SCHOOL OF COMPUTER SCIENCE

Formalising Ruler and Compass Constructions onto Lean 4 With a Second Line Added to the Title

And what not to do on the way

Oscar Martin Dickie

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Monday 11th August, 2025

Abstract

This project aims to provide a constructive guide on how to verify that the natural numbers are constructible under ruler and compass constructions using Lean 4. The reader is assumed to have knowledge equivalent to someone finishing the second year of the Maths and Computer Science degree at the University of Bristol. Some knowledge of real analysis, computation theory and functional programming languages will be useful in providing clarity to some of the arguments. The objective of this project is to employ mathematical concepts, algorithmic logic and type theory in order to prove the constructibility of the natural numbers in Lean 4. A second aim is to educate the reader on some potential pitfalls and traps I encountered when utilising theorem provers. In order to meet these targets, it is imperative to address the requirements for specific data types within the proof, acknowledge the limitations on what can be proven dictated by these data types, and provide an introduction to the relevant type theory addressing the challenges encountered. In the future, I anticipate to apply these findings in greater proofs. My results and guide will be based on Lean 4: the theorem prover and its documentation.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Oscar Martin Dickie, Monday 11th August, 2025

Contents

List of Figures

List of Listings

Ethics Statement

This project did not require ethical review, as determined by my supervisor, François Dupressoir

Notation, Acronyms and Definitions

- $\forall$ means 'for all' or 'for any'.
- $\exists$ means 'there exists'
- $s \in S$ means 's is an element of set S'. $\in$ indicates membership.

Chapter 1

Introduction, Aims and Motivation

Straightedge and compass constructions are also known as Euclidean constructions, ruler-and-compass constructions, and classical constructions. These constructions refer to the shapes, lines, points and other geometric figures that can be created using a ruler and compass. These constructions were first attempted by the ancient Greeks [5], who were able to construct sums, differences, products and square roots of given lengths.

1.1 Definition and set-up

I have aligned my definitions to align from the principles outlined in David A. Cox's book 'Galois Theory' [13]:

We are given the base set of constructed points (0,0),(1,0) in a 2 dimensional Cartesian space ($\Re^2$).

$$\mathbf{B} = \{(0,0),(1,0)\} \in \Re^2$$

We wish to **construct** more points in $\Re^2$. We may **only** use the following two tools:

- A **ruler** or **straightedge**, allows us to draw a line between any two points already constructed, or extend a line indefinitely.
- A **compass**, allows us to draw a circle whose centre is at any constructed point and whose radius is the distance between any two constructed points.

There are 3 ways to **construct** new points.

1. By intersecting two lines
2. By intersecting a line and a circle.
3. By intersecting two circles.

I will collectively refer to the circles and lines described above by a ruler or compass as **drawings**. It isn't necessary that the perimeter of a circle cross another point, but most of my results do use this aplication of the rules.

1.1.1 Constructible

The first two constructed points are (0,0) and (1,0). Points are **constructible** when they are created from **drawings** of previously constructed points. Each construction must terminate, meaning we cannot construct a point to be the limit of an approaching sequence of other points.

1.2 What is a theorem prover?

This following introduction is sourced from the Lean prover community [22]:
A theorem prover allows the programmer to create an algorithm which proves certain mathematical propositions. It is a piece of software that allows the user to define mathematical objects, allowing

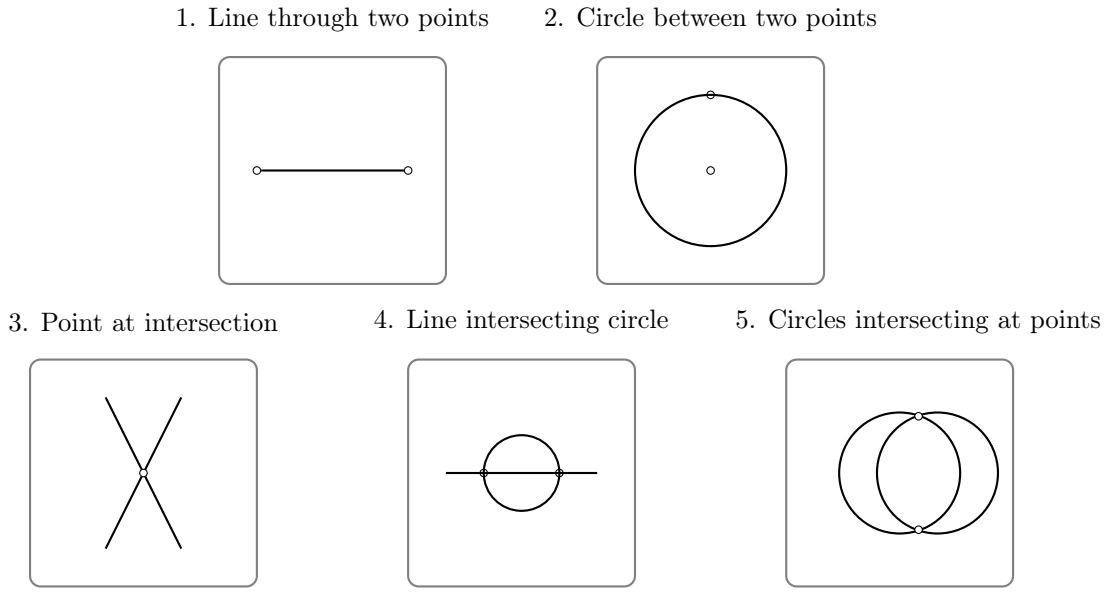

Figure 1.1: Different Geometric Configurations with Points

properties of objects to be designated, and proving that these designations are valid. The system checks that proofs on these objects are correct down to their logical foundation. The software tool verifies the correctness of a mathematical proof. In a formalization, all definitions are precisely specified and all proofs are virtually guaranteed to be correct.

1.2.1 Lean theorem prover

Lean is a functional programming language and interactive theorem prover [21]. The Lean mathematical library, mathlib, serves as a library for the community driven documentation which aims to build a repository of mathematical proofs formalised in Lean [33]. One of the projects aims is to formalise all of the undergraduate's course at Imperial College London.

1.3 Aims and objectives

The original aim of this project was to formalise Corollary 10.1.8. 'If α is constructible then $[\mathbb{Q}(\alpha) : \mathbb{Q}]$ is a power of 2' from the book 'Galois Theory' by David A. Cox [13]. I was motivated the goal of extend the mathib documentation on field extensions. The corollary seemed difficult to prove without a background in theorem proving. With this in mind I thought it necessary that I could first prove that the set of all constructible numbers were constructible, this would help me validate that all my algorithms to construct constructible numbers were valid. I considered breaking this down into steps by first proving that all rationals were constructible. Due to complications that could have arisen in proving this statement, proving that the natural numbers are constructible became my primary aim. Whilst still aiming to use the knowledge gained and apply it to proving the more difficult case. I encountered many challenges in formalising this proof onto Lean 4. A secondary aim is to provide this dissertation as a resource to help outline the challenges and why I faced them.

This dissertation will follow the following structure:

1. Describe the algorithms I used to generate the functions for my proof and argue that the natural numbers have an algorithm to prove their constructibility.
2. Justify the need for certain mathematical objects in this proof.

3. Define a 'Type' and how Lean uses them in proofs.
4. Introduce the reader to the creation of the relevant types.
5. Discuss the properties of the relevant types.
6. Critically evaluate what my proof. Outlining its successes and errors with respect to the constraints of the types.

1.4 Claims and Contributions

Ultimately, my **primary finding**, formalized in Lean 4, provides a constructive proof demonstrating the constructibility of the natural numbers. The code can be found in the auxiliary submission point or at the this GitHub link [1].

1.4.1 Originality

The formal development of Euclidean constructions in Lean is original in so far as there is no documentation of it in Lean's mathematical library [3]. To my knowledge, and as of this dissertation there exists no documentation of it. In particular under the sections of Euclidean geometry, field extensions, or terms related Euclidean constructions.

1.4.2 Methodology of my Primary Finding

The methodology of my primary finding included several key techniques [1]: using induction on finite sets of points to formalise a constructive proof of my theorem; deriving DecidableEq on the rationals to enable smooth evaluation of set membership; employing automated theorem-proving tools to simplify complex expressions and proof goals; and deconstructing my main theorem into its pertinent lemmas to facilitate its proof.

1.4.3 Findings

The natural numbers can be constructively proven in Lean 4 [1], this is made easy thanks to being able to derive DecidableEq on Rat (6.2.2), which allows for the Point, Line, and Circle types to derive an instance of DecidableEq (6.3). In turn this leads to an algorithm which is able to decide membership on Finset Point, Finset Line, and Finset Circle (6.3.1). This makes the constructive proof of my primary finding much easier.

1.4.4 Practical Applications

The methodology outlined in my primary finding provides a potential framework for proving the constructibility of the rational numbers. The functions needed would be structurally similar, albeit the functions would have to change to accommodate the type needed for this new proof. Many of the previously proved theorems and lemmas will not be of use [1], possibly except for one_step_construction_subset. It proves the result that for finite sets S ⊂ F, one_step_construction S ⊂ one_step_construction F. This could have many applications in proving constructibility.

1.4.5 Theoretical Advancements

As of this dissertation, I have not yet extended the mathlib documentation with my result as it doesn't extend the documentation in a meaningful way. However, I anticipate extending the documentation on field extension theory through the use of Construido.
The data type Construido supposedly supports a proof demonstrating the constructibility of the rational numbers (3.4). In theory, the type would support a similar, constructive, proof to the one in the primary finding. It would be able to support this thanks to a derivation of DecidableEq, allowing decidability of set memberships (6.2.2).

1.4.6 Broad Impact

Broadly, these claims have led me to create a guide to Lean 4, aimed at mathematicians and second-year students studying math and computer science. The guide intends to provide an analysis to the relevant type theory, algorithