. Lecture notes, University of Bonn. 2025.