Bachelor thesis
Computing Science

Radboud University

# Intuitionism in Lean

*Author:*
Rick Koenders
s4576519

*First supervisor/assessor:*
Dr. Freek Wiedijk
freek@cs.ru.nl

*Second assessor:*
Prof. dr. Herman Geuvers
herman@cs.ru.nl

August 21, 2020

**Abstract**

Lean, as an interactive theorem prover, is relatively new compared to older theorem provers like Coq and Isabelle. The Lean mathematical library (mathlib) contains formalizations of various mathematics definitions and theorems. Most of these work the context of classical logic, assuming that the law of the excluded middle (LEM) holds true.

Intuitionism provides a logical foundation for mathematics in which LEM is actually false. As such, formalizing parts of intuitionism is a good test to see how Lean behaves in circumstances that are seldom explored. We base our formalization around the lecture notes for the course Intuitionistic Mathematics by W. Veldman at the Radboud University. Meanwhile, we will note on difficulties we encounter with Lean and the process of formalizing intuitionism.

We formalized notions such as sequences of natural numbers, rational segments and the intuitionistic real numbers, what it means for a statement to be reckless, Brouwer's continuity principle and the statement of the fan theorem.

All in all, Lean proved to be a good tool for formalizing intuitionism. Though there is a bit of a learning curve, we did not encounter any big problems. We did have some difficulty with coercions between natural numbers, finite numbers and the rational numbers. Since Lean is based on constructive logic, it did not have any problems with intuitionistic principles which imply that some statements in classical logic are false.

# Contents


**1 Introduction** **3**
1.1 Overview . . . . . 4

**2 Preliminaries** **5**
2.1 Constructive Mathematics . . . . . 5
2.1.1 Propositional Logic . . . . . 5
2.1.2 First Order Logic . . . . . 6
2.2 Lean . . . . . 8
2.2.1 Proving a statement . . . . . 8
2.2.2 Definitions . . . . . 11

**3 Formalization** **13**
3.1 Natural Sequences . . . . . 13
3.1.1 Finite Sequences . . . . . 17
3.1.2 `len_seq (nat.succ n)` $\simeq \mathbb{N}$ . . . . . 19
3.2 Reckless Statements . . . . . 22
3.2.1 Categorizing the Principles of Omniscience . . . . . 23
3.2.1.1 LEM $\to$ WLEM and LPO $\to$ WLPO . . . . 23
3.2.1.2 PO $\to$ LPO and WPO $\to$ WLPO . . . . . 25
3.2.1.3 WLPO $\to$ LLPO . . . . . 26
3.2.2 Proving Statements are Reckless . . . . . 30
3.2.2.1 $\forall$ `P : Prop,` $\neg\neg$`P` $\to$ `P` is reckless . . . . . 31
3.2.2.2 $\forall a, b \in \mathcal{N}[a \leq b \to a < b \vee a = b]$ is reckless . 32
3.3 Brouwer's Continuity Principle . . . . . 34
3.3.1 Applications of Brouwer's Continuity Principle . . . . 35
3.4 Rational Segments and Real Sequences . . . . . 42
3.4.1 Rational Segments . . . . . 42
3.4.2 Real Sequences . . . . . 45
3.4.3 Trichotomy on $\mathcal{R}$ is reckless . . . . . 47
3.4.4 Defining addition on $\mathcal{R}$ . . . . . 51
3.5 Spreads and Fans . . . . . 56

**4 Related Work** **58**

**5 Conclusions** **59**
5.1 Acknowledgments . . . . . 60

**A Appendix** **63**
A.1 Index of all theorems . . . . . 63
A.1.1 nat_seq.lean . . . . . 63
A.1.2 reckless.lean . . . . . 64
A.1.3 bcp.lean . . . . . 66
A.1.4 segment.lean . . . . . 66
A.1.5 real.lean . . . . . 67
A.1.6 fin_seq.lean . . . . . 69
A.1.7 coding_sequences.lean . . . . . 70
A.1.8 fan.lean . . . . . 70
A.2 Overview of common Lean tactics . . . . . 72
A.2.1 apply . . . . . 72
A.2.2 cases . . . . . 72
A.2.3 exact . . . . . 73
A.2.4 exfalso . . . . . 73
A.2.5 have . . . . . 73
A.2.6 induction . . . . . 73
A.2.7 intro . . . . . 74
A.2.8 left/right . . . . . 74
A.2.9 refl . . . . . 74
A.2.10 revert . . . . . 75
A.2.11 rw . . . . . 75
A.2.12 set . . . . . 76
A.2.13 split . . . . . 76
A.2.14 suffices . . . . . 76
A.2.15 use . . . . . 76

# Chapter 1

# Introduction

In this thesis, we will formalize parts of the lecture notes for the course Intuitionistic Mathematics by Wim Veldman.[16]

We will do this in the proof assistant Lean.[9] For more information on Lean, see section 2.2.

There are several reasons why formalizing intuitionism in Lean is interesting, including:

- To test if Lean can handle it. Since most formalizations in Lean are done with classical logic (with the Lean classical namespace), it would be interesting to see how Lean will behave in a system of logic that contradicts many classical statements. Ultimately, since Lean itself is based on constructive logic, with the classical statements built on top of it, we do not foresee this to be a problem.

- To test if Lean is user friendly. Lean is relatively new compared to other theorem provers, which have been around for a couple of decades. Even so, most mathematicians still use paper and pencil to do their proofs, mostly because it is easier. Improving the user friendliness of Lean is therefore important to getting more mathematicians to formalize new statements in a proof assistant. During our formalization, we will comment on any problems we face and where these problems come from.

- To have a computer check if the underlying math is correct. This is the underlying motivation for many formalization projects. Mathematicians can make mistakes, so if we can catch those mistakes with computers, we would have fewer mistakes. This is mainly important once more complicated math is getting formalized, at which point we can rely on proof assistants that are already tested in other circumstances. We also do not think this will be a problem in our case.

The Xena Project aims to formalize the entire undergraduate mathematics curriculum at the Imperial College London. Neither this project, nor

the Lean community's math library mathlib has formalized intuitionistic mathematics in Lean. The main contribution of this thesis will, as such, be the formalization in Lean of intuitionistic concepts such as recklessness and Brouwer's continuity principle.

## 1.1 Overview

In the preliminaries (section 2), we will go into further detail what constructive mathematics and intuitionism are and what the differences are with classical mathematics. We show how proving a statement in Lean works, and how we can define concepts in Lean.

Section 3 contains most of the actual work; the formalization. We will formalize concepts in the following order: Natural sequences $\mathcal{N}$, what is means for a statement to be reckless, Brouwer's continuity principle, rational segments and the intuitionistic real numbers, and finally the definitions of spreads, fans and the fan theorem. We explain all definitions, along with some important or interesting theorems for which we explain the proofs.

After this we discuss related work, and in the conclusions we discuss what can be learned from our formalization.

The appendix contains an index to all theorems that we have formalized in Lean. Proofs have been omitted there, because they take up too much space, but we have included a link to our github page which has the full Lean code. After that is an overview of common Lean tactics that we use in our proofs. These are in alphabetical order to make referencing the list easy.

# Chapter 2

# Preliminaries

## 2.1 Constructive Mathematics

Intuitionism is a subset of constructive mathematics. In constructive mathematics, the proof is the most important thing about a statement; the meaning of the logical symbols follows from how statements including those symbols can be proven.

### 2.1.1 Propositional Logic

For propositional logic, the Brouwer-Heyting-Kolmogorov interpretation gives meaning to the logical operators.[14] For the propositions P and Q, the proofs of the following are given by:

- $P \wedge Q$ is a proof of $P$ and a proof of $Q$
- $P \vee Q$ is a proof of $P$ or a proof of $Q$
- $P \to Q$ is a function that transforms a proof of $P$ into a proof of $Q$
- $\perp$ is the proposition with no proof
- $\neg P$ is an abbreviation for $P \to \perp$

This interpretation has some interesting consequences. For example, in classical mathematics the following statement holds:

**Example:** In classical logic: $P \vee Q \leftrightarrow \neg(\neg P \wedge \neg Q)$
**Proof:** First we prove $P \vee Q \to \neg(\neg P \wedge \neg Q)$. Assume $P \vee Q$ and assume $\neg P \wedge \neg Q$ are true. We now have to prove $\perp$. From $\neg P \wedge \neg Q$ we know that both $\neg P$ and $\neg Q$ are true. For $P \vee Q$ there are two cases: If $P$ is true, then we obtain a contradiction from $P$ and $\neg P$. If $Q$ is true, then we obtain a contradiction from $Q$ and $\neg Q$.
We now have to prove $\neg(\neg P \wedge \neg Q) \to P \vee Q$. Assume $\neg(\neg P \wedge \neg Q)$. If $P$,

then we immediately conclude $P \vee Q$. If $\neg P$, then we know that $\neg Q$ would lead to a contradiction with $\neg(\neg P \wedge \neg Q)$. So we conclude that $Q$ must be true, and therefore $P \vee Q$ must be true. $\square$

Constructively, the first part of this proof is completely fine. However, there are two classical "mistakes". The first is the case distinction on $P$ and $\neg P$. To make this argument, we would first have to prove $P \vee \neg P$. Classically this is easy: Use the Law of Excluded Middle!

**Defintion:** LEM: For all propositions $P$ we have that $P \vee \neg P$.

Constructively, this "law" does not hold. Think of any big unsolved problem in mathematics today. If you claim you can prove $P \vee \neg P$, you claim that you can prove $P$ or that you can prove $\neg P$. But since the problem you thought of is unsolved, clearly those proofs do not exist. LEM is sometimes also called to Principle of Omniscience (PO), because claiming that you know LEM is true is like claiming you are omniscient: You know for every possible statement whether it is true or not.

The second mistake is concluding that $Q$ must be true. The assumptions $\neg P$ and $\neg(\neg P \wedge \neg Q)$ do indeed mean that $\neg Q$ would lead to a contradiction, and therefore $\neg Q$ is not true. So: $\neg\neg Q$. The mistake in classical reasoning is that for all propositions $Q$ it holds true that $\neg\neg Q \to Q$, also known as double negation elimination. This is equivalent to LEM.

**Proposition:** LEM $\leftrightarrow$ ($\neg\neg Q \to Q$ holds for all propositions $Q$)
**Proof:** See section 2.2.1: Lean.

All differences we have seen so far can all be explained by the fact that classical mathematicians use two different definitions of the logical operators. Sometimes $P \vee Q$ means that they have a proof of either one, and sometimes they mean that $\neg P$ and $\neg Q$ cannot both be true. In constructive mathematics, only the first interpretation is seen as correct. The same principle applies to $P \to Q$, which is classically equivalent to $\neg P \vee Q$. However, constructively the second interpretation is stronger than the first.

### 2.1.2 First Order Logic

First order logic introduces the quantifiers $\forall$ and $\exists$. We can see what the quantifiers mean by looking at how you can prove propositions with them:

- A proof of $\forall x[Px]$ is a function that takes as input $x$ and produces as output a proof of $Px$
- A proof of $\exists x[Px]$ is an element $x$, together with a proof of $Px$

This interpretation of $\exists$ explains the word "constructive" in constructive mathematics: To prove a statement with $\exists$, you have to construct the

element you claim exists. This interpretation is stronger than the classical interpretation. We illustrate this with an example:

**Example:** In classical logic: $\neg\forall x[Px] \to \exists x[\neg Px]$
**Proof:** Assume $\neg\forall x[Px]$. We show the result with a proof by contradiction. Assume $\neg\exists x[\neg Px]$. Let $x$ be an element. If $\neg Px$, then $\exists x[\neg Px]$, which is a contradiction with our assumption $\neg\exists x[\neg Px]$. So $Px$ must be true. So $\forall x[Px]$. This is a contradiction with our assumption $\neg\forall x[Px]$. Therefore, $\exists x[\neg Px]$. $\square$

This classical proof uses proof by contradiction, which assumes that we can perform double negation elimination. The above proof fails constructively. However, the contrapositive is still true.

**Proposition:** $\exists x[\neg Px] \to \neg\forall x[Px]$
**Proof:** Assume $\exists x[\neg Px]$ and assume $\forall x[Px]$. From $\exists x[Px]$ we get an element $y$ such that $\neg Py$. However, from $\forall x[Px]$ we know that $Py$ is true. This is a contradiction. Therefore, $\neg\forall x[Px]$. $\square$

One may wonder how those two proofs are different. They look like they both use proof by contradiction. This is not the case! Proof by contradiction is: $\forall P \in Prop[(\neg P \to \bot) \to P]$. This is different from the second proof, which uses: $\forall P \in Prop[(P \to \bot) \to \neg P]$, which is **not** proof by contradiction, but merely the definition of $\neg$.

## 2.2 Lean

Lean is an open-source theorem prover developed by Microsoft Research.[9] It was released in 2013. We used Lean 3, but Lean 4 is already in development. Lean 3 has an active community of people formalizing parts of mathematics. This is partly focused on formalizing the most important theorems of mathematics, but there have also been some efforts to formalize more modern and complicated pieces of math. One example of this is the formalization of the definition of perfectoid spaces.

We will give a quick overview of proving statements in Lean below. If you would like to learn more, a good place to start is the Natural Number Game by K. Buzzard.[5] After that, quickly reading through Theorem Proving in Lean[1] or The Hitchhiker's Guide to Logical Verification[2] should be enough to start formalizing statements in Lean. Another good resource is the Lean Zulip chat, where all things Lean are discussed. The subsection for new members is ideal to ask questions about anything you do not understand. Or you can search for threads where other people have asked the same question and have already gotten an answer.[8]

Lean supports Unicode. When using the IDE Visual Studio Code, you can type Unicode characters with predefined shortcuts. For example, to type $\forall$, you need to type \all. To type $\vee$, type \or and to type $\leftrightarrow$, type \iff. Later on we will also use Unicode symbols to refer to types like $\mathbb{N}$, $\mathbb{Q}$, $\mathcal{N}$, $\mathbb{S}$, and $\mathcal{R}$. These can be typed with \nat, \rat, \McN, \bbS, and \McR respectively. All Unicode characters you will see in Lean code here is also present in our actual Lean files. Lean code resembles actual pencil and paper math more closely because of this and is therefore, in our opinion, easier to read.

### 2.2.1 Proving a statement

In the previous section we claimed that the law of excluded middle and double negation elimination are equivalent. This statement makes for a good introductory example into Lean. The full proof is as follows:

```
example : (∀ P : Prop, P ∨ ¬P) ↔ (∀ Q : Prop, ¬¬Q → Q) :=
begin
  split,
  {
    intros h Q nnq,
    cases h Q with hq nq,
    {
      exact hq,
    },
    {
      exfalso,
      exact nnq nq,
```

```
        }
    },
    {
        intros h P,
        apply h (P ∨ ¬P),
        exact not_not_or P,
    }
end
```

We will look at this first proof line by line to show what is going on.

```
example : (∀ P : Prop, P ∨ ¬P) ↔ (∀ Q : Prop, ¬¬Q → Q) :=
```

This line is the statement that we are going to prove. The first word shows that this is an example. Other keywords that can be used here are `lemma`, `theorem`, `def`, `axiom`, `structure`, and others. An example is just a lemma or theorem without a name, which would normally be located right behind the opening keyword. For example, this lemma has the name `not_not_or`:

```
lemma not_not_or : ∀ P : Prop, ¬¬(P ∨ ¬P) :=
```

Behind the name we could insert some assumptions, but those were not necessary for this statement. After the assumptions is a colon, which tells Lean that everything after it is the actual statement that will need to be proven. The end of the statement is indicated by a colon and an equality symbol (:=), after which the proof will start.

Lean has two different proof styles: terms and tactics. These can be compared to the different modes of Haskell. Term style proofs are the default way of doing things, combining different terms to create a full proof. Tactic style proofs are like do-notation in Haskell. It opens a monad that lets you write proofs line by line. Tactic style proofs are indicated by `begin` at the start of the proof and `end` at the end of it. Tactic style proofs are more similar to how you would write a proof with pencil and paper. If you are working in an interactive IDE, this proof style also allows you to look at the state of Lean line-by-line. This makes tactic mode more user friendly and easier for beginners to get started with. We will often omit `begin` and `end` while discussing proofs, because we work predominantly with tactic style proofs. Mentioning `begin` and `end` every time would quickly get tiring.

The next line in the proof is:

```
split,
```

The tactic `split` can be used when the goal is a conjunction. The result will be two separate goals: the left side and the right side. So if the goal was $P \wedge Q$, after using split there will be two goals: the first one is to prove $P$ and the second one is to prove $Q$. We use it here on $\leftrightarrow$ to turn it into $\rightarrow$ and $\leftarrow$. We will now first have to prove the statement:

```
(∀ P : Prop, P ∨ ¬P) → (∀ Q : Prop, ¬¬Q → Q)
```

To prove an implication, we should assume that the left side is true and show that the right side must be true as well. To assume we would normally use the tactic `intro`. In this case, we can shorten our proof by introducing multiple assumptions on one line. Our next steps would be to assume that `Q` is a proposition and that `¬¬ Q` is true. We can shorten those three steps into one with the tactic `intros`:

```
intros h Q nnq,
```

After this line, we have three new assumptions, and a different goal. In an interactive proof, the assumptions are at the top. At the bottom the current goal is shown after the symbol ⊢.

```
h : ∀ (P : Prop), P ∨ ¬P,
Q : Prop,
nnq : ¬¬Q
⊢ Q
```

We now want to use the law of excluded middle, which we have called `h`, in our proof to show that $Q \vee \neg Q$ is true. We could do this with the following tactic:

```
have h1 : Q ∨ ¬Q := h Q,
```

The tactic `have` introduces a new assumption. In this case we introduced the hypothesis with the name `h1`, with type `Q ∨ ¬Q` and with proof `h Q`. Stating the type is optional, since Lean can figure that out from the proof. Sometimes it can still be useful to explicitly state the type though. For example, when the proof is long, you can proof `have` statements using `by`, which introduces a new goal. Technically the following will still work:

```
have h1, by {
   exact h Q,
}
```

but this quickly becomes illegible for humans when proofs get more complicated. Note that in the real proof we skipped this step by combining it with our next tactic.
After obtaining $Q \vee \neg Q$ we want to split this into two cases: Either $Q$ is true or $\neg Q$ is true. We can do this using the `cases` tactic.

```
cases h Q with hq nq,
```

After this tactic, we will have two goals. One in which the assumption `hq : Q` holds and the goal is `⊢ Q`, and one in which `nq : ¬Q` holds and the goal is `⊢ Q`. Note that the goals are the same but the assumptions are different. The first case is easy: The assumption `hq : Q` is a proof of our goal. So we can close it using the tactic `exact`:

```
exact hq,
```

In the second case, the assumption `nq` : ¬`Q` actually contradicts the desired goal. But it also contradicts our assumption `nnq` : ¬¬`Q`. Recall that $\neg P$ is an abbreviation for $P \to \bot$. In Lean, we can prove any goal by proving `false` instead. This rule is commonly known as "Ex falso sequitur quod libet", which is reflected in the tactic name `exfalso`. We can now complete the proof of (∀ `P` : `Prop`, `P` ∨ ¬`P`) → ∀ `Q` : `Prop`, ¬¬`Q` → `Q` with:

```
exfalso,
exact nnq nq,
```

We still have to show that the other implication holds as well: (∀ (`Q` : `Prop`), ¬¬`Q` → `Q`) → ∀ (`P` : `Prop`), `P` ∨ ¬`P`. We do this in three lines. First we assume what the necessary: `h` : (∀ (`Q` : `Prop`), ¬¬`Q` → `Q`) is true and $P$ is a proposition. Note that `h` (`P` ∨ ¬ `P`) is now of the type ¬¬(`P` ∨ ¬`P`) → (`P` ∨ ¬`P`), and that our goal is ⊢ `P` ∨ ¬`P`. We can use the tactic `apply` to combine these two facts. After using `apply`, our new goal will be ⊢ ¬¬(`P` ∨ ¬`P`). We now use the `not_not_or` mentioned previously to complete the proof.

```
intros h P,
apply h (P ∨ ¬P),
exact not_not_or P,
```

You may have noticed that the original proof also uses a lot of curly brackets. These serve two purposes. Firstly, they tell the interactive version of Lean that you want to focus on a single goal, even when there are multiple available. This is why we use curly brackets after the tactics `split` and `cases`. Secondly, they look nicer. It shows a clear delineation between different parts of the proof, which makes it easier to read.

### 2.2.2 Definitions

A definition in Lean has similar syntax to theorems. The difference is that they are interpreted differently. Take the following example:

```
def nat_seq := ℕ → ℕ

notation `𝒩` := nat_seq
```

This states that the defintion of `nat_seq` is ℕ → ℕ. Note that the type is implicit, but Lean figures out that the type is `Type`. The notation command allows us to write $\mathcal{N}$ instead of `nat_seq`. We can also declare the type explicitly:

```
def lt (a b : 𝒩) : Prop :=
  ∃ n : ℕ, (∀ i : ℕ, i < n → a i = b i) ∧ a n < b n
```

`lt` takes as input two elements of type $\mathcal{N}$ and then defines a proposition, which has type `Prop`. This means that `lt` has the type $\mathcal{N} \to \mathcal{N} \to$ `Prop`. The

semantics of this definition will be explained in section 3.1.
A different option for definitions in Lean is the `structure` keyword.

```
structure fin_seq := mk :: (len : ℕ) (seq : fin len → ℕ)
```

A structure has fields that provide the actual definitions. In the example, a `fin_seq` has a length `len : ℕ` and a function from the set `fin len`, the set of all natural numbers less than `len`, to ℕ. The type `fin` is itself defined as a `structure` in the Lean core library. The definition is:

```
structure fin (n : nat) := (val : nat) (is_lt : val < n)
```

In the definition of `fin_seq`, we also included `mk`. This is the make function, which allows us to make an instance of the structure in-line. The two following definitions use the two different styles of defining instances of structures. Both define a sequence of length 1 with a predefined output.

```
def singleton (n : ℕ) : fin_seq := {
  len := 1,
  seq := λ i, n
}

def singleton' (n : ℕ) : fin_seq := fin_seq.mk 1 (λ i, n)
```

These definitions may looks different, but they are equal. The statement `singleton n = singleton' n` can be proven with the tactic `refl`, which only uses that `=` is reflexive ($\forall x[x = x]$).

We also did not provide the type of `i` in both of these definitions. Lean can infer the type, which is this case is `fin 1`, since the type of `singleton.seq` has to be `fin 1` $\to \mathbb{N}$.

# Chapter 3

# Formalization

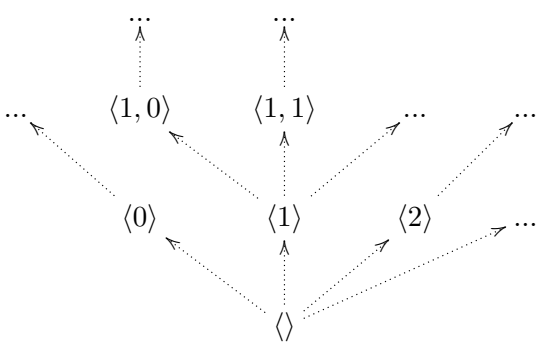
## 3.1 Natural Sequences

A natural sequence is a list of natural numbers, where each entry is also indexed by the natural numbers. We could see a sequence as a function that takes as input an index, and gives as output the element at that index. In Lean, we formalize this as follows:

```
def nat_seq := ℕ → ℕ

notation `𝒩` := nat_seq
```

The Baire Space $\mathcal{N}$ is the set of all infinite natural sequences. Brouwer thought of $\mathcal{N}$ a different way. To think of an element of $\mathcal{N}$, he said, you first need to think of the empty sequence $\langle\rangle$ and then make a choice for the first element: $\langle n_0 \rangle$. Then you choose the second element $\langle n_0, n_1 \rangle$ and then continue choosing elements forever to make an infinite list. The full set $\mathcal{N}$ can then be thought of as the tree of all possible decisions.




Using this tree as the definition in Lean would make other statements more complicated than necessary, so we opted to use the way a set theorist would usually define sequences instead.

Comparisons on $\mathcal{N}$ are given by the lexicographical order. To determine how two sequences are related, start by comparing the two elements at index 0. If they are equal, go on to index 1. If one is less than the other, we say that the entire sequence is less than the entire other sequence. If they elements are equal all the way through, then the two sequences must be equal. In Lean, this notion of equality can be expressed as:

```
def eq (a b : 𝒩) : Prop := ∀ n : ℕ, a n = b n

infix '=' := eq
```

Because Leans supports functional extensionality, this definition of `eq` is equivalent to the definition given by the usual =-sign in Lean:

```
lemma eq_iff {a b : 𝒩} : a = b ↔ a =' b := function.funext_iff
```

The corresponding notion for one sequence being less than another sequence can be expressed as:

```
def lt (a b : 𝒩) : Prop :=
  ∃ n : ℕ, (∀ i : ℕ, i < n → a i = b i) ∧ a n < b n

infix '<' := lt
```

The part `(∀ i : ℕ, i < n → a i = b i)` in the definition of `lt` says that the start of the sequence is the same, but they diverge at the index `n` according to `a n < b n` (note that we can choose `n` to be 0 if the first elements are different).

The first non-trivial statement about these definitions is that $<$ is transitive:

```
@[trans] theorem lt_trans (a b c : 𝒩) : a < b → b < c → a < c
```

The tag `@[trans]` allows us to use the tactic `transitivity` in the future whenever we need to prove a statement of the form $a < c$ for $a, c \in \mathcal{N}$. Using `transitivity b` will transform the goal from $\vdash$ `a < c` into two different goals: $\vdash$ `a < b` and $\vdash$ `b < a`.

This proof in Lean is interesting when compared to a normal pencil a paper proof. We will go over both proofs first and then note some differences.

The proofs start out the same: Assume that `a < b` and `b < c` are true. In Lean: `intros hab hbc`. Then: Find n and m such that:

$$\forall i < n[a(i) = b(i)] \wedge a(n) < b(n) \text{ and } \forall i < m[b(i) = c(i)] \wedge b(m) < c(m)$$

This is writing out the definition of `<` on $\mathcal{N}$ and using the fact that $\exists x[P(x)]$ means that we can really obtain the element $x$ such that $P(x)$ is satisfied. We do this in Lean with `cases hab with n hn` and `cases hbc with m hm`. Because they are $\wedge$ statements, we can split `hn` and `hm` into two parts using `cases` as well. The state of the proof is now:

```
a b c : 𝒩,
n m : ℕ,
p₁ : ∀ (i : ℕ), i < n → a i = b i,
p₂ : a n < b n,
q₁ : ∀ (i : ℕ), i < m → b i = c i,
q₂ : b m < c m
⊢ a < c
```

We need to prove `a < c`, which by definition requires us to provide a natural number $k$ such that: $\forall i < k[a(i) = c(i)] \wedge a(k) < c(k)$. The pencil and paper proof now makes a case distinction between $n < m$, $n = m$ and $m < n$, and uses the smaller of the two as the number $k$. This case distinction can be done because comparison on $\mathbb{N}$ is decidable. We postpone making this distinction to shorten the proof in Lean a little bit. Instead, we tell Lean to use the number $min(n, m)$: `use min n m`. The resulting goal will be an and-statement, so we use `split` to obtain two separate goals.

We prove the first goal, `∀ i : ℕ, i < min n m → a i = c i`, by first assuming `i < min n m` is true for a given `i : ℕ`. Then we use the lemma `lt_min_iff` from the Lean math library (mathlib[13]) which states:
`i < min n m ↔ i < m ∧ i < n`
The first goal now follows from the assumptions $\mathtt{p}_1$ and $\mathtt{q}_1$:

```
rw p₁ i hi.elim_left,
exact q₁ i hi.elim_right,
```

We can now use the case distinction on $<$ to reach the second conclusion. To do this in Lean, we use `cases nat.lt_trichotomy n m with nltm h`. `nat.lt_trichotomy` states that for all natural numbers $n$ and $m$ we have $n < m \vee n = m \vee m < n$, so we now have two new goals, one with the assumptions `nltm : n < m` and one with the assumption `h : n = m ∨ m < n`. In the first case we use the lemma `min_left : n ≤ m → min n m = n` (use `nat.le_of_lt` to turn `n < m` intro `n ≤ m`). In the second case use `min_self : min m m = m` and in the thrid case use `min_right`. Completing the proof is now easy in all three cases, requiring only rewrites using the assumptions $\mathtt{p}_1$ and $\mathtt{q}_1$.

As a comparison, the complete proof provided by Wim Veldman in his lecture notes (slightly altered to avoid reasoning about finite sequences): Assume $a < b$ and $b < c$. Find $n, m$ such that $\forall i < n[a(i) = b(i)] \wedge a(n) < b(n)$ and $\forall i < m[b(i) = c(i) \wedge b(m) < c(m)$. If $m = n$, then $\forall i < m[a(m) = c(m)] \wedge a(m) < b(m) < c(m)$, and, therefore, $a < c$. If $n < m$, then $\forall i < n[a(i) = c(i)$ and $a(n) < b(n) = c(n)$ and, therefore, $a < c$. If $m < n$, then $\forall i < m[a(i) = c(i)$ and $a(m) = b(m) < c(m)$, and, therefore, $a < c$.[16]

This proof illustrates a common theme: In pencil and paper proofs, steps in reasoning that are deemed too easy are often omitted. Another difference is that pencil and paper proofs show the "state" of the proof ("What has been proven so far?") rather than the steps in reasoning to advance the proof

("What do we need to use to obtain the result?"). All in all, this means that Lean proofs in tactic mode are usually longer than pencil and paper proofs.

This proof is also exemplary of many of the proofs about $\mathcal{N}$ in Lean: use the definitions, some reasoning about $<$ on the natural numbers, and maybe `lt_trichotomy`. The only exceptions are statements that easily follow from other statements, statements about a finite portion of sequences, and the theorem `uncountable`, which we will discuss below.

For the first exception, a good example is `eq_stable`. It builds on the definition of `apart` and the theorem `eq_iff_not_apart`:

```
def apart (a b : 𝒩) := ∃ n, a n ≠ b n
infix `#` := apart
theorem eq_iff_not_apart (a b : 𝒩) : a =' b ↔ ¬ a # b
```

The last theorem can be proven in a similar way to `lt_trans`. Using these three things, the proof of `eq_stable` is as follows: First rewrite the hypothesis ¬¬ a =' b to ¬¬¬ a # b and the goal a =' b to ¬ a # b, and use the fact that ¬¬¬P → ¬P for all propositions P:

```
theorem eq_stable (a b : 𝒩) : ¬¬ a =' b → a =' b :=
begin
  rw eq_iff_not_apart,
  exact not_of_not_not_not,
end
```

For the last theorem in this section we'd like to discuss `uncountable`, which proves that $\mathcal{N}$ is uncountable using Cantor's diagonal argument.

```
theorem uncountable (f : ℕ → 𝒩) : ∃ a : 𝒩, ∀ n : ℕ, a # (f n) :=
begin
  use λ n : ℕ, (f n n) + 1,
  intro n,
  use n,
  exact nat.succ_ne_self (f n n),
end
```

The assumption `(f : ℕ → 𝒩)` can be interpreted a list of elements of $\mathcal{N}$. Our goal is then to create an element not on the list to show that the function is not surjective (onto). The first line of the proof is the element that we create: `λ n : ℕ, (f n n) + 1`. For element $n$ on the list we look at its value at $n$ and add one to make sure our new element is different. We then have to prove `∀ n : ℕ, a # (f n)`. The tactic `intro n` gives us a natural number to reason about: what element of the list are we looking at? According to the definition of #, we then have to provide a natural number where `a` and `f n` are not equal. This is, by construction of `a`, at $n$. We know that
`a n = (f n n) + 1`, so we have to prove `f n n ≠ (f n n) + 1`, which is precisely the lemma `nat.succ_ne_self (f n n)`, completing the proof.

### 3.1.1 Finite Sequences

A finite sequence is a sequence with a finite number of elements. To better integrate with $\mathcal{N}$, we will only consider finite sequences of natural numbers here. In theory, most definitions here could be generalized to obtain finite sequences of any type. Our definition for finite sequences will be:

```
structure fin_seq := mk :: (len : ℕ) (seq : fin len → ℕ)
```

So a finite sequence is a structure with a length, and a function from `fin len` to ℕ. This function, `seq`, can be thought of as the actual sequence: `(seq 0, seq 1, seq 2, ...)`.

Now we will define some useful constructions for finite sequences. Firstly, we can take an infinite sequence and make it finite:

```
def finitize (a : 𝒩) (n : ℕ) : fin_seq := {
  len := n,
  seq := λ i, a i.val
}
```

We have to define the sequence with a lambda expression, because `i` has type `fin n`, while `a` has type ℕ → ℕ. We use `i.val` here, because `val` transforms an element of type `fin n` to the type ℕ.

Doing this, we obtain a finite sequence of length `n`:

```
lemma finitize_len (a : 𝒩) (n : ℕ) : (finitize a n).len = n := rfl
```

We can also do this for finite sequences, shortening them. The only necessary condition for this is that the new length we have in mind is less than the original length of the sequence:

```
def shorten (a : fin_seq) (m : ℕ) (h : m ≤ a.len) : fin_seq := {
  len := m,
  seq := λ i, a.seq (fin.cast_le h i)
}
```

We can also go into the opposite direction, extending a finite sequence with an infinite sequence to make a new infinite sequence, or extending it with another finite sequence to make a new finite sequence:

```
def extend_inf (a : fin_seq) (b : 𝒩) : 𝒩 :=
  λ i, if h : i < a.len
    then a.seq (fin.cast_le h i)
    else b (i − a.len)

def extend (a b : fin_seq) : fin_seq := {
  len := a.len + b.len,
  seq := λ i, if h : i.val < a.len
          then a.seq (fin.cast_le h i)
```

```
            else b.seq (fin.sub_nat a.len
              (fin.cast (add_comm a.len b.len) i)
              begin -- need to prove: a.len ≤ (fin.cast _ i).val
                rw not_lt at h,
                transitivity i.val,
                exact h,
                refl,
              end),
}
```

The only tricky part here is the fact that we want to use the higher indices to access elements of `b`, so to subtract the length of `a` from `i`, we need to prove that `a.len` is less than `i`.

We define the empty sequence $\langle\rangle$ as follows:

```
def empty_seq : fin_seq := {
  len := 0,
  seq := λ i, 0
}
```

The value 0 in the definition of `empty_seq.seq` does not actually matter, since `fin 0` has no inhabitants. We can prove that any sequence that also has `len = 0` will have the same outputs as `empty_seq`:

```
lemma empty_seq_eq {a : fin_seq} (ha : a.len = 0) :
  ∀ i, empty_seq.seq i = a.seq (fin.cast _ i) :=
begin
  intro i, -- assume i ∈ fin 0
  exfalso, -- this is impossible
  have hi := i.is_lt, -- i < 0
  rw lt_iff_not_ge' at hi, -- ¬0 ≤ i
  apply hi, -- goal: 0 ≤ i
  exact zero_le i.val,
end
```

One problem with the definition of `fin_seq` is that equality is impossible if the lengths are not equal. If we have two natural numbers, $n$ and $m$, and we know $n < m$ or $n > m$, then the fact that a sequence with type `fin n` $\to \mathbb{N}$ is not the same as a sequence of type `fin m` $\to \mathbb{N}$ makes sense. The bigger problem is that this is also the case when $n = m$, since Lean sees `fin n` and `fin m` as two different types, even when `n = m`. Equality on $\mathcal{N}$ (both the one we defined and the built in one) behaves much more nicely, which is why we can prove things like:

```
lemma extend_inf_eq {a : fin_seq} {b₁ b₂ : 𝒩} (h : b₁ =' b₂) :
  extend_inf a b₁ =' extend_inf a b₂
lemma empty_extend_eq_self (a : 𝒩) :
  extend_inf empty_seq a =' a
```

If we want to talk about a sequence with a fixed length, then we can do that with the following definition:

```
def len_seq (n : ℕ) : Type := fin n → ℕ
```

Two elements of `len_seq 0` are always equal. The proof of this uses functional extensionality, followed by the same argument we used for `fin_seq` with `len = 0`.

```
lemma len_seq_0_unique (x y : len_seq 0) : x = y
```

### 3.1.2 `len_seq (nat.succ n)` $\simeq \mathbb{N}$

For any $n \in \mathbb{N}$, we have that `len_seq (nat.succ n)` has the same cardinality as $\mathbb{N}$. They are equivalent, in the sense that there exists a function from `len_seq (nat.succ n)` to $\mathbb{N}$ with an function from $\mathbb{N}$ to `len_seq (nat.succ n)` that is both a left and right inverse. In classical mathematics, you would show this with a bijection between `len_seq (nat.succ n)` and $\mathbb{N}$, but getting an instance of $\simeq$ from a bijection is noncomputable. From mathlib:

```
noncomputable def of_bijective {α β} {f : α → β}
  (hf : bijective f) : α ≃ β := ...
```

So to prove our claim `len_seq (nat.succ n)` $\simeq \mathbb{N}$, we will have to provide the two functions and the proof that they are left and right inverses explicitly. Here is the definition from mathlib:

```
structure equiv (α : Sort*) (β : Sort*) :=
(to_fun    : α → β)
(inv_fun   : β → α)
(left_inv  : left_inverse inv_fun to_fun)
(right_inv : right_inverse inv_fun to_fun)

infix ` ≃ `:25 := equiv
```

The reason why we have `nat.succ n` in the statement of this theorem is that the proof does not work for `len_seq 0`, since it only has one element. It is equivalent to the type `punit`, which also only has one element: `punit.star`. The following lemma captures this (proofs of `left_inv` and `right_inv` omitted):

```
lemma len_seq_0_equiv_punit : len_seq 0 ≃ punit := {
  to_fun := λ a, punit.star,
  inv_fun := λ x, λ i, 0,
  left_inv := ...,
  right_inv := ...,
}
```

We will prove the claim about `len_seq (nat.succ n)` by induction. The statement we will prove is:

```
theorem len_seq_equiv_nat (n : ℕ) : len_seq (nat.succ n) ≃ ℕ :=
```

We begin the proof with:

```
induction n with d hd,
```

For the base case, we need to prove:

```
lemma len_seq_1_equiv_nat : len_seq 1 ≃ ℕ
```

We know that `len_seq 1` is the sequence with only one element. A sequence with only one natural number could be seen as just being that natural number, so the statement `len_seq 1 ≃ ℕ` should be no surprise. For the function `to_fun : len_seq 1 → ℕ`, we can use the only entry in the sequence. The inverse is then the sequence that has that natural number as its output:

```
to_fun := λ a, a (fin.mk 0 nat.succ_pos'),
inv_fun := λ n, λ i, n,
```

To prove `left_inv`, we have to prove for an arbitrary `a : fin 1` that applying `to_fun` and then applying `inv_fun` gives us back the original `a`. We do this with functional extensionality: Two functions are equal if their outputs are equal for every input.

```
intro a,
rw function.funext_iff,
intro i,
```

Our goal (simplified) is now ⊢ `a ⟨0, _⟩ = a i`. We prove this by proving that ⟨0, _⟩ = i. This is true because `fin 1` only has one element: 0.

```
have hi : i = fin.mk 0 nat.succ_pos', by {
   rw fin.eq_iff_veq,
   have h := nat.le_of_lt_succ i.is_lt,
   rw nat.le_zero_iff at h,
   exact h,
},
```

In the first line we use that two elements of `fin` are equal if their values are equal. Then we use that for an element `i` of `fin 1` we have that `i < 1`, which we can change to `i ≤ 0`. But a natural number is always bigger than or equal to zero, so we know `i = 0`, which proves what we wanted. The main goal is then closed with `rw hi`.

Proving that `inv_fun` is a right inverse for `to_fun` can be done in two lines:

```
right_inv := begin
  intro n,
  refl,
end
```

This works because the left and right side are definitionally equal to each other. Using `dsimp`, which only simplifies by writing out definitions, yields `⊢ n = n`.

Now that we know that the base case holds, we need to prove the inductive step. We do this with transitivity of ≃. This gives us two new goals:

```
⊢ len_seq (nat.succ (nat.succ d)) ≃ len_seq (nat.succ d)
⊢ len_seq (nat.succ d) ≃ ℕ
```

The second goal is exactly our induction hypothesis `hd`. For the first one, we use the following lemma:

```
lemma len_seq_succ_equiv_len_seq (n : ℕ) :
  len_seq (nat.succ (nat.succ n)) ≃ len_seq (nat.succ n)
```

For this we can use the pairing functions `nat.mkpair` and `nat.unpair`. For `to_fun`, we do nothing to the first n elements of the sequence, but for the last two entries, we use `nat.mkpair` to make them into a single element. For the inverse, we use `nat.unpair` to get two elements from the last element. The full definitions are (with proofs that some numbers are really elements of the correct `fin` replaced by _):

```
to_fun := λ a, λ i, if i.val = n
  then nat.mkpair
    (a ⟨n, _⟩)
    (a ⟨nat.succ n, _⟩
  else a ⟨i.val, _⟩,
inv_fun := λ a, λ i, if h₁ : i.val < n
  then a ⟨i.val, _⟩
  else if h₂ : i.val = n
    then (nat.unpair (a ⟨i.val, _⟩)).1
    else (nat.unpair (a ⟨i.val − 1, _⟩)).2,
```

We will omit the proof that these functions really are each others inverses, because they are mostly making a lot of case distinctions.

## 3.2 Reckless Statements

A reckless statement is a proposition that is true in classical mathematics, but for which a weak counterexample exists in intuitionistic mathematics. This counterexample is usually given in the form of the problem of the existence of 99 nines in the decimal expansion of $\pi$.

**Definition:** $d : \mathrm{N} \to \{0, ..., 9\}$ such that $\pi = 3 + \sum_{n=0}^{\infty} d(n) \cdot 10^{-n-1}$
**Definition:** $k_{99} = \mu N \in \mathbb{N}[\forall i < 99[d(N+i) = 9]]$

$k_{99}$, if it exists, is the smallest natural number such that the decimal expansion of $\pi$ has 99 consecutive nines starting at $k_{99}$. The problem for the intuitionistic mathematician is then: Does $k_{99}$ exists? There is no proof of this fact, and there is also no proof that $k_{99}$ does not exist. So there is no proof of the statement:

$$\exists n \in \mathbb{N}[n = k_{99}] \vee \forall n \in \mathbb{N}[n \neq k_{99}]$$

We can translate this statement to the realm of sequences of natural numbers by defining a sequence which encapsulates this problem.

**Definition:** $\alpha_{99} \in \mathcal{N}$ is defined by

$$\alpha_{99}(n) = \begin{cases} 0 \text{ if } n \neq k_{99} \\ 1 \text{ if } n = k_{99} \end{cases}$$

Translating the previously obtained unproved statement we obtain another unproved statement:

$$\forall n \in \mathbb{N}[a_{99}(n) = 0] \vee \exists n \in \mathbb{N}[a_{99}(n) \neq 0]$$

This statement gives rise to the definition of LPO, the limited principle of omniscience. It is called limited, because it is weaker than the principle of omniscience, the law of excluded middle.

**Definition:** LPO : $\forall a \in \mathcal{N}\ [\forall n \in \mathbb{N}[a(n) = 0] \vee \exists n \in \mathbb{N}[a(n) \neq 0]]$

For our formalization in Lean, we have chosen to use LPO to define the notion of reckless statements. The reason for this is because it is thinkable that the unproved statements about $k_{99}$ and $a_{99}$ can be proven in the future. For example, a computer search of the decimal expansion of $\pi$ could show that $\exists n \in \mathbb{N}[n = k_{99}]$ is true. If this happens, these statements will need to be replaced by more computationally expensive statements that cannot feasibly be proven at the technological level at the time. Replacing them with $k_{9999}$ and $a_{9999}$ could be an option then. However, having to update the definition

of reckless statements in a computer formalization would imply that the formalization is faulty. Therefore, we have defined reckless statements using LPO instead of $k_{99}$. A reckless statement in our definition is then a statement that is classically true, but implies that LPO is true:

```
def PO : Prop := ∀ P : Prop, P ∨ ¬P
def LPO : Prop := ∀ a : 𝒩, (∀ n : ℕ, a n = 0) ∨ (∃ n : ℕ, a n ≠ 0)
def reckless_LPO : Prop → Prop :=
  λ P : Prop, (PO → P) ∧ (P → LPO)
```

This will be our main way to prove statements are reckless. There are, however, some alternative principles of 'omniscience' that are weaker than PO (LEM) or LPO that can also be used to categorize statements. Two of these are negative versions of PO and LPO, which we will call WLEM (weak law of excluded middle) and WLPO (weak limited principle of omniscience). The third is the weakest of the five that we will discuss here, called LLPO, the lesser limited principle of omniscience.

```
def WLEM : Prop := ∀ P : Prop, ¬P ∨ ¬¬P
def WLPO : Prop := ∀ a : 𝒩, (∀ n : ℕ, a n = 0) ∨ (¬∀ n : ℕ, a n = 0)
def LLPO : Prop := ∀ a : 𝒩,
  (∀ k : ℕ, (∀ i : ℕ, i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 0) ∨
  (∀ k : ℕ, (∀ i : ℕ, i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 1)
```

The definition of LLPO says that, given any sequence of natural numbers $a \in \mathcal{N}$, we can decide if the first index where $a(k) \neq 0$ occurs is even or odd, **before** even knowing what that index actually is. Constructively, this turns out the be impossible. Classically, however, this is possible by making clever use of the other, stronger, principles of omniscience.

### 3.2.1 Categorizing the Principles of Omniscience

We have now defined five different principles of omniscience. Some of those imply others. The diagram[1] below shows all implications between them. All the arrows correspond to statements that we have proven in Lean.

$$\begin{tikzcd}
LEM/PO \arrow[r] \arrow[d] & WLEM \arrow[d] & \\
LPO \arrow[r] & WLPO \arrow[r] & LLPO
\end{tikzcd}$$

#### 3.2.1.1 LEM → WLEM and LPO → WLPO

The implications LEM → WLEM and LPO → WLPO are easy. To prove WLEM from PO, assume PO is true and that P is a proposition. Apply PO to P and perform a case distinction. In the case of `P`, prove ¬¬`P` by

[1] Subdiagram of diagram found in [10], section 6.5

showing ¬P leads to a contradiction. In the case that ¬P, proving ¬P ∨ ¬¬P is trivial. The full Lean code follows this outline of the proof exactly:

```
theorem PO_implies_WLEM : PO → WLEM :=
begin
  intros po P,
  cases po P with hp np,
  {-- case: P
    right, -- need to prove: ¬¬P
    intro np,
    exact np hp,
  },
  {-- case: ¬P
    left, -- need to prove: ¬P
    exact np,
  }
end
```

The proof of LPO → WLPO is quite similar. The only part of the proof that would requiring further explanation is the proof of:

$$\exists n \in \mathbb{N}, a(n) \neq 0 \to \forall n \in \mathbb{N}, a(0) = 0 \vee \neg \forall n \in \mathbb{N}, a(n) = 0$$

Starting from the state:

```
ane : ∃ (n : ℕ), a n ≠ 0
⊢ (∀ (n : ℕ), a n = 0) ∨ ¬∀ (n : ℕ), a n = 0
```

We want to prove `¬∀ (n : ℕ), a n = 0`, which we can select with the tactic `right`. Recall that $\neg P$ for a proposition $P$ is shorthand for $P \to \bot$, so we can use `intro aeq` to add `∀(n : ℕ), a n = 0` to the assumptions, resulting in the following tactic state:

```
ane : ∃ (n : ℕ), a n ≠ 0,
aeq : ∀ (n : ℕ), a n = 0
⊢ false
```

The way we obtain a contradiction from here is clear: Find the natural number $n \in \mathbb{N}$ such that $a(n) \neq 0$ using `ane` and obtain a contradiction with `aeq`. In Lean, this looks as follows:

```
cases ane with n hn,
exact hn (aeq n),
```

In the last line we use that `a n ≠ 0` is shorthand for `(a n = 0) → false`, which allows us to transform `aeq n`, a proof that `a n = 0`, into a proof of `false`.

#### 3.2.1.2 PO → LPO and WPO → WLPO

The next statements that we will prove are PO → LPO and WPO → WLPO. These proofs are also similar, as they rely on two crucial steps: choosing the right proposition to apply PO or WPO to, and using the lemma `forall_not_of_not_exists`.

```
lemma forall_not_of_not_exists {a : Sort u} {p : a → Prop} :
  ¬(∃ x, p x) → (∀ x, ¬p x)
```

As the conclusion of this lemma is that we can get a ∀-statement from an ∃-statement, it is often smart to apply PO and WPO to an ∃-statement, because getting an ∃-statement from a ∀-statement is often impossible. This is due to the fact that ∃ is interpreted strongly in constructive mathematics.

For PO → LPO, we will choose the statement `∃ n : ℕ, a n ≠ 0`. We can use PO to obtain:

$$\exists n \in \mathbb{N}[a(n) \neq 0] \vee \neg\exists n \in \mathbb{N}[a(n) \neq 0]$$

The left side of this expression is precisely the right side of LPO, and we can use `forall_not_of_not_exists` to obtain the left side of LPO from the right side of this expression. Here is the full proof with some additional command to make it more legible:

```
theorem PO_implies_LPO : PO → LPO :=
begin
  intro po, -- Assume PO is true
  intro a, -- Let a be a sequence of natural numbers (a ∈ 𝒩)
  cases po (∃ n : ℕ, a n ≠ 0) with h₁ h₂, -- Use PO
  {-- case: ∃ n : ℕ, a n ≠ 0
    right, -- Need to prove: ∃ n : ℕ, a n ≠ 0
    exact h₁,
  },
  {-- case: ¬∃ n : ℕ, a n ≠ 0
    left, -- Need to prove: ∀ n : ℕ, a n = 0
    have h₃ := forall_not_of_not_exists h₂,
    simp at h₃, -- Simplify the result of the previous line
    exact h₃,
  }
end
```

The proof of WLEM → WLPO is similar. We again reason about the statement `∃ n : ℕ, a n ≠ 0`, but we have to use WLEM now. The second case of the proof of PO → LPO is exactly the same as the first case in this proof:

```
(¬∃ n : ℕ, a n ≠ 0) → (∀ n : ℕ, a n = 0)
```

The second case is slightly harder than the first case of PO → LPO, because it requires more reasoning about ¬, though ultimately it proves itself. We will start our analysis of the proof from the tactic state directly after using `right`:

```
a : 𝒩,
nnh : ¬¬∃ (n : ℕ), a n ≠ 0
⊢ ¬∀ (n : ℕ), a n = 0
```

The goal is of the form $\neg P$, so we use the tactic `intro h` to obtain a new assumption and the goal `false`. We can then use `apply nnh` to tactic state into:

```
a : 𝒩,
nnh : ¬¬∃ (n : ℕ), a n ≠ 0,
h : ∀ (n : ℕ), a n = 0
⊢ ¬∃ (n : ℕ), a n ≠ 0
```

We again have a goal of the form $\neg P$, so we use `intro nex` to obtain the hypothesis `∃ (n : ℕ), a n ≠ 0`. We can use the tactic `cases nex with n nhn` to obtain the natural number `n` such that nhn : a n ≠ 0 from the hyptothesis `nex`. This, however, is a contradiction with our assumption `h : ∀(n : N), a n = 0`. So we can close our goal with the tactic `exact nhn (h n)`.

#### 3.2.1.3 WLPO → LLPO

The last implication we will prove is WLPO → LLPO. The proof of this implication is more complex than the previous four, mostly because we need to extract as much information as possible from WLPO. After the opening tactic `intros h a`, we have the tactic state:

```
a : 𝒩,
wlpo : ∀ (a : 𝒩), (∀ (n : ℕ), a n = 0) ∨ ¬∀ (n : ℕ), a n = 0
⊢ (∀ k : ℕ, ∀ i : ℕ) i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 0)
∨ (∀ k : ℕ, (∀ i : ℕ, i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 1)
```

We will now have to apply WLPO to the appropriate element of $\mathcal{N}$. This choice is the most important part of the proof, as everything else follows naturally from what WLPO says about it. It is also the most interesting part of our formalization, since it is the first time we use the tactic `set` and the first time we have to actively reason about the class `decidable`. The

element of $\mathcal{N}$ we have chosen is:

$$d(n) = \begin{cases} 0 \text{ if } (n\%2 = 0) \wedge (\exists i \in \mathbb{N}[i < n \wedge a(i) \neq 0]) \\ a(n) \text{ if } (n\%2 = 0) \wedge \neg(\exists i \in \mathbb{N}[i < n \wedge a(i) \neq 0]) \\ 0 \text{ if } \neg(n\%2 = 0) \end{cases}$$

This definition could be simplified a bit, but for the purpose of our formalization we can apply `split_ifs` more easily if the conditions are more split up. We can use the `set` tactic to define this sequence:

```
set d : 𝒩 := λ n, if n % 2 = 0 then
  if (∃ i : ℕ, i < n ∧ a i ≠ 0) then 0 else a n else 0 with ddef,
```

Lean will now add two new hypothesis to the tactic state. First it recognizes that we want to set `d` to be an element of $\mathcal{N}$. This step also records the definition of `d`, but does not provide an easy way to use that defintion in our proof. The hypothesis `ddef` will be used for that, as it provides an equality between `d` and the definition of `d`:

```
ddef : d = λ (n : ℕ), ite (n % 2 = 0)
  (ite (∃ (i : ℕ), i < n ∧ a i ≠ 0) 0 (a n)) 0
```

An important thing to note about `ite`-constructions in Lean is that the conditions that are used need to be decidable. If they were not, Lean will not accept the defintion because the function is not well defined. To let Lean know that our conditions are decidable, we need to implement the `decidable` class. For many simple statements this is already done. In our case, we do not need to prove `∀ n : ℕ, decidable (n % 2 = 0)`, as this is already implemented. However, we do need to provide the proof for the following statement:

```
instance start_lt_not_zero_decidable (a : 𝒩) (n : ℕ) :
  decidable (∃ i : ℕ, i < n ∧ a i ≠ 0)
```

For proving statements involving `decidable`, it is useful to know how it is defined:

```
class inductive decidable (p : Prop)
| is_false (h : ¬p) : decidable
| is_true  (h : p) : decidable
```

So `decidable` is an inductive type with two options: The statement is either false or it is true (and we know which case we are in). With this definition in mind, we now know that we can use `cases` on statements with `decidable` just like we would on statements with ∨. We can also prove statements with `decidable` by using `apply is_false` and `apply is_true` at the appropriate times.

We will omit the explanation of the full proof of this statement. A summary of it would be: prove the statement using induction on `n`. For `n = 0`, the statement `∃ i : ℕ, i < n ∧ a i ≠ 0` is false, because `i < 0` is impossible for natural numbers. We can use `apply is_false` to change the goal to `¬∃ (i : ℕ), i < 0 ∧ a i ≠ 0`, which can be proven easily.

For the inductive case, we will rely on the fact that we know that the basis is `decidable`, along with the fact that ≠ is `decidable` on the natural numbers. The last statement is already proven in Lean under the name `ne.decidable`. The full proof of the inductive case is then based purely on case distinctions for statements that were proven previously and using `is_false` and `is_true` at appropriately.

Having properly defined what `d` is, we can now apply WLPO to it by using `cases wlpo d with deq nd`. This will add two new cases in which we need to prove that LLPO holds:

```
case or.inl
deq : ∀ (n : ℕ), d n = 0

case or.inr
nd : ¬∀ (n : ℕ), d n = 0
```

We will start with the first case. Because `d` is 0 everywhere, we know that the first instance of `a n ≠ 0` must be at an odd index (or `a` is the zero-sequence). After `right`, `intros k hk`, we arrive at the following tactic state (previously mentioned assumptions omitted):

```
deq : ∀ (n : ℕ), d n = 0,
k : ℕ,
hk : (∀ (i : ℕ), i < k → a i = 0) ∧ a k ≠ 0
⊢ k % 2 = 1
```

We can apply `deq` to `k` to obtain the insight that `d k = 0`. With the following three tactics we can obtain this fact with the defintion of `d` written out fully simplified:

```
have hdk := deq k,
rw ddef at hdk,
simp at hdk,
```

We will also use a fact about natural numbers modulo 2 that will often be useful when reasoning about LLPO: all natural numbers are either 0 or 1 modulo 2. In this case, we use this fact with the theorem `nat.mod_two_ne_zero`, which states (along with some other useful theorems about `n % 2`:

```
theorem mod_two_ne_zero {n : ℕ} : ¬(n % 2 = 0) ↔ n % 2 = 1
```

```
theorem mod_two_ne_one {n : ℕ} : ¬(n % 2 = 1) ↔ n % 2 = 0
lemma mod_two_eq_zero_or_one (n : ℕ) : n % 2 = 0 ∨ n % 2 = 1
```

Using `rw ← nat.mod_two_ne_zero`, our new goal becomes `¬k % 2 = 0`. We can then use `intro hkm` to obtain the hypothesis that `k % 2 = 0` and the goal `false`. At this point, we use `split_ifs at hdk` to make a case distinction on the `if/else`-conditions of `d`. Lean recognizes that we are in the case where `k % 2 = 0` holds, so we now only have two cases based on the different possibilities for `∃ i : ℕ, i < k ∧ a i ≠ 0`. The tactic state for the first case is:

```
k : ℕ,
hk : (∀ (i : ℕ), i < k → a i = 0) ∧ a k ≠ 0,
hkm : k % 2 = 0,
h : ∃ (i : ℕ), i < k ∧ a i ≠ 0,
hdk : 0 = 0
⊢ false
```

The hypothesis `hdk : 0 = 0` is a result of the definition of `d` and the previous hypothesis `d k = 0`. This hypothesis is useless, so we will have to rely on `h`. We can see that `h` is in contradiction with `hk`, so we can close the goal as follows:

```
cases h with i hi,
apply hi.elim_right,
exact hk.elim_left i hi.elim_left,
```

In the second case, we know that `¬∃ i : ℕ, i < k ∧ a i ≠ 0` holds and our hypothesis `d k = 0` changed into `a k = 0`. This is, again, a direct contradiction with `hk`, so we can close the second goal using:

```
exact hk.elim_right hdk,
```

This fully solves the problem for the case where WLPO gave us the information that $\forall n \in \mathbb{N}[d(n) = 0]$ holds. We will now look at the second case: $\neg\forall n \in \mathbb{N}[d(n) = 0]$. The start is similar to the first case. After the following steps, we arrive at the tactic state below:

```
left,
intros k hk, -- need to prove: k % 2 = 0
rw ← nat.mod_two_ne_one,
intro hkm,
apply nd,
intro n,
```

```
nd : ¬∀ (n : ℕ), d n = 0,
k : ℕ,
```

```
hk : (∀ (i : ℕ), i < k → a i = 0) ∧ a k ≠ 0,
hkm : k % 2 = 1,
n : ℕ
⊢ d n = 0
```

At this point, we can use `ddef` for the full definition `d` and use `split_ifs` to obtain three new goals, dependent on the three cases in the if/else-clauses for `d`. Two of these goals are trivial, because they ask us to show `0 = 0` holds, which can be proven with `refl`. So we will focus on the case where we need to prove `a n = 0` by using `n % 2 = 0` and `∀ x : ℕ, x < n → a x = 0`. For this we need to combine what we know about `n` and `k`. Just like in 3.1, we can compare `n` to `k` using `lt_trichotomy`. In the case `n < k`, we can `hk` to immediately prove that `a n = 0` holds:

```
exact hk.elim_left n nlt,
```

In the case that `n = k`, we obtain a contradiction from `hkm: k % 2 = 1` and `h : n % 2 = 0`:

```
exfalso,
rwa [ ← nat.mod_two_ne_one, neq] at h,
exact h hkm,
```

In the case that `k < n`, we obtain a contradiction from
`∀ (x : ℕ), x < n → a x = 0` and `a k ≠ 0`:

```
exfalso,
apply hk.elim_right,
exact h_1 k ngt,
```

This concludes the proof that WLPO → LLPO.

### 3.2.2 Proving Statements are Reckless

Now that we have proven implications between the different principles of omniscience, we can start proving that statements are reckless. Recall that the definition of a reckless statement we are using in this thesis is:

```
def reckless_LPO : Prop → Prop :=
  λ P : Prop, (PO → P) ∧ (P → LPO)
```

So to prove a statement is reckless, we have to show that it is true classically by using PO to prove it, and we have to show that it does not hold intuitionistically, by showing that it implies LPO. We also have an analogous version `reckless_LLPO`, for statements that are weaker than LPO but still imply LLPO.

#### 3.2.2.1 ∀ P : Prop, ¬¬P → P is reckless

The statements that we will prove to be reckless here are actually equivalent to one of the principles of omniscience. We will show one example for the different principles. The first statement we will consider is equivalent to PO:

```
theorem reckless_not_not_implies :
  reckless_LPO (∀ P : Prop, ¬¬P → P)
```

This theorem shows that double negation elimination is not possible for all propositions. This also means that proof by contradiction is not always justified. To prove this statement, we start with the tactic `split`. We will now have to prove that PO implies $\neg\neg P \to P$ for all propositions $P$. After `intros po P nnp` we arrive at the state:

```
po : PO,
P : Prop,
nnp : ¬¬P
⊢ P
```

At this point, we will use PO to obtain more information. By using `cases po P with hp np`, we now obtain two new goals: One where we know `hp : P`, and one where we know `np : ¬P`. In the first case, the hypothesis `hp : P` is precisely the goal, so using `exact hp` closes it. In the second case, the hypothesis `np : ¬P` contradicts our earlier assumption that `nnp : ¬¬P`, so we can close this goal with the two tactics:

```
exfalso,
exact nnp np,
```

Having completed the proof of `PO → ∀ P : Prop, ¬¬P → P`, we will now have to prove `(∀ P : Prop, ¬¬P → P) → LPO`. The full proof is:

```
intro h,
apply PO_implies_LPO,
intro P,
apply h,
exact not\_not\_or P,
```

First we introduce our hypothesis `h : ∀ P : Prop, ¬¬P → P`. Then we change our goal from LPO to PO by using `PO_implies_LPO`. PO is stronger, but our proof will be simpler because we do not need to reason about sequences of natural numbers. We introduce the proposition P and apply the hypothesis `h` to change the goal from `P ∨ ¬P` to `¬¬(P ∨ ¬P)`. We then use the lemma `not_not_em` from the Lean core library:

```
lemma not_not_em (P : Prop) : ¬¬(P ∨ ¬P)
```

The tactic `exact not_not_em P` completes the proof.

#### 3.2.2.2 $\forall a, b \in \mathcal{N}[a \leq b \to a < b \vee a = b]$ is reckless

Next we would like to show the same for a statement that is equivalent to LPO. The statement we have chosen for this is:

$$\forall a, b \in \mathcal{N}[a \leq b \to a < b \vee a = b]$$

We will prove: `reckless_LPO (∀ a b : 𝒩, a ≤ b → a < b ∨ a =' b)`. We use the tactic `split` to obtain two goals: PO implies the statement, which itself implies LPO. For the first part of the proof, we obtain the following state after using the tactic `intros po a b hab`:

```
po : PO,
a b : 𝒩,
hab : a ≤ b,
⊢ a < b ∨ a =' b
```

We use PO to obtain `a < b ∨ ¬a < b`. This gives us two new cases to think about. In the first case, we know `a < b`. But this is exactly the left part of our goal. In the second case, we know `¬a < b`. We show that `a =' b` holds, using two theorems about natural sequences:

```
theorem le_iff_not_lt (a b : 𝒩) : a ≤ b ↔ ¬ b < a
theorem eq_of_le_le {a b : 𝒩}
  (hab : a ≤ b) (hba : b ≤ a) : a =' b
```

We already knew `a ≤ b`. The first theorem gives us `b ≤ a` from `¬ b < a`, and then we can use the second theorem to complete the proof.

For the second part of the proof, we are given that `∀ a b : ℕ, a ≤ b → a < b ∨ a =' b` is true, and an element `a` in $\mathcal{N}$, for which we have to show: `(∀ n : ℕ, a n = 0) ∨ ∃ n : ℕ, a n ≠ 0`. We use the following statements about natural sequences to help us:

```
def zero : 𝒩 := λ n, 0
lemma zero_le (a : 𝒩) : zero ≤ a
```

We can then use our hypothesis `∀ a b : ℕ, a ≤ b → a < b ∨ a =' b` to make a case distinction: either `zero < a` or `zero =' a`:

```
cases h1 nat_seq.zero a (nat_seq.zero_le a) with zlt zeq,
```

We can use the two following theorem to complete the proof for the first case, since `∃ n : ℕ, a n ≠ 0` is equivalent to `a # nat_seq.zero`.

```
theorem apart_iff_lt_or_lt (a b : 𝒩) : a # b ↔ a < b ∨ b < a
theorem apart_symm (a b : 𝒩) : a # b ↔ b # a
```

In the second case, we need to use `zero =' a`, from which we will show `∀ n : ℕ, a n = 0`. But this is easy if we use the theorem:

```
theorem eq_symm {a b: 𝒩} : a =' b ↔ b =' a
```

After all, `a =' b` is defined as `∀ n : ℕ, a n = b n`. So we can complete the proof with the tactic: `exact nat_seq.eq_symm.mp zeq`.

We now know that `a < b ∨ a =' b` is stronger than `a ≤ b`. This is precisely because of the ∨, which dictates that we have to know which case is true. Is `a < b` true, or is `a =' b` true. `a ≤ b` does not give this information.

## 3.3 Brouwer's Continuity Principle

Brouwer's Continuity Principle (BCP) is the following statement:
Let $R$ be a subset of $\mathcal{N} \times \mathbb{N}$, then:

$$\forall a \in \mathcal{N} \exists n \in \mathbb{N}[aRn] \to \forall a \in \mathcal{N} \exists m, n \in \mathbb{N} \forall b \in \mathcal{N}[\overline{a}m = \overline{b}m \to bRn]$$

In this definition, $\overline{a}m$ is the finite sequence of length $m$ which has the same start as $a$, or $a$ truncated after the first $m$ entries. Additionally, we write $aRn$ to denote $(a, n) \in R$.

In short, BCP says: if we have a relation $R$ such that for every sequence $a$ there is a natural number $n$ with $aRn$, then we should be able to decide this on the basis of an initial part of $a$. Brouwer justified this principle as follows: We say that we know $\forall a \in \mathcal{N} \exists n \in \mathbb{N}[aRn]$. This $a$ could also be constructed in a step-by-step fashion, walking along an infinite decision tree. However, at some point we need to provide the $n$ such that $aRn$. This has to be done while $a$ is not finished and we do not know how if will continue. So if we can provide $n$ after the first $m$ positions of $a$ are given, then $aRn$ is not only true for $a$, but for all continuations of $\overline{a}m$.

This justification is, however, not a proof. As such, we cannot formalize BCP as a `theorem`. The other possible options are using `axiom` or using `def`. We have chosen to formalize it as a definition, similar to how Bishop-style constructive mathematics is done. Brouwer used it as an axiom, but this is undesirable for a formalization because:

- Axioms are always true, but BCP is not true in classical mathematics. Formalizing BCP as a definition allows classical mathematicians to reason about statements like `¬BCP` and `¬BCP → P`.
- Axioms can always be used in a proof. This means that you would have to look at every proof to determine if BCP was used in it. If it is instead formalized as a definition, the statement of the theorem would explicitly mention that BCP is being used (`BCP → P` instead of `P`).

The full statement of BCP is then:

```
def BCP : Prop := ∀ R : 𝒩 → ℕ → Prop,
  (∀ a : 𝒩, ∃ n : ℕ, R a n) →
  ∀ a : 𝒩, ∃ m n : ℕ, ∀ b : 𝒩,
  (∀ i : ℕ, i < m → a i = b i) → R b n
```

This version is slightly different then the version given above. The first difference is the definition of $R$. Instead of $R$ being a subset of $\mathcal{N} \times \mathbb{N}$, it is now a function of type $\mathcal{N} \to \mathbb{N} \to$ `Prop`. Subsets, Cartesian products and membership testing is much more tedious than using a function to obtain a `Prop`, true or false. This definition allows us to write `R a n` instead of `(a, n) ∈ R`. All in all, this makes this definition easier to work with.

The second difference is the condition for the conclusion: `∀ i : ℕ, i < m → a i = b i`. This is also done to make things easier. With this definition, there is no need for reasoning about finite sequences and elements of `fin n`. This definition is equivalent, as demonstrated by the two following theorems which can be found in fin_seq.lean, in which we formalized finite sequences:

```
theorem finitize_initial_iff_start_eq (a b : 𝒩) (n : ℕ) :
  finitize a n ⊑ b ↔ (∀ j : ℕ, j < n → a j = b j)
theorem finitize_eq_iff_start_eq (a b : 𝒩) (n : ℕ) :
  finitize a n = finitize b n ↔ (∀ j : ℕ, j < n → a j = b j)
```

These two theorems demonstrate that the condition $\forall j \in \mathbb{N}[j < n \to a(j) = b(j)]$, is equivalent to $\overline{a}n \sqsubseteq b$ and to $\overline{a}n = \overline{b}n$. They also provide easy ways to convert between the two, in cases where reasoning about finite sequences is preferred.

### 3.3.1 Applications of Brouwer's Continuity Principle

Next, we will prove some statements that require BCP in their proofs. In section 3.1 we proved that the following theorem is true:

```
theorem uncountable (f : ℕ → 𝒩) : ∃ a : 𝒩, ∀ n : ℕ, a # (f n)
```

So: A function from $\mathbb{N}$ to $\mathcal{N}$ is never surjective. Using BCP, we can prove the opposite side of the same coin: A function from $\mathcal{N}$ to $\mathbb{N}$ is not injective. A classical mathematician may formalize this as:

```
theorem not_injective (f : 𝒩 → ℕ) :
  BCP → ¬(∀ a b : 𝒩, f a = f b → a =' b)
```

This is based on the definition of `injective` being:
`∀ a b : 𝒩, f a = f b → a =' b`. However, we would like to show that a stronger version of this statement also holds:

```
theorem strongly_not_injective (f : 𝒩 → ℕ) :
  BCP → ∀ a : 𝒩, ∃ b : 𝒩, a # b ∧ f a = f b
```

Note that this theorem states that `f` is nowhere injective. For any `a` in $\mathcal{N}$ we can construct a `b` in $\mathcal{N}$ such that `f(a) = f(b)` but `a # b`. For a function that is simply not injective, we would only have to show that such an `a` exists, but we claim that any `a` works.

The most important part of proving a statement with BCP is finding the correct relation $R$ to apply BCP to. For this proof, we will use $R$ as defined by $aRn \iff f(a) = n$. In Lean, we can define this with the `set` tactic:

```
set R : 𝒩 → ℕ → Prop := λ (a : 𝒩) (n : ℕ), f a = n with hr,
```

`R` will then be an element of type $\mathcal{N} \to \mathbb{N} \to$ `Prop` and `hr` will be the hypothesis `R = λ (a : 𝒩) (n : ℕ), f a = n`. To use BCP, we will now need

to know that ∀ a : $\mathcal{N}$, ∃ n : ℕ, R a n holds. This is simple, because we can choose n to be f a, which will make R a n equal to the statement f a = f a. In Lean:

```
have g₁ : ∀ a : 𝒩, ∃ n : ℕ, R a n,
{
  intro a,
  use f a,
  rw hr,
},
```

Now we are ready to use BCP:

```
have bcpa := (bcp R g₁) a,
cases bcpa with m bcpa_m,
cases bcpa_m with n bcpcon,
```

After these last few lines, we have now obtained the following proof state:

```
a : 𝒩,
m n : ℕ,
bcpcon : ∀ b : 𝒩, (∀ i : ℕ, i < m → a i = b i) → R b n
⊢ ∃ b : 𝒩, a # b ∧ f a = f b
```

We will now need to construct an element b of $\mathcal{N}$ such that a # b ∧ f a = f b holds. From bcpcon we know that f(b) = f(a) if $\overline{a}m = \overline{b}m$, so we will need to construct b such that a and b start out the same, but eventually become different. We have done this in the following lemma:

```
lemma exists_start_eq_ne (a : 𝒩) (n : ℕ) :
  ∃ b : 𝒩, (∀ i : ℕ, i < n → a i = b i) ∧ a n ≠ b n
```

We won't go into the proof in detail, but the most important line of the proof is the first line, where we define what b we will use for the rest of the proof. Everything else follows naturally.

```
set b : 𝒩 := λ i : ℕ, if i < n then a i else a i + 1 with hb,
```

The proof of our main theorem now continues as follows:

```
cases exists_start_eq_ne a m with b hb,
use b,
split,
```

After the tactic split we have two new goals: a # b and f a = f b. The first goal can easily be closed with the help of hb : (∀ i : ℕ, i < m → a i = b i) ∧ a m ≠ b m:

```
use m,
exact hb.elim_right,
```

To prove that `f a = f b` holds, we need to use the hypothesis `bcpcon`. We know from `hb` that `∀ i : ℕ, i < m → a i = b i` holds, and `∀ i : ℕ, i < m → a i = a i` always holds because = is reflexive. So we can conclude that `R a n` and `R b n` hold. So `f a = n` and `f b = n`, which lets us complete the proof with simple rewrites. In Lean:

```
have g₂ : R a n,
{
   apply bcpcon a,
   intros i hi,
   refl,
},
have g₃ : R b n,
{
   apply bcpcon b,
   intros i hi,
   exact hb.elim_left i hi,
},
rw hr at g₂ g₃,
rwa [g₂, g₃],
```

This completes the proof of `strongly_not_injective`. The weaker theorem `not_injective` now easily follows:

```
theorem not_injective (f : 𝒩 → ℕ) :
   BCP → ¬(∀ a b : 𝒩, f a = f b → a =' b) :=
begin
   intros bcp h, -- Need to prove: false
   have h0 := h nat_seq.zero, -- We know: f 0̲ = f b → 0̲ =' b
   cases strongly_not_injective f bcp nat_seq.zero with b hb,
   have hb0 := h0 b hb.elim_right, -- Conclude: 0̲ =' b,
   exact (nat_seq.ne_of_apart _ _ hb.elim_left) hb0,
end
```

In the third line we conclude that `zero # b ∧ f zero = f b`. In the last line, we obtain a contradiction by combining `zero # b` and `zero =' b`. We used `nat_seq.zero` in this proof because we had defined it previously. Any other element of $\mathcal{N}$ would have worked too.

One thing may seems strange about the theorem `strongly_not_injective` though, and that is that it claims that every function $\mathcal{N} \to \mathbb{N}$ is nowhere injective. Surely we could define a function that sends at least one element to a unique natural number? For example the following function may seem to contradict the theorem:

$$f(a) = \begin{cases} 0 \text{ if } a = \underline{0} \\ 1 \text{ otherwise} \end{cases}$$

It seems like $\underline{0}$ is the only element of $\mathcal{N}$ that gets send to 0 by $f$. However, $f$ is not a function, because we cannot determine for every element $b$ of $\mathcal{N}$ what $f(b)$ will be. For example, we do not know what the output of $f$ will be for the element $a_{99}$ (as defined in section 3.2). The problem with functions like $f$ defined above is that equality on $\mathcal{N}$ is not decidable. However, using LPO we could make it decidable. Recall:

```
def LPO : Prop := ∀ a : 𝒩, (∀ n : ℕ, a n = 0) ∨ (∃ n : ℕ, a n ≠ 0)
def eq (a b : 𝒩) : Prop := ∀ n : ℕ, a n = b n
```

By using LPO, we can determine where every element of $\mathcal{N}$ is sent by $f$. The problem with this is: BCP and LPO cannot be true at the same time.

```
theorem BCP_implies_not_LPO : BCP → ¬LPO
```

So if we assume that BCP is true, then LPO is not merely a statement that has no constructive proof, it is actually false. Of course, this means that the opposite is true too: In classical mathematics, where LPO is true, BCP is false.

The proof of the theorem relies on the correct choice of a relation $R$ to apply BCP to. Our proof begins by assuming that BCP and LPO are true with the aim of proving `false`. Then we define $R$:

```
intros bcp lpo,
set R : 𝒩 → ℕ → Prop := λ a, λ i, if i = 0
  then ∀ n : ℕ, a n = 0 else ∃ n, a n ≠ 0 with hR,
```

We now need to prove that `∀ a : 𝒩, ∃ n : ℕ, R a n` holds, because that is required to apply BCP to `R`. We can use LPO for this: If LPO says that `∀ n : ℕ, a n = 0` is true, then we know $aR0$ holds, and if LPO says that `∃ n, a n ≠ 0` holds, then we know that $aR1$ (or $aRn$ for any $n \neq 0$) holds. In Lean:

```
have hr : ∀ a : 𝒩, ∃ n : ℕ, R a n, by
{
  intro a,
  cases lpo a with aeq0 ane0,
  {-- case: ∀ n : ℕ, a n = 0
    use 0,
    rw hR,
    split_ifs, --We only need to consider the case 0 = 0
    exact aeq0,
  },
  {-- case: ∃ n : ℕ, a n ≠ 0
    use 1,
    rw hR,
    split_ifs,
    {-- case: 1 = 0, impossible
```

```
            exfalso,
            exact nat.one_ne_zero h,
        },
        {-- case: ¬1 = 0, need to prove: ∃ n : ℕ, a n ≠ 0
            exact ane0,
        }
    }
},
```

At this point we can apply BCP to any element of $\mathcal{N}$ and work from there. We will work with $a = \underline{0}$ (`nat_seq.zero`). BCP should then give us that $aR0$ holds, and that any element $b$ of $\mathcal{N}$ starting with sufficiently many zeroes also satisfies $bR0$.

```
have bcp_r_0 := (bcp R hr) nat_seq.zero,
cases bcp_r_0 with m bcp_r_0₁,
cases bcp_r_0₁ with n bcp_r_0₂,
cases nat.eq_zero_or_pos n with hn₁ hn₂,
```

From these 4 lines we know that $bRn$ holds if $b$ starts with `m` zeroes. The last line allows us to distinguish between two cases: either `n = 0` or `n > 0`. Of course we know from the definition of `nat_seq.zero` and `R` that only `n = 0` can be true at this point, but since our goal is to obtain a contradiction and prove `false`, considering what happens if `n > 0` can help us get closer to our goal. In the case that `n > 0`, we can apply the conclusion of BCP to $\underline{0}$ again, obtaining the hypothesis:

```
h₀ : (∀ i : ℕ, i < m → 0̲ i = 0̲ i) → R 0̲ n
```

(The above line is shortened by using $\underline{0}$ in place of `nat_seq.zero` to make everything fit on the same line, making it more readable. Lean always shows `nat_seq.zero` as the name for the sequence where all entries are 0).
Since $\underline{0}$ i = $\underline{0}$ i always holds because equality is reflexive, we can now easily conclude that `R nat_seq.zero n` must hold:

```
have h₁ : (∀ i : ℕ, i < m → nat_seq.zero i = nat_seq.zero i),
  by simp,
have h₂ := h₀ h₁,
```

We now use the definition of `R` to make progress, changing `h₂` from `R nat_seq.zero n` into `ite (n = 0) (∀ n : ℕ, nat_seq.zero n = 0) (∃ n : ℕ, nat_seq.zero n ≠ 0)`. Using `split_ifs`, we obtain two new goals. In the first goal, we may assume `n = 0` holds. But this is a clear contradiction with our earlier assumption that `n > 0`. So we can prove `false` in one line: `exact (ne_of_gt hn₂) h`,

In the second case, we can assume `¬ n = 0`, which gives us no valuable information, but we also know that `(∃ n : ℕ, nat_seq.zero n ≠ 0)`

holds. However, this is a contradiction with the definition of `nat_seq.zero`, so we can prove `false` in this case as well:

```
cases h₂ with k hk,
apply hk,
refl,
```

This proves that `n > 0` is impossible. We jump back now to immediately after the line
`cases nat.eq_zero_or_pos n with hn₁ hn₂,`
and consider the case where `n = 0`. In this case, the contradiction will come from the conclusion of BCP for our choice of $R$, which tells us that a sequence starting with `m` zeroes can only be $\underline{0}$. This yields a contradiction, as we can easily define the following element of $\mathcal{N}$ which proves that this is false:

```
set b : 𝒩 := λ k, if k < m then 0 else 1 with hb,
```

Completing the proof is now easy. We prove that `b` starts with `m` zeroes, which is true by definition of `b`. Then we obtain the conclusion `R b n`, which we can rewrite to `∀ n : ℕ, b n = 0`. Since this holds for any natural number, it must also hold for `m`. But by definition, `b m = 1`, and `¬1 = 0` is false (as stated by the lemma `nat.one_ne_zero`).

This concludes the proof of `BCP → ¬LPO`. An immediate corollary to this is `BCP → ¬PO`, since we already know that `PO → LPO`. We can also improve our claim by proving `BCP → ¬WLPO` without any major alterations to our proof. The only major difference will be our definition of `R`, which will now have to be:

```
set R : 𝒩 → ℕ → Prop := λ a, λ i, if i = 0
  then ∀ n : ℕ, a n = 0 else ¬∀ n : ℕ, a n = 0 with hR,
```

The third application of BCP that we have formalized is the following theorem:

```
theorem apart_iff_forall_ne_or_ne (bcp : BCP) (a b : 𝒩) :
  a # b ↔ ∀ c : 𝒩, a ≠' c ∨ c ≠' b
```

The implication `a # b → ∀ c : 𝒩, a ≠' c ∨ c ≠' b` is a direct consequence of two theorems from `nat_seq`:

```
theorem apart_cotrans (a b : 𝒩) (h : a # b) :
  ∀ c : McN, a # c ∨ c # b
theorem ne_of_apart (a b : 𝒩) : a # b → a ≠' b
```

The interesting implication is the one the other way around. We will not go into detail for the proof, since it is very similar to the previous two theorems. It is the standard formula for this chapter: Choose a good `R`, show that the preconditions for BCP hold, apply BCP, finish the proof. We will leave you with the definition of `R`, omitting the rest of the proof. The principle behind this definition of `R` is the same one we used in the previous proof:

```
set R : 𝒩 → ℕ → Prop := λ c, λ n, if n = 0 then c ≠' a else c ≠' b with hR,
```

## 3.4 Rational Segments and Real Sequences

Our next objective will be to define the intuitionistic version of the real numbers in Lean. Lean already has a definition of the classical real numbers using Cauchy sequences. Understanding the full details of the classical construction is not needed to understand the intuitionistic version, but you can find the full definition of the real numbers as found in mathlib below to make comparing the two definitions easier.

```
def is_cau_seq {α : Type*} [discrete_linear_ordered_field α]
  {β : Type*} [ring β] (abv : β → α) (f : ℕ → β) : Prop :=
  ∀ ϵ > 0, ∃ i, ∀ j ≥ i, abv (f j − f i) < ϵ
def cau_seq {α : Type*} [discrete_linear_ordered_field α]
  (β : Type*) [ring β] (abv : β → α) : Type* :=
  {f : ℕ → β // is_cau_seq abv f}
instance equiv : setoid (cau_seq β abv) :=
  λ f g, lim_zero (f − g)
def Cauchy := @quotient (cau_seq _ abv) cau_seq.equiv
def real := @cau_seq.completion.Cauchy ℚ _ _ _ abs _
```

This definition is constructive, and is similar to how Troelstra and Van Dalen define the real numbers with Cauchy sequences.[15]

This definition uses equivalence classes. This could be an issue, since equivalence may not be decidable for some sequences, so it is not clear for all Cauchy sequences in which equivalence class they belong. This is similar to the problem of comparing natural sequences, where we could not always determine if an element of $\mathcal{N}$ was equal to $\underline{0}$ or not. The definition we will use for the real numbers will therefore not use equivalence classes. Instead, we will define an equality on real sequences, and deal with two real sequences being equal to each other when it is relevant for the theorem we are proving.

### 3.4.1 Rational Segments

Before we can define the intuitionistic real numbers, we first have to define rational segments. These segments will represent an interval on the number line. By taking a sequence of rational segments that keep getting smaller, we can hone in on a real value. The definition of rational segments we will use is the following:

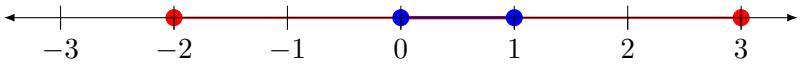
```
def segment := {s : ℚ × ℚ // s.fst ≤ s.snd}
notation `𝕊` := segment
def fst (s : 𝕊) : ℚ := (subtype.val s).fst
def snd (s : 𝕊) : ℚ := (subtype.val s).snd
```

Segments are defined as a subtype of $\mathbb{Q} \times \mathbb{Q}$. The definition speaks for itself. We introduce the notation $\mathbb{S}$, `s.fst` and `s.snd` to make working with

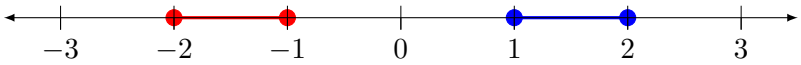
rational segments easier. To get the information `s.fst ≤ s.snd` from a rational segment `s`, we can use `subtype.property s`.

Next, we will define some comparisons on $\mathbb{S}$. Since a segment represents an interval, there are two important notions here. The first is when a segment contains another, and the second is when a segment is less that the other.

The picture below shows an example of one segment that contains another segment. In this case, the segment (-2, 3) contains the segment (0, 1). Also denoted as (0, 1) ⊑ (-2, 3).




The next picture shows an example of one segment that is less than another. We define being less than as laying completely to the left of the other segment. We denote this as (-2, -1) ≤ (1, 2).




The pictures above should help with the intuition for what the two comparisons represent. The formalization in Lean is as follows:

```
def contained (s t : 𝕊) : Prop := t.fst ≤ s.fst ∧ s.snd ≤ t.snd
infix ⊑“ := contained
def proper_contained (s t : 𝕊) : Prop :=
  t.fst < s.fst ∧ s.snd < t.snd
infix ⊏“ := proper_contained
def lt (s t : 𝕊) : Prop := s.snd < t.fst
infix ‘<‘ := lt
def le (s t : 𝕊) : Prop := s.fst ≤ t.snd
infix ‘≤‘ := le
```

We have a defined both the normal comparisons and strict comparisons. These definitions completely capture the behaviour that can be seen in the images above. The definition ≤ makes the three following statements trivial:

```
lemma le_iff_not_lt (s t : 𝕊) : s ≤ t ↔ ¬ t < s
lemma lt_iff_not_le (s t : 𝕊) : s < t ↔ ¬ t ≤ s
theorem le_refl (s : 𝕊) : s ≤ s
```

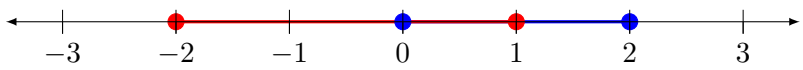
We also have that ⊑ and ⊏ are transitive. The proofs of these theorems rely only on the definition of ⊑ and ⊏ and the fact that ≤ and < on $\mathbb{Q}$ are transitive. ⊑ is also reflexive.

```
theorem contained_refl (s : 𝕊) : s ⊑ s
theorem contained_trans (s t v : 𝕊)
  (h₁ : s ⊑ t) (h₂ : t ⊑ v) : s ⊑ v
theorem proper_contained_trans (s t v : 𝕊)
  (h₁ : s ⊏ t) (h₂ : t ⊏ v) : s ⊏ v
```

We will use the following lemma when proving that our addition on the real numbers is well-defined:

```
lemma contained_bounds_le (s t : 𝕊) (h : s ⊑ t) :
  s.snd − s.fst ≤ t.snd − t.fst :=
begin
  apply sub_le_sub,
  exact h.elim_right, -- snd s ≤ snd t
  exact h.elim_left, -- fst s ≤ fst t
end
```

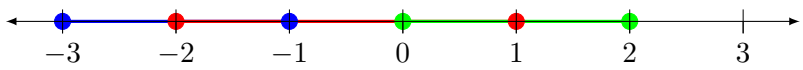
After the first line, we obtain two goals, which we can immediately close because they are part of the definition of `s ⊑ t`. The only thing we need to do now in preparation of defining real numbers is definition the relation ≈, which represents when two segments touch each other in some way. This can be when one contains the other, or when only a portion of their intervals overlap with each other. We already know what the first case looks like. The second case is shown in the picture below:




The crucial part of this picture is that the leftmost edge of both segments is to the left of the rightmost edge of the other segment. In Lean, we capture this notion by using the definition of ≤:

```
def touches (s t : 𝕊) : Prop := s ≤ t ∧ t ≤ s
infix ‘≈‘ := touches
```

This relation is reflexive and transitive, but is does not define an equivalence relation because it is not transitive. A counterexample to ≈ being transitive is shown in the picture below:




(-3, -1) touches (-2, 1) and (-2, 1) touches (0, 2), but (-3, -1) and (0, 2) do not touch at all. Now we will prove that ≈ is reflexive and transitive. These theorems are immediately obvious from the definition of ≈ and that ≤ on 𝕊 is reflexive:

```
theorem touches_refl (s : 𝕊) : s ≈ s :=
begin
  split,
  refl,
  refl,
end

theorem touches_symm (s t : 𝕊) : s ≈ t ↔ t ≈ s :=
```

```
begin
  exact and.comm,
end
```

### 3.4.2 Real Sequences

Now we are ready to define the intuitionistic real numbers. We will use the name real sequences for them, to avoid confusion with the classical real numbers that are already defined in Lean. A real sequence will be a sequence of rational segments that is both shrinking and dwindling. We call a sequence shrinking if each consecutive entry is contained within the previous one, and we call a sequence dwindling if for every positive rational number we can find a segment in the sequence such that the interval defined by that the length of that segment is smaller than our given rational number. In Lean, these two properties and real sequences are defined as follows:

```
def shrinking (r : ℕ → 𝕊) := ∀ n, r (n + 1) ⊑ r n
def dwindling (r : ℕ → 𝕊) :=
  ∀ q : ℚ, q > 0 → ∃ n : ℕ, (r n).snd − (r n).fst < q
def real_seq := {r : ℕ → 𝕊 // shrinking r ∧ dwindling r}
notation ‘ ℛ ‘ := real_seq
def seq (r : ℛ) : ℕ → 𝕊 := subtype.val r
```

Our definition of shrinking only says that each consecutive segment has to be contained in the previous one, but using induction on the natural numbers and transitivity of $\sqsubseteq$ on $\mathbb{S}$, we can easily proof the following theorem:

```
theorem contained_of_le (r : ℛ) {n m : ℕ} (h₁ : n ≤ m) :
  r.seq m ⊑ r.seq n
```

We will now define some comparisons on $\mathcal{R}$. These all rely on the comparisons between rational segments.

```
def lt (x y : ℛ) : Prop := ∃ n : ℕ, x.seq n < y.seq n
infix ‘<‘ := lt
def le (x y : ℛ) : Prop := ∀ n : ℕ, x.seq n ≤ y.seq n
infix ‘≤‘ := le
def apart (x y : ℛ) : Prop := x < y ∨ y < x
infix ‘#‘ := apart
def eq (x y : ℛ) : Prop := ∀ n : ℕ, x.seq n ≈ y.seq n
infix ‘=’‘:50 := eq
def ne (x y : ℛ) : Prop := ¬ x =’ y
infix ‘≠’‘:50 := ne
```

We also need the fact that real sequences are shrinking for these definitions to make sense. For example, the definition of $<$ only says that at one point the first segment needs to be less than the second. However, because both

sequences are shrinking, this implies that all segments after that point also satisfy that the first is less than the second.

We can immediately prove some things about these comparisons. For example, writing out the definitions and using the theorem `contained_of_le` shown above shows that $<$ is transitive. That $<$ is also co-transitive requires a proof that also needs us to use the definition of the real numbers themselves.

```
theorem lt_cotrans (x y z : ℝ) (h₁ : x < y) : x < z ∨ z < y
```

The first thing we do is write out the definition of `x < y`, using the tactic `cases h₁ with n hn`. After some rewriting, we can obtain the following information:

```
h₁ : segment.fst (seq y n) - segment.snd (seq x n) > 0
```

Crucially, `segment.fst (seq y n) - segment.snd (seq x n)` can be seen as a rational number $q \in \mathbb{Q}$, and we have just proven that $q > 0$. We know that `z` is a real number, so it is dwindling. We can combine these two facts:

```
cases z.dwindling
  (segment.fst (seq y n) − segment.snd (seq x n)) hn with m hm,
```

This gives us the following information:

```
m : ℕ,
hm : segment.snd (z.val m) - segment.fst (z.val m) <
  segment.fst (seq y n) - segment.snd (seq x n)
```

We will now need the following lemma about rational numbers, for which we will omit the proof for brevity:

```
lemma lt_or_lt_from_sub_lt_sub {a b c d : ℚ}
  (h : a − b < c − d) : a < c ∨ d < b
```

We can apply this lemma to `hm`, from which we obtain the information that either `segment.snd (z.val m) < segment.fst (seq y n)` or `segment.snd (seq x n) < segment.fst (z.val m)`. In the first case, we can prove that `z < y`, and in the second case we can prove that `x < z`, which would complete our proof that $<$ is co-transitive. We will only show the proof for `z < y`, as the second proof is completely analogous to the first. To show that `z < y` holds, we first have to provide a natural number for which we will prove that the segment $z(n)$ lays to the left of $y(n)$. Since real numbers are shrinking, and we know `segment.snd (z.val m) < segment.fst (seq y n)`, we could use any number bigger than or equal to $max(m, n)$. We will use $max(m, n)$ itself. We will also use the tactic `right` to change our goal from `x < z ∨ z < y` to `z < y`:

```
right,
```

```
use max m n,
```

Now we have to prove `seq z (max m n) < seq y (max m n)`. We now use the theorem `contained_of_le` twice. Though not necessary for the proof, we will show the result of writing out the definition of $\sqsubseteq$ here to make the proof easier to follow. $\mathtt{h}_1$ and $\mathtt{h}_2$ contain these results, the rest is information we already knew and our goal:

```
zlty : segment.snd (z.val m) < segment.fst (seq y n)
h₁ : segment.fst (seq z m) ≤ segment.fst (seq z (max m n))
   ∧ segment.snd (seq z (max m n)) ≤ segment.snd (seq z m),
h₂ : segment.fst (seq y n) ≤ segment.fst (seq y (max m n))
   ∧ segment.snd(seq y (max m n)) ≤ segment.snd (seq y n)
⊢ seq z (max m n) < seq y (max m n)
```

We can now use the two following lemmas:

```
lemma lt_of_le_of_lt [preorder α] : ∀ {a b c : α},
  a ≤ b → b < c → a < c
lemma lt_of_lt_of_le [preorder α] : ∀ {a b c : α},
  a < b → b ≤ c → a < c
```

The last two tactics complete the proof:

```
apply lt_of_le_of_lt h₁.elim_right,
exact lt_of_lt_of_le zlty h₂.elim_left,
```

This was a decent amount of work for a fact that does not get talked about much by classical mathematicians. The reason for this is simple: In classical mathematics, co-transitivity is a trivial consequence of transitivity and the fact that $a < b \vee a = b \vee b < a$ (`lt_trichotomy`) holds for all real numbers $a$ and $b$. A classical proof would go as follows: Let $x$, $y$ and $z$ be real numbers such that $x < y$. We need to show: $x < z \vee z < y$. In the case that $x < z$, we are done. In the case that $x = z$, then $x < y$ tells us $z < y$. In the case that $z < x$, then we get $z < y$ by transitivity.

### 3.4.3 Trichotomy on $\mathcal{R}$ is reckless

So why did we not use this in our proof? The reason is that sometimes, an intuitionistic mathematicians cannot tell which case of the three is true, so we cannot say that the whole statement is true. The statement $\forall x, y \in \mathcal{R}[x < y \vee x = y \vee y < x]$ is actually equivalent to LPO. We will show here that it is reckless by proving that it is implied by PO, but that it implies LPO.

```
theorem reckless_LPO_real_lt_eq_gt :
  reckless_LPO (∀ x y : ℝ, x < y ∨ x =' y ∨ y < x)
```

We will not show the full proof of the fact that PO implies that the statement is true. However, the main ideas are: Use PO on the propositions `x < y` and `y < x`, together with the two following theorems about real numbers:

```
theorem le_iff_not_lt (x y : ℝ) : x ≤ y ↔ ¬ y < x
theorem eq_of_le_of_le (x y : ℝ) : x ≤ y → y ≤ x → x =' y
```

The more interesting part is that `∀ x y : ℝ, x < y ∨ x =' y ∨ y < x` implies that LPO is true. Recall that LPO is the statement:

$$\forall a \in \mathcal{N}[\forall n[a(n) = 0] \vee \exists n[a(n) \neq 0]]$$

So in proving that our statement implies LPO, we will have to make a connection between the real numbers and sequences of natural numbers. For this we have defined the function `snap` which maps an element $a \in \mathcal{N}$ to a real number $x \in \mathcal{R}$. The full definition of `snap` is:

```
def snap (a : 𝒩) : ℝ := subtype.mk (λ n : ℕ,
  if h : (∃ i : ℕ, i ≤ n ∧ a i ≠ 0)
  then segment.inclusion (1 / nat.succ (nat.find h))
  else segment.two_sided_inclusion (1 / nat.succ n)
def inclusion (q : ℚ) : 𝕊 := subtype.mk (q, q)
def two_sided_inclusion (q : ℚ) (hq : q > 0) : 𝕊 :=
  subtype.mk (−q, q)
```

Let's examine the behaviour of `snap` with an example. Say we are inspecting the element $a = \langle 0, 0, 0, 1, ...\rangle \in \mathcal{N}$. The first ($0^{\text{th}}$) entry of $a$ is 0, so `∃ i : ℕ, i ≤ n ∧ a i ≠ 0` (with `n = 0`), is false. We go to the `else` clause in our definition, and find that the first segment of our real number will be (-1, 1). The second entry is also 0, so our next segment is $(-\frac{1}{2}, \frac{1}{2})$. The same goes for the third entry, which gives the segment $(-\frac{1}{3}, \frac{1}{3})$. If our $a$ was equal to $\underline{0}$, then this pattern would continue, with the left and right boundaries of the segments getting closer and closer to 0. In that case, `snap a` would be equal to $0_{\mathcal{R}}$. However, the next entry in our example is a 1, which means that `∃ i : ℕ, i ≤ n ∧ a i ≠ 0` is true (for `n = 3`). We go to the `if` clause in our definition. This `if` clause is why we called the function `snap`: The real number we are defining will now snap into place. `nat.find h` will find us the smallest `n` such that the statement in `h` is true, which is the `n` we are currently at: `n = 3`. So our next segment will be $(\frac{1}{4}, \frac{1}{4})$. Whatever the next entry into the sequence is, `h` will now always stay true with `i = 3`, and `nat.find h` will always return 3, so from now on every segment will be $(\frac{1}{4}, \frac{1}{4})$. The real number that we have defined here represents the number $\frac{1}{4}$.

Some more observations about `snap`:

- $(\forall n[a(n) = 0] \iff \text{snap a} = 0_{\mathcal{R}}) \wedge$
  $(\exists i[a(i) \neq 0] \iff \text{snap a} < 0_{\mathcal{R}} \vee \text{snap a} > 0_{\mathcal{R}})$. We will prove this later, because this is a crucial part of the proof that

$\forall x, y \in \mathcal{R}[x < y \vee x = y \vee y < x]$ implies LPO and the reason why we defined `snap` like this in the first place.

- The definition uses `nat.succ` in the denominator of fractions. This is purely to prevent dividing by 0 and is of no further significance.

- Making an element of a subtype requires providing the element and a proof that it satisfies the subtype property. For the definitions of `inclusion` and `two_sided_inclusion`, that means proving that `q` $\leq$ `q` and `-q` $\leq$ `q` for `q > 0`. For `snap`, we have to prove that `snap a` is an element of $\mathcal{R}$, which requires us to prove that it is shrinking and dwindling. All three of these proofs have been omitted.

Now it's time to prove that `∀ x y : ℛ, x < y ∨ x =' y ∨ y < x` implies LPO. The first line of our proof is `intros h a`, where `h` will be the assumption `∀ x y : ℛ, x < y ∨ x =' y ∨ y < x` and `a` will be any element of $\mathcal{N}$. Now, we can use `h` to obtain information about `a` by reasoning about `snap a`:

```
have hsnap := h (snap a) (real_seq.inclusion_const 0),
cases hsnap with hlt hge,
```

Here `real_seq.inclusion_const 0` is the sequence that is always 0, so it represents the real number $0_{\mathcal{R}}$. When talking about these sequences in Lean, we have to say things like `real_seq.inclusion_const 0` and `real_seq.seq (snap a)` to talk about the sequence itself (without the proof that it is a real number). Writing everything down in this way can get very long, so we will shorten everything a bit from here on out to improve readability.

We obtain the information $snap(a) < 0_{\mathcal{R}} \vee snap(a) = 0_{\mathcal{R}} \vee snap(a) > 0_{\mathcal{R}}$. We do a case distinction on this.

The first case is $snap(a) < 0$. This is actually impossible, since by definition of `snap` we have that $snap(a) \geq 0_{\mathcal{R}}$. So we use `exfalso` to change our goal to `false`. We then use `cases hlt with n hn` to write out the definition of $<$ on $\mathcal{R}$, obtaining a natural number `n` together with the assumption:

```
hn : (snap a) n < (inclusion_const 0) n
```

Writing out the full definitions of everything and using `split_ifs` on the condition in `snap`, we obtain two new goals. In the first we have to prove `false` from `(nat.find _ + 1)`$^{-1}$ `< 0`, and in the second we have to prove `false` from `(n + 1)`$^{-1}$. We do this in both cases in the same way: Any natural number plus one is bigger than 0, so the inverse is also bigger than zero, which yields the contradiction we were looking for. The full code for this will be below, though it may not be easily readable since we used the tactic `simp` to do the work for us instead of writing out everything ourselves. We only had to indicate which theorems and lemmas are needed.

```
simp [real_seq.seq, real_seq.inclusion_const,
    segment.inclusion, snap, segment.two_sided_inclusion,
    segment.lt, segment.fst, segment.snd] at hn,
split_ifs at hn,
{-- case: ∃ i : ℕ, i ≤ n ∧ a i ≠ 0
   apply not_le_of_lt hn,
   simp [le_of_lt, nat.cast_add_one_pos],
},
{-- case: ¬∃ i : ℕ, i ≤ n ∧ a i ≠ 0
   apply not_le_of_lt hn,
   simp [le_of_lt, nat.cast_add_one_pos],
}
```

Next we will look at the case `snap a =' inclusion_const 0`. In this case we will prove that $\forall n \in \mathbb{N}[a(n) = 0]$ holds, proving that LPO is true in this case.

```
intro n,
have hn := heq n,
```

After the two lines above, we now have the goal `a n = 0`, and the hypothesis:

```
hn : (snap a) n ≈ (inclusion_const 0) n
```

The definition of $\approx$ on $\mathbb{S}$ now gives us that:

```
hn : ((snap a) n).fst ≤ ((inclusion_const 0) n).snd ∧
   ((inclusion_const 0) n).fst ≤ ((snap a) n).snd
```

We care mostly about the part before the $\wedge$, because rewriting the part after it only gives us the information that a natural number is bigger than or equal to 0, which we already knew. Writing out the full defintions and using `split_ifs` on the left part of `hn`, we get two cases.

In the first case, we know `∃ i : ℕ, i ≤ n ∧ a i ≠ 0` and `(nat.find _ + 1)`$^{-1}$ `≤ 0`. That last part, which is derived from the definition of `snap` and the assumption `hn`, is actually impossible for the same reason as mentioned previously. We close this case in the same way as before.

In the second case, we know `¬∃ i : ℕ, i ≤ n ∧ a i ≠ 0` and `-(n + 1)`$^{-1}$ `< 0`. There is now nothing wrong with the second part, so we must use the first part to prove `a n = 0`. We can use the theorem `for_all_of_not_exists` to obtain `∀ i : ℕ, ¬ (i ≤ n ∧ a i ≠ 0)`. When we aply this to `n`, we know that `i ≤ n` is true, so `a i ≠ 0` must be false. Equality on natural numbers is decidable, so $\neg a(n) \neq 0$ implies $a(n) = 0$, which closes the goal of proving `a n = 0` for the case `snap a =' inclusion_const 0`. The last few lines can be written down much shorter in Lean with the help of `simp`:

```
have han := (forall_not_of_not_exists h_1) n,
```

```
simp at han,
exact han,
```

The last case we need to consider is `inclusion_const 0 < snap a`. In this case, we know that `snap` snapped to the right of 0, so there must be a natural number $n$ such that $a(n) \neq 0$. We can prove LPO by making that our goal with the tactic `right`. We can write out the definitions just like in the previous two cases and use `split_ifs` to obtain two cases.

In the first case, we know `∃ i : ℕ, i ≤ n ∧ a i ≠ 0`. Ignoring that `i ≤ n`, this is actually our goal. So we can close the goal in the following way:

```
cases h_1 with i hi,
use i,
exact hi.elim_right,
```

In the second case, we know `¬∃ i : ℕ, i ≤ n ∧ a i ≠ 0` and `0 < -(n + 1)`$^{-1}$. This last hypothesis is actually impossible, so we can prove `false` from it in the following way.

```
exfalso,
rwa [lt_neg, neg_zero] at hn,
apply not_le_of_lt hn,
simp [le_of_lt, nat.cast_add_one_pos],
```

This completes the proof that $\forall xy \in \mathcal{R}, x < y \vee x =' y \vee y < x$ implies LPO, and also the proof that the statement is reckless. We did not actually need $y$ to be a free variable in this, since we have just proven that $\forall x \in \mathcal{R}, x < 0_{\mathcal{R}} \vee x =' 0_{\mathcal{R}} \vee 0_{\mathcal{R}} < x$ already implies that LPO is true. A version of this theorem with a fixed $y \neq 0$ could also always be reduced to the version with only $x$ and 0 by the observation that $x - y$ is also a real number, so by choosing $x - y$ as our real number to think about, $x - y < 0_{\mathcal{R}} \vee x - y =' 0_{\mathcal{R}} \vee 0_{\mathcal{R}} < x - y$ becomes $x < y \vee x =' y \vee y < x$. These two versions are therefore completely equivalent. We only choose the version with $y$ as a free variable because it looks more general than the version where $y$ is fixed to be $0_{\mathcal{R}}$.

### 3.4.4 Defining addition on $\mathcal{R}$

The last argument in the previous section only makes sense if we have defined addition and subtraction on the real numbers and some of their properties. To define addition on real sequences, we first have to define addition for rational segments. The definition we use for this is:

```
def add (s t : 𝕊) : 𝕊 := subtype.mk (s.fst + t.fst, s.snd + t.snd)
```

We simply add the first part of both segments together and do the same for the second part. For this to be a considered an element of $\mathbb{S}$, we also have

to prove that is satisfies the subtype property (∀ `s : 𝕊, s.fst ≤ s.snd`). We can do this for `s + t` by combining that we already know that `s` and `t` are elements of 𝕊:

```
begin
  apply add_le_add,
  exact subtype.property s,
  exact subtype.property t,
end
```

Addition on 𝕊 is associative and commutative. The proofs of this consist of writing out the definitions and using that addition is associative and commutative on ℚ.

The definition for addition on real sequences is to add the segments in both sequences to each other.

```
def add (x y : ℛ) : ℛ := subtype.mk
  (λ n, segment.add (x.seq n) (y.seq n))
```

Again, we need to prove that this definition is really an element of ℛ by showing that it satisfies the subtype property. That is, it needs to be shrinking and dwindling. We start our proof by obtaining the subtype properties of `x` and `y` and split the goal into its two different components:

```
have hx := subtype.property x,
have hy := subtype.property y,
split,
```

The proof that `x + y` is shrinking relies purely on the facts that `x` and `y` are shrinking and the lemma:

```
lemma add_le_add {a b c d : α} (h₁ : a ≤ b) (h₂ : c ≤ d) :
  a + c ≤ b + d
```

The full proof of `shrinking` is then:

```
intro n,
split,
{
  apply add_le_add,
  exact (hx.elim_left n).elim_left,
  exact (hy.elim_left n).elim_left,
},
{
  apply add_le_add,
  exact (hx.elim_left n).elim_right,
  exact (hy.elim_left n).elim_right,
}
```

We will now prove that `x + y` is also dwindling. This will primarily rely on the fact that `x` and `y` are themselves dwindling. We can use an argument you could also find in a real analysis textbook when proving $\lim_{n\to\infty} a_n + b_n = \lim_{n\to\infty} a_n + \lim_{n\to\infty} b_n$ for convergent sequences $(a_n)$ and $(b_n)$. We need to prove that a segment in the sequence `x + y` is smaller than some `q > 0`. Because `x` and `y` are dwindling, we can find a point for each of them where they have a segment smaller than $\frac{q}{2}$. We can then use the fact that they are both shrinking (in the form of the the theorem `contained_of_le` we introduced previously) to prove that they must share an index where their segments are both smaller than $\frac{q}{2}$. Adding these together will then always yield a segment smaller than `q`.

This proof translates well into Lean. We start with `intros q hq`, which gives us `hq : q > 0` for a rational number `q` and the goal:

```
⊢ ∃ n : ℕ, segment.snd (segment.add (seq x n) (seq y n)) -
  segment.fst (segment.add (seq x n) (seq y n)) < q
```

To apply the fact that `x` and `y` are dwindling to $\frac{q}{2}$, we first need to prove $\frac{q}{2} > 0$. We do this with the lemma `div_pos`, which gives us two goals: `0 < q` and `0 < 2`. The first one is our assumption `hq` and the second one already proven in Lean under the name `zero_lt_two` and `two_pos` (the only difference is that `two_pos` works for ordered fields while `zero_lt_two` works for linear ordered semirings). We can then use that `x` and `y` are dwindling.

```
have hq2 : q / 2 > 0, by
{
  rw gt_iff_lt,
  apply div_pos,
  exact gt_iff_lt.elim_left hq,
  exact zero_lt_two,
},
cases hx.elim_right (q / 2) hq2 with xn hxn,
cases hy.elim_right (q / 2) hq2 with yn hyn,
```

This gives us the indices `xn` and `yn` at which their respective segments become less than $\frac{q}{2}$. We now need to pick some number that is bigger than or equal to both. We use `max xn yn` for this. After writing out the definitions involved and doing some simplifying, we arrive at the following proof state:

```
q : ℚ
hq : q > 0
hq2 : q/2 > 0
xn : ℕ,
yn : ℕ,
hxn : segment.snd (x.val xn) - segment.snd (x.val xn) < q/2,
```

```
hyn : segment.snd (y.val yn) - segment.snd (y.val yn) < q/2,
⊢ (seq x (max xn yn)).val.snd - (seq x (max xn yn)).val.fst +
((seq y (max xn yn)).val.snd - (seq y (max xn yn)).val.fst) < q
```

We would now like to prove that everything before the plus sign is smaller than `q/2` and everything after it is also smaller than `q/2` and that that closes our goal. To do this, we first change `q` into `q/2 + q/2`:

```
have hqa : (q / 2) + (q / 2) = q, by
{
  rw ← mul_two, -- q/2 + q/2 = (q/2)*2
  rw div_mul_cancel, -- (q/2)*2 = q
  exact two_ne_zero, -- proving we are not dividing by 0
},
rw ← hqa,
```

We can then use the lemma `add_lt_add`, which states:

```
lemma add_lt_add {a b c d : α} (h₁ : a < b) (h₂ : c < d) :
  a + c < b + d
```

Using the tactic `apply add_lt_add` gives us two new goals (which are $h_1$ and $h_2$ in the lemma):

```
⊢ (seq x (max xn yn)).val.snd − (seq x (max xn yn)).val.fst < q / 2
⊢ (seq y (max xn yn)).val.snd − (seq y (max xn yn)).val.fst < q / 2
```

The only difference in these two goals is that the second one is about `y` instead of `x`. The proofs are as such also completely analogous to each other. We will only show the proof for `x`. The first step is to prove the following:

```
have hm : segment.snd (seq y (max xn yn)) −
  segment.fst (seq y (max xn yn)) ≤ (segment.snd (y.val yn) −
  segment.fst (y.val yn)),
```

Proving this can be done by using two statements we have introduced previously, along with the lemma `le_max_left` from the standard library. Completing the proof then only takes one more lemma from the standard library.

```
lemma contained_bounds_le (s t : 𝕊) (h : s ⊑ t) :
  s.snd − s.fst ≤ t.snd − t.fst
theorem contained_of_le (r : ℛ) {n m : ℕ} (h₁ : n ≤ m) :
  r.seq m ⊑ r.seq n
lemma le_max_left (a b : α) : a ≤ max a b
lemma lt_of_le_of_lt [preorder α] : ∀ {a b c : α},
  a ≤ b → b < c → a < c
```

The full proof for `x` is then:

```
have hm : segment.snd (seq x (max xn yn)) −
   segment.fst (seq x (max xn yn)) ≤ (segment.snd (x.val xn) −
   segment.fst (x.val xn)), by
{
   apply segment.contained_bounds_le,
   apply contained_of_le _ (le_max_left _ _),
},
exact lt_of_le_of_lt hm hxn,
```

So now we know that the definition of addition really has another real number as its output.

We have also proven some elementary statements about addition. The proofs of these can be almost completely handled by `simp` because they consist mostly of writing out the full definitions of everything. These statements are:

```
theorem add_assoc {x y z : ℝ} :
   add (add x y) z =' add x (add y z)
theorem add_comm {x y : ℝ} : add x y =' add y x
theorem add_zero {x : ℝ} : add x 0 =' x
theorem zero_add {x : ℝ} : add 0 x =' x
theorem eq_implies_add_eq_add {x y z : ℝ} :
   y =' z → add x y =' add x z
```

The definition of subtraction is now addition with the additive inverse of the second input. These are defined as follows:

```
def neg (x : ℝ) : ℝ := subtype.mk (λ n, (x.val n).neg)
def sub (x y : ℝ) : ℝ := add x (neg y)
```

We do not need to prove that `sub` results in an element of $\mathcal{R}$, since `add` already provides us with that information. We do need to prove it for `neg`, but the proof is easier than the one for `add` so we will omit it here. We have also proven some statements about `sub`, which we will state here without proof:

```
theorem sub_self_eq_zero (x : ℝ) : sub x x =' 0
theorem sub_add (x y : ℝ) : sub (add x y) y =' x
theorem sub_add_comm {x y z : ℝ} :
   sub (add x y) z =' add x (sub y z)
```

This last theorem actually has an very short proof: `exact add_assoc`.

## 3.5 Spreads and Fans

The fan theorem is an axiom that was used by Brouwer to prove that all functions $\mathcal{R} \to \mathcal{R}$ are continuous. We will not do this here, but we will define what the fan theorem is.

Fans are a special kinds of spreads. We will therefore first define what a spread is. A spread is a set of elements of $\mathcal{N}$ that satisfies a spread law. A spread law is a function $\sigma$ : `fin_seq` $\to \mathbb{N}$ which indicates which sequences are "accepted" and which are not. We say that $\sigma$ accepts the element $a$ of `fin_seq` if $\sigma(a) = 0$. A spread law also has to satisfy two properties:

1. $\sigma(\langle\rangle) = 0$
2. $\sigma(a) = 0 \iff$ there is an extension of $a$ that is also accepted by $\sigma$

In Lean, we formalized this as follows:

```
def is_spread_law (σ : fin_seq → ℕ) : Prop :=
  σ empty_seq = 0 ∧
  (∀ s : fin_seq, σ s = 0 ↔
  ∃ n : ℕ, σ (extend s (singleton n)) = 0)
```

So every element of a spread has at least one extension, and possibly infinitely many. Imagine we keep extending $\langle\rangle$ for an infinite number of times. We would get an element of $\mathcal{N}$. This element would also satisfy that truncating it at any point would yield a finite sequence that is accepted by $\sigma$. The spread defined by $\sigma$ consists of precisely those elements. In Lean, we use subtypes for this:

```
def spread (σ : fin_seq → ℕ) (hσ : is_spread_law σ) : Type :=
  {a : 𝒩 // ∀ n : ℕ, σ (finitize a n) = 0}
```

A fan law is a spread law that satisfies one additional property: Each finite sequence that is accepted only has a finite number of accepted extensions. Fans are then defined exactly like spreads, but with a fan law instead of a spread law.

```
def is_fan_law (β : fin_seq → ℕ) : Prop :=
  is_spread_law β ∧
  (∀ s : fin_seq, (β s = 0 →
  ∃ n : ℕ, ∀ m : ℕ, β (extend s (singleton m)) = 0 → m ≤ n))
def fan (β : fin_seq → ℕ) (hβ : is_fan_law β) : Type :=
  {a : 𝒩 // ∀ n : ℕ, β (finitize a n) = 0}
```

The fan theorem is about bars in fans. A bar could be seen as a kind of wall in a fan. Every element of a fan is made by walking along a path of finite sequences that are all accepted by the fan law. A bar is then a set of finite sequences such that every element of the fan has a finite sequence that is an element of the bar.

```
def is_bar (β : fin_seq → ℕ) (hβ : is_fan_law β)
  (B : set fin_seq) : Prop :=
  ∀ a : fan β hβ, ∃ n : ℕ, finitize a.val n ∈ B
```

The fan theorem has multiple different phrasings that are all equivalent. One would be that every bar has a subset that is still a bar. We formalized another version in Lean:

```
def fan_theorem (β : fin_seq → ℕ) (hβ : is_fan_law β)
  (B : set fin_seq) (hB : is_bar β hβ B) : Prop :=
  ∃ m : ℕ, ∀ a : fan β hβ, ∃ n : ℕ,
  n ≤ m ∧ finitize a.val n ∈ B
```

That is: If B is a bar for a fan, then every element of the fan will run into the bar before a fixed upper bound `m`.

# Chapter 4

# Related Work

H. Bergwerf made a Coq formalization of intuitionism, based on the same lecture notes by W. Veldman. Besides being in two different languages, our formalization has a more complete notion of a statement being reckless, while Bergwerf formalized a lot of other concepts that we did not include in our formalization, like Kleene's T-predicate and the different versions of the axiom of choice.[3]

K. Buzzard runs the Xena Project, which aims to formalize the entire pure math undergraduate curriculum of the Imperial College London in Lean. The actual projects are usually done by undergraduate students whom are guided and supervised by Buzzard.[4] Among many others, examples of math that has been formalized in the Xena Project include:

- Schemes, by R. F. Mir
- Group cohomology, by A. Ciobanu
- Euclidean geometry, by A. Sever
- Combinatorics, by B. Mehta

There have been a decent amount of papers written about formalizations in Lean.[7] We will mention two of them. The full list can be found on the Lean Community website.

J. M. Han and F. van Doorn wrote a formal proof on the independence of the continuum hypothesis in ZFC (Zermelo-Fraenkel set theory with the axiom of choice), which they called the Flypitch project. Their formalization established that CH is consistent in ZFC, and that ¬CH is also consistent in ZFC.[11]

K. Buzzard, J. Commelin and P. Massot wrote a paper about the formalization of the definition of perfectoid spaces in Lean. Perfectoid spaces are considered modern by mathematical standards. As such, this formalization put Lean on the map as a serious theorem prover.[6]

# Chapter 5

# Conclusions

We formalized a significant part of the lecture notes for the course Intuitionistic Mathematics by W. Veldman. Lean proved to be a good tool for this. Though we did not expect there to be major problems, it is good to confirm that Lean handled intuitionistic principles such as Brouwer's Continuity Principle well.

In the process of formalizing, we encountered some minor problems. Most of these are due to how mathematicians do math with pencil and paper, though there were also some problems that originated from Lean or from the way we formalized different notions. The problems we encountered were:

- Coercions and casting between natural numbers, elements of `fin` and rational numbers. Sometimes we would encounter the need to prove statements like `↑n = n`, which proved difficult enough that we needed to rewrite major portions of proofs to avoid running into this problem. This is a well-known problem in Lean that is actively being worked on.[12]

- Equality of two elements of `fin_seq` was near impossible to prove. This is due to two sequences of type `fin n` $\to \mathbb{N}$ and `fin m` $\to \mathbb{N}$ having two different types, even when `n = m`. This is mainly a problem with our formalization.

- Missing proofs. Mathematicians often leave proofs as an exercise to the reader, or state that the proofs are well-known and that they will not go into further detail. The problem that arises is then that no modern literature contains proofs of certain statements. As such, we wrote most proofs ourselves, instead of adapting established proofs from literature. This is not really a problem, but it is annoying.

- Computers think in steps, mathematicians think in results. In a Lean proof, you will need to provide what you are doing and what lemmas and theorems you are using. Mathematicians often think that the

reader will be able to figure out themselves what is being done and only show the results (the proof state) that follows from these actions. For an example of this, see the proof that $<$ is transitive on $\mathcal{N}$ in section 3.1.

## 5.1 Acknowledgments

I want to thank Freek Wiedijk for supervising this thesis. He always was available to provide feedback and point me towards interesting pieces of mathematics I had never heard of before, even though there were some difficulties with corona, the summer break and my horrible planning.

I also want to thank Wim Veldman for being a great teacher for intuitionism and logic in general. His course inspired me to consider this topic for my thesis, which turned out to be a lot of fun.

Special thanks to Reid Barton, Patrick Massot and all other members of the Lean Zulip chat for answering questions about Lean. Though most of my questions were already asked previously, so I did not have to ask them again, reading through the answers helped me a lot.

The natural number game by Kevin Buzzard and Mohammad Pedramfar introduced me to Lean and helped me understand how writing proofs in Lean tactic mode works. It is a great 'game' for learning Lean, you should consider playing it if you have not done so already.

Lastly, I want to thank Jeremy Avigad for making the `lstlean.tex` file, which makes Lean code in LaTeX much more readable (and it looks much better too, in my opinion).

# Bibliography


[1] Jeremy Avigad, Leonardo de Moura, , and Soonho Kong. Theorem proving in lean, 2017. https://leanprover.github.io/theorem_proving_in_lean/.

[2] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, and Johannes Hölzl. The hitchhiker's guide to logical verification, 2020. Lecture notes for the MSc-level course Logical Verfication at the VU Amsterdam. https://lean-forward.github.io/logical-verification/2020/index.html.

[3] Herman Bergwerf. Intuinistic principles in coq, 2020. https://github.com/bergwerf/intuitionism.

[4] Keven Buzzard. Xena project, 2018-2020. http://wwwf.imperial.ac.uk/~buzzard/xena/.

[5] Keven Buzzard and Mohammad Pedramfar. Natural number game, 2020. https://wwwf.imperial.ac.uk/~buzzard/xena/natural_number_game/.

[6] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 299–312, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Leanprover Community. Papers. https://leanprover-community.github.io/papers.html.

[8] The Lean Community. Lean zulip chat. https://leanprover.zulipchat.com/.

[9] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[10] Hannes Diener. Constructive reverse mathematics, 2018. Habilitationsschrift.

[11] Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 353–366, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Robert Y. Lewis and Paul-Nicolas Madelaine. Simplifying casts and coercions, 2020.

[13] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[14] A. S. Troelstra. *History of constructivism in the 20th century*, page 150–179. Lecture Notes in Logic. Cambridge University Press, 2011.

[15] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, Vol 1*. ISSN. Elsevier Science, 1988.

[16] Wim Veldman. Notes on intuitionistic mathematics, 2019. Lecture notes for the course "Intuitionistische Wiskunde" at the Radboud University.

# Appendix A

# Appendix

## A.1 Index of all theorems

All these statements are proven. Proofs omitted for brevity. To see the full proofs, look at the full code: `https://github.com/SCRK16/Intuitionism`

### A.1.1 nat_seq.lean

```
def nat_seq := ℕ → ℕ
notation ‘𝒩‘ := nat_seq
def zero : 𝒩 := λ n, 0
def eq (a b : 𝒩) : Prop := ∀ n : ℕ, a n = b n
infix ‘=’‘:50 := eq
lemma eq_iff {a b : 𝒩} : a = b ↔ a =’ b
def ne (a b : 𝒩) : Prop := ¬ a =’ b
infix ‘≠’‘:50 := ne
def lt (a b : 𝒩) : Prop :=
  ∃ n : ℕ, (∀ i : ℕ, i < n → a i = b i) ∧ a n < b n
infix ‘<‘ := lt
def le (a b : 𝒩) : Prop :=
  ∀ n : ℕ, (∀ i : ℕ, i < n → a i = b i) → a n ≤ b n
infix ‘≤‘ := le
theorem le_of_eq (a b : 𝒩) (h : a =’ b) : a ≤ b
lemma imp_eq_iff_imp_eq (a b : 𝒩) (n : ℕ) :
  (∀ i : ℕ, i < n → a i = b i) ↔ (∀ i : ℕ, i < n → b i = a i)
lemma imp_eq_trans (a b c : 𝒩) (n : ℕ)
  (h₁ : ∀ i : ℕ, i < n → a i = b i)
  (h₂ : ∀ i : ℕ, i < n → b i = c i) :
  ∀ i : ℕ, i < n → a i = c i
theorem eq_trans (a b c : 𝒩) : a =’ b → b =’ c → a =’ c
theorem eq_symm (a b : 𝒩) : a =’ b ↔ b =’ a
```

```
theorem eq_refl (a : 𝒩) : a =' a
theorem ne_symm (a b : 𝒩) : a ≠' b ↔ b ≠' a
lemma lt_eq_lt_le (a b : 𝒩) (n m : ℕ)
  (h1 : ∀ i : ℕ, i < n → a i = b i) (h2 : a m < b m) :
  n ≤ m
lemma lt_eq_ne_le (a b : 𝒩) (n m : ℕ)
  (h1 : ∀ i : ℕ, i < n → a i = b i) (h2 : a m ≠ b m) :
  n ≤ m
lemma first_zero_eq (a : 𝒩) (n m : ℕ)
  (hn1 : ∀ i : ℕ, i < n → a i = 0) (hn2 : a n ≠ 0)
  (hm1 : ∀ i : ℕ, i < m → a i = 0) (hm2 : a m ≠ 0) :
  n = m
theorem le_of_lt (a b : 𝒩) (less: a < b) : a ≤ b
theorem lt_trans (a b c : 𝒩) : a < b → b < c → a < c
lemma all_eq_or_exists_neq (a b : 𝒩) (n : ℕ) :
  (∀ i : ℕ, i < n → a i = b i) ∨
  (∃ i : ℕ, i < n ∧ (∀ j : ℕ, j < i → a j = b j) ∧ a i ≠ b i)
lemma nat_lt_cotrans (a b : ℕ) (h : a < b) :
  ∀ c : ℕ, a < c ∨ c < b
theorem lt_cotrans (a b : 𝒩) (h : a < b) :
  ∀ c : 𝒩, a < c ∨ c < b
theorem le_iff_not_lt (a b : 𝒩) : a ≤ b ↔ ¬ b < a
theorem le_trans (a b c : 𝒩) : a ≤ b → b ≤ c → a ≤ c
theorem le_stable (a b : 𝒩) : ¬¬a ≤ b → a ≤ b
theorem eq_of_le_le {a b : 𝒩} (hab : a ≤ b) (hba : b ≤ a) :
  a =' b
def apart (a b : 𝒩) : Prop := ∃ n, a n ≠ b n
infix `#` := apart
theorem ne_of_apart (a b : 𝒩) (r : a # b) : a ≠' b
theorem eq_iff_not_apart (a b : 𝒩) : a =' b ↔ ¬ a # b
theorem eq_stable (a b : 𝒩) : ¬¬ a =' b → a =' b
theorem apart_iff_lt_or_lt (a b : 𝒩) : a # b ↔ a < b ∨ b < a
theorem apart_cotrans (a b : 𝒩) (h : a # b) :
  ∀ c : 𝒩, a # c ∨ c # b
theorem apart_symm (a b : 𝒩) : a # b ↔ b # a
lemma zero_le (a : 𝒩) : zero ≤ a
lemma apart_zero_lt (a : 𝒩) (h : a # zero) : zero < a
theorem uncountable (f : ℕ → 𝒩) :
  ∃ a : 𝒩, ∀ n : ℕ, a # (f n)
```

### A.1.2 reckless.lean

This whole file has the assumption: `variables P Q : Prop`

```
def PO : Prop := ∀ Q : Prop, Q ∨ ¬Q
```

```
def LPO : Prop := ∀ a : 𝒩,
  (∀ n : ℕ, a n = 0) ∨ (∃ n : ℕ, a n ≠ 0)
def reckless_LPO : Prop → Prop := λ P : Prop,
  (PO → P) ∧ (P → LPO)
def LLPO : Prop := ∀ a : 𝒩,
  (∀ k : ℕ, (∀ i : ℕ, i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 0)
  ∨ (∀ k : ℕ, (∀ i : ℕ, i < k → a i = 0) ∧ a k ≠ 0 → k % 2 = 1)
def reckless_LLPO : Prop → Prop := λ P : Prop,
  (PO → P) ∧ (P → LLPO)
theorem PO_implies_LPO : PO → LPO
lemma exists_reckless : ∃ P : Prop, reckless_LPO P
theorem LPO_implies_LLPO : LPO → LLPO
theorem reckless_not_not_implies :
  reckless_LPO (∀ P : Prop, ¬¬P → P)
theorem reckless_implies_not_or :
  reckless_LPO (∀ P Q : Prop, (P → Q) → (Q ∨ ¬P))
theorem reckless_LPO_le_implies_lt_or_eq :
  reckless_LPO (∀ a b : 𝒩, a ≤ b → a < b ∨ a =' b)
theorem implies_not_implies_not :
  ∀ P Q : Prop, (P ∨ ¬P → ¬Q) → ¬Q
theorem reckless_LPO_implies_implies :
  reckless_LPO (∀ P Q : Prop, (P ∨ ¬P → Q) → Q)
instance start_le_not_zero_decidable (a : 𝒩) (n : ℕ) :
  decidable (∃ i : ℕ, i ≤ n ∧ a i ≠ 0)
instance start_lt_not_zero_decidable (a : 𝒩) (n : ℕ) :
  decidable (∃ i : ℕ, i < n ∧ a i ≠ 0)
def snap (a : 𝒩) : ℛ := subtype.mk (λ n : ℕ,
  if h : (∃ i : ℕ, i ≤ n ∧ a i ≠ 0)
  then segment.inclusion (1 / nat.succ (nat.find h))
  else segment.two_sided_inclusion (1 / nat.succ n)
theorem reckless_LPO_real_lt_eq_gt :
  reckless_LPO (∀ x y : ℛ, x < y ∨ x =' y ∨ y < x)
def WLEM : Prop := ∀ P : Prop, ¬P ∨ ¬¬P
def WLPO : Prop := ∀ a : 𝒩, (∀ n : ℕ, a n = 0) ∨ (¬∀ n : ℕ, a n = 0)
theorem PO_implies_WLEM : PO → WLEM
theorem LPO_implies_WLPO : LPO → WLPO
theorem weak_LEM_implies_weak_LPO : WLEM → WLPO
theorem weak_LPO_implies_LLPO : WLPO → LLPO
theorem weak_LEM_implies_LLPO : WLEM → LLPO
theorem reckless_LLPO_not_not_or :
  reckless_LLPO (∀ P Q : Prop, ¬¬(P ∨ Q) → (¬¬P ∨ ¬¬Q))
theorem reckless_LLPO_not_and_implies_not_or_not :
  reckless_LLPO (∀ P Q : Prop, ¬(P ∧ Q) → (¬P ∨ ¬Q))
lemma or_not_implies_not_not_implies (h : P ∨ ¬P) : ¬¬P → P
```

```
theorem reckless_LLPO_not_not_implies_or :
  reckless_LLPO (∀ P : Prop, (¬¬P → P) → P ∨ ¬P)
example : (∀ P : Prop, ¬¬P → P) → (∀ P : Prop, P ∨ ¬P)
```

### A.1.3 bcp.lean

```
def BCP : Prop := ∀ R : 𝒩 → ℕ → Prop,
  (∀ a : 𝒩, ∃ n : ℕ, R a n) →
  ∀ a : 𝒩, ∃ m n : ℕ, ∀ b : 𝒩,
  (∀ i : ℕ, i < m → a i = b i) → R b n
lemma exists_start_eq_ne (a : 𝒩) (n : ℕ) :
  ∃ b : 𝒩, (∀ i : ℕ, i < n → a i = b i) ∧ a n ≠ b n
theorem strongly_not_injective (f : 𝒩 → ℕ) :
  BCP → ∀ a : 𝒩, ∃ b : 𝒩, a # b ∧ f a = f b
theorem not_injective (f : 𝒩 → ℕ) :
  BCP → ¬(∀ a b : 𝒩, f a = f b → a =' b)
lemma grow_tail (a b : 𝒩) (n : ℕ)
  (h₁ : ∀ i : ℕ, i >= nat.succ(n) → a i = b i)
  (h₂ : a n = b n) : ∀ i : ℕ, i >= n → a i = b i
lemma tail_equal_not_forall_equal_implies_exists_ne (a b : 𝒩)
  (n : ℕ) (h₁ : ∀ i : ℕ, i >= n → a i = b i)
  (h₂ : ¬ ∀ i : ℕ, a i = b i) : ∃ i : ℕ, i < n ∧ a i ≠ b i
lemma ite_cond_eq (a b d : 𝒩) (n : ℕ)
  (hd : d =' (λ i, ite (i < n) (b i) (a i))) :
  ∀ (i : ℕ), i ≥ n → d i = a i
theorem apart_iff_forall_ne_or_ne (bcp : BCP) (a b : 𝒩) :
  a # b ↔ ∀ c : 𝒩, a ≠' c ∨ c ≠' b
theorem BCP_implies_not_LPO : BCP → ¬reckless.LPO
theorem BCP_implies_not_WLPO : BCP → ¬reckless.WLPO
```

### A.1.4 segment.lean

```
def segment := {s : ℚ × ℚ // s.fst ≤ s.snd}
notation `𝕊` := segment
def fst (s : 𝕊) : ℚ := (subtype.val s).fst
def snd (s : 𝕊) : ℚ := (subtype.val s).snd
def proper (s : 𝕊) : Prop := s.fst < s.snd
def contained (s t : 𝕊) : Prop := t.fst ≤ s.fst ∧ s.snd ≤ t.snd
infix ⊑``:50 := contained
def proper_contained (s t : 𝕊) : Prop :=
   t.fst < s.fst ∧ s.snd < t.snd
infix ⊏``:50 := proper_contained
def lt (s t : 𝕊) : Prop := s.snd < t.fst
infix `<` := lt
```

```
def le (s t : 𝕊) : Prop := s.fst ≤ t.snd
infix `≤` := le
def inclusion (q : ℚ) : 𝕊 := subtype.mk (q, q)
def has_zero : has_zero 𝕊 := { zero := inclusion 0 }
def two_sided_inclusion (q : ℚ) (hq : q > 0) : 𝕊 :=
    subtype.mk (−q, q)
lemma two_sided_inclusion_contained {q₁ q₂ : ℚ} {hq₁ : q₁ > 0}
    {hq₂ : q₂ > 0} (h : q₁ ≤ q₂) :
    two_sided_inclusion q₁ hq₁ ⊑ two_sided_inclusion q₂ hq₂
theorem contained_trans (s t v : 𝕊)
    (h₁ : s ⊑ t) (h₂ : t ⊑ v) : s ⊑ v
theorem proper_contained_trans (s t v : 𝕊)
    (h₁ : s ⊏ t) (h₂ : t ⊏ v) : s ⊏ v
theorem contained_refl (s : 𝕊) : s ⊑ s
lemma le_iff_not_lt (s t : 𝕊) : s ≤ t ↔ ¬ t < s
lemma lt_iff_not_le (s t : 𝕊) : s < t ↔ ¬ t ≤ s
theorem lt_trans (s t v : 𝕊) (h₁ : s < t) (h₂ : t < v) : s < v
theorem le_refl (s : 𝕊) : s ≤ s
def touches (s t : 𝕊) : Prop := s ≤ t ∧ t ≤ s
infix `≈` := touches
theorem touches_refl (s : 𝕊) : s ≈ s
theorem touches_symm (s t : 𝕊) : s ≈ t ↔ t ≈ s
def add (s t : 𝕊) : 𝕊 := subtype.mk (s.fst + t.fst, s.snd + t.snd)
theorem add_assoc (s t v : 𝕊) : add (add s t) v = add s (add t v)
lemma fst_add_comm {s t : 𝕊} : fst (add s t) = fst s + fst t
lemma snd_add_comm {s t : 𝕊} : snd (add s t) = snd s + snd t
theorem add_comm (s t : 𝕊) : add s t = add t s
lemma contained_bounds_le (s t : 𝕊) (h : s ⊑ t) :
    s.snd − s.fst ≤ t.snd − t.fst
instance : add_comm_semigroup 𝕊
def neg (s : 𝕊) : 𝕊 := subtype.mk (−s.snd, −s.fst)
```

### A.1.5 real.lean

```
def shrinking (r : ℕ → 𝕊) := ∀ n, r (n + 1) ⊑ r n
def dwindling (r : ℕ → 𝕊) := ∀ q : ℚ, q > 0 →
   ∃ n : ℕ, (r n).snd − (r n).fst < q
def real_seq := {r : ℕ → 𝕊 // shrinking r ∧ dwindling r}
notation ℛ`` := real_seq
def seq (r : ℛ) : ℕ → 𝕊 := subtype.val r
def fst (r : ℛ) : ℕ → ℚ := λ n, (r.seq n).fst
def snd (r : ℛ) : ℕ → ℚ := λ n, (r.seq n).snd
lemma shrinking (r : ℛ) : shrinking r.val
lemma dwindling (r : ℛ) : dwindling r.val
```

```
theorem contained_of_le (r : ℝ) {n m : ℕ} (h₁ : n ≤ m) :
  r.seq m ⊑ r.seq n
def lt (x y : ℝ) : Prop := ∃ n : ℕ, x.seq n < y.seq n
infix `<` := lt
def le (x y : ℝ) : Prop := ∀ n : ℕ, x.seq n ≤ y.seq n
infix `≤` := le
def apart (x y : ℝ) : Prop := x < y ∨ y < x
infix `#` := apart
def eq (x y : ℝ) : Prop := ∀ n : ℕ, x.seq n ≈ y.seq n
infix `=''`:50 := eq
def ne (x y : ℝ) : Prop := ¬ x =' y
infix `≠''`:50 := ne
theorem lt_trans (x y z : ℝ) (h₁ : x < y) (h₂ : y < z) : x < z
lemma lt_or_lt_from_sub_lt_sub {a b c d : ℚ}
  (h : a − b < c − d) : a < c ∨ d < b
theorem lt_cotrans (x y z : ℝ) (h₁ : x < y) : x < z ∨ z < y
theorem apart_cotrans (x y z : ℝ) (h : x # y) : x # z ∨ z # y
theorem le_iff_not_lt (x y : ℝ) : x ≤ y ↔ ¬ y < x
theorem eq_of_le_of_le (x y : ℝ) : x ≤ y → y ≤ x → x =' y
theorem le_trans (x y z : ℝ) (h₁ : x ≤ y) (h₂ : y ≤ z) : x ≤ z
theorem le_refl (x : ℝ) : x ≤ x
theorem eq_iff_not_apart (x y : ℝ) : x =' y ↔ ¬x # y
theorem eq_trans (x y z: ℝ) (h₁ : x =' y) (h₂ : y =' z) : x =' z
theorem eq_symm (x y : ℝ) : x =' y ↔ y =' x
theorem eq_refl (x : ℝ) : x =' x
theorem le_stable (x y : ℝ) : ¬¬x ≤ y → x ≤ y
theorem eq_stable (x y : ℝ) : ¬¬x =' y → x =' y
def inclusion_const (q : ℚ) : ℝ :=
  subtype.mk (λ _ , segment.inclusion q)
def has_zero : has_zero ℝ := { zero := inclusion_const 0 }
lemma zero : 0 = inclusion_const 0
def add (x y : ℝ) : ℝ :=
  subtype.mk (λ n, segment.add (x.seq n) (y.seq n))
theorem add_assoc {x y z : ℝ} :
  add (add x y) z =' add x (add y z)
theorem add_comm {x y : ℝ} : add x y =' add y x
theorem add_zero {x : ℝ} : add x 0 =' x
theorem zero_add {x : ℝ} : add 0 x =' x
theorem eq_implies_add_eq_add {x y z : ℝ} :
  y =' z → add x y =' add x z
def neg (x : ℝ) : ℝ := subtype.mk (λ n, (x.val n).neg)
def sub (x y : ℝ) : ℝ := add x (neg y)
theorem sub_self_eq_zero (x : ℝ) : sub x x =' 0
theorem forall_exists_additive_inverse :
```

```
  ∀ x : ℛ, ∃ y : ℛ, add x y =' 0
theorem sub_add (x y : ℛ) : sub (add x y) y =' x
theorem sub_add_comm {x y z : ℛ} :
  sub (add x y) z =' add x (sub y z)
```

### A.1.6 fin_seq.lean

```
structure fin_seq := mk :: (len : ℕ) (seq : fin len → ℕ)
def finitize (a : 𝒩) (n : ℕ) : fin_seq := {
  len := n,
  seq := λ i, a i.val
}
lemma finitize_len (a : 𝒩) (n : ℕ) : (finitize a n).len = n
def is_initial_of (a : fin_seq) (b : 𝒩) : Prop :=
  ∀ i : fin a.len, a.seq i = b i
infix ⊑``:50 := is_initial_of
lemma is_initial_of_self (a : 𝒩) {n : ℕ} : (finitize a n) ⊑ a
def shorten (a : fin_seq) (m : ℕ) (h : m ≤ a.len) : fin_seq := {
  len := m,
  seq := λ i, a.seq (fin.cast_le h i)
}
def extend (a b : fin_seq) : fin_seq := {
  len := a.len + b.len,
  seq := λ i, if h : i.val < a.len
    then a.seq (fin.cast_le h i)
    else b.seq (fin.sub_nat a.len
    (fin.cast (add_comm a.len b.len) i))
}
def extend_inf (a : fin_seq) (b : 𝒩) : 𝒩 :=
  λ i, if h : i < a.len
    then a.seq (fin.cast_le h i)
    else b (i − a.len)
lemma extend_inf_eq {a : fin_seq} {b₁ b₂ : 𝒩} (h: b₁ =' b₂) :
  extend_inf a b₁ =' extend_inf a b₂
lemma eq_extend_inf {a₁ a₂ : fin_seq} {b : 𝒩}
  (h₁ : a₁.len = a₂.len)
  (h₂ : ∀ i, a₁.seq i = a₂.seq (fin.cast h₁ i)) :
  extend_inf a₁ b =' extend_inf a₂ b
def empty_seq : fin_seq := {
  len := 0,
  seq := λ i, 0
}
lemma empty_seq_eq {a : fin_seq} (ha : a.len = 0) :
  ∀ i, empty_seq.seq i = a.seq (fin.cast _ i)
```

```
lemma empty_extend_eq_self (a : 𝒩) : extend_inf empty_seq a ='
    a
def singleton (n : ℕ) : fin_seq := {
   len := 1,
   seq := λ i, n
}
theorem finitize_initial_iff_start_eq (a b : 𝒩) (n : ℕ) :
   finitize a n ⊑ b ↔ (∀ j : ℕ, j < n → a j = b j)
theorem finitize_eq_iff_start_eq (a b : 𝒩) (n : ℕ) :
   finitize a n = finitize b n ↔ (∀ j : ℕ, j < n → a j = b j)
lemma finitize_initial_iff_finitize_eq (a b : 𝒩) (n : ℕ) :
   finitize a n ⊑ b ↔ finitize a n = finitize b n
def tail (a : fin_seq) : fin_seq := {
   len := a.len − 1,
   seq := λ i, a.seq (fin.cast_le _ i)
}
lemma tail_singleton_len_zero :
   ∀ n : ℕ, (tail (singleton n)).len = 0
def len_seq (n : ℕ) : Type := fin n → ℕ
def to_fin_seq {n : ℕ} : len_seq n → fin_seq := λ f, {
   len := n,
   seq := f,
}
lemma fin_len_eq {n : ℕ} {a : len_seq n} : (to_fin_seq a).seq = a
lemma len_fin_eq (a : fin_seq) : (to_fin_seq a.seq) = a
lemma len_seq_0_unique (x y : len_seq 0) : x = y
```

### A.1.7 coding_sequences.lean

```
lemma len_seq_1_equiv_nat : len_seq 1 ≃ ℕ
lemma len_seq_0_equiv_punit : len_seq 0 ≃ punit
lemma len_seq_succ_equiv_len_seq (n : ℕ) :
   len_seq (nat.succ (nat.succ n)) ≃ len_seq (nat.succ n)
theorem len_seq_equiv_nat (n : ℕ) : len_seq (nat.succ n) ≃ ℕ
theorem len_seq_equiv_nat' (n : ℕ) (h : ¬n = 0) : len_seq n ≃ ℕ
```

### A.1.8 fan.lean

```
def is_spread_law (σ : fin_seq → ℕ) : Prop :=
   σ empty_seq = 0 ∧ (∀ s : fin_seq, σ s = 0 ↔
   ∃ n : ℕ, σ (extend s (singleton n)) = 0)
def is_fan_law (β : fin_seq → ℕ) : Prop :=
   is_spread_law β ∧ (∀ s : fin_seq, (β s = 0 →
   ∃ n : ℕ, ∀ m : ℕ, β (extend s (singleton m)) = 0 → m ≤ n))
lemma fan_law_is_spread_law (β : fin_seq → ℕ)
```

```
  (hβ : is_fan_law β) : is_spread_law β
def spread (σ : fin_seq → ℕ) (hσ : is_spread_law σ) : Type :=
  {a : 𝒩 // ∀ n : ℕ, σ (finitize a n) = 0}
def fan (β : fin_seq → ℕ) (hβ : is_fan_law β) : Type :=
  {a : 𝒩 // ∀ n : ℕ, β (finitize a n) = 0}
lemma fan_is_spread (β : fin_seq → ℕ) (hβ : is_fan_law β) :
  fan β hβ = spread β (fan_law_is_spread_law β hβ)
def fan_theorem (β : fin_seq → ℕ) (hβ : is_fan_law β)
  (B : set fin_seq) (hB : is_bar β hβ B) : Prop := ∃ m : ℕ,
  ∀ a : fan β hβ, ∃ n : ℕ, n ≤ m ∧ finitize a.val n ∈ B
```

## A.2 Overview of common Lean tactics

This section will give a quick overview of tactics that are commonly used in Lean proofs. The general structure of this section will be:

1. Proof state before using the tactic
2. Tactic
3. Proof state after using the tactic

For tactics that can be used in multiple scenarios (like `cases`), all common scenarios will be listed. Tactics like `simp` and `library_search` will not be included in this list, because their behaviour is too complicated to fit nicely in this format.

Proof states shown here will be simplified. Types of elements may be omitted. In most cases, the type does not matter for the functionality of the tactic. Tactics are listed in alphabetical order.

### A.2.1 apply

```
h : P → Q
⊢ Q
```

```
apply h,
```

```
⊢ P
```

### A.2.2 cases

1.

```
h : P ∧ Q
⊢ R
```

```
cases h with hp hq,
```

```
hp : P,
hq : Q,
⊢ R
```

2.

```
h : P ∨ Q
⊢ R
```

```
cases h with hp hq,
```

```
2 goals:
hp : P,
⊢ R
```

```
hq : Q,
⊢ R
```

3.

```
h : ∃ n : ℕ, succ n = 2
⊢ P
```

```
cases h with n hn,
```

```
n : ℕ,
hn : succ n = 2
⊢ P
```

### A.2.3 exact

```
h : P,
⊢ P
```

```
exact h,
```

```
goals accomplished
```

### A.2.4 exfalso

```
⊢ P
```

```
exfalso,
```

```
⊢ false
```

### A.2.5 have

```
⊢ P
```

```
have h : Q,
```

```
2 goals:
⊢ Q
hq : Q

⊢ P
```

### A.2.6 induction

```
n : ℕ,
⊢ 1 + n = succ n
```

```
induction n with d hd,
```

```
2 goals:
⊢ 1 + 0 = succ 0

hd : 1 + d = succ d
⊢ 1 + succ d = succ (succ d)
```

### A.2.7 intro

1.
```
⊢ P → Q
```

```
intro hp,
```

```
hp : P,
⊢ Q
```

2.
```
P : A → Prop,
⊢ ∀ a :  A, P a,
```

```
intro a,
```

```
P : A → Prop,
a : A,
⊢ P a,
```

Note: The tactic `intros` behaves exactly like `intro`, except that `intros` can introduce multiple assumptions in one line. `intros x y z` can always be replaced by `intro x`, `intro y`, `intro z` and vice versa.

### A.2.8 left/right

1.
```
⊢ P ∨ Q
```

```
left,
```

```
⊢ P
```

2.
```
⊢ P ∨ Q
```

```
right,
```

```
⊢ Q
```

### A.2.9 refl

```
⊢ a = a
```

```
   refl,
goals accomplished
```

### A.2.10 revert

```
hp : P
⊢ Q
```

```
   revert hp,
```

```
⊢ P → Q
```

### A.2.11 rw

1.

```
h : a = b
⊢ P a
```

```
   rw h,
```

```
h : a = b
⊢ P b
```

2.

```
h : P ↔ Q
⊢ P
```

```
   rw h,
```

```
⊢ Q
```

Note: The tactic `rwa` does the same as `rw`, but `rwa` can be given a list of assumptions to rewrite with instead of only a single one. So: `rw h`, `rw g` can be done in one line with `rwa [h, g]`.

In our two examples we used `rw` to find what is on the left hand side and replace it with what is on the right hand side. We can also use `rw` to rewrite from right to left if that is desirable. See the following example:

```
h : a = b
⊢ P b
```

```
   rw ← h,
```

```
h : a = b
⊢ P a
```

### A.2.12 set

```
⊢ P
```

```
set n : ℕ := 1 with hn,
```

```
n : ℕ,
hn : n = 1,
⊢ P
```

### A.2.13 split

```
⊢ P ∧ Q
```

```
split,
```

```
2 goals:
⊢ P
⊢ Q
```

Note: `split` can also be used on `P ↔ Q`, since `P ↔ Q = (P → Q ∧ Q → P)`.

### A.2.14 suffices

```
⊢ P
```

```
suffices h : Q,
```

```
2 goals:
hq : Q,
⊢ P

⊢ Q
```

### A.2.15 use

```
⊢ ∃ n : ℕ, succ n = 2
```

```
use 1,
```

```
⊢ succ 1 = 2
```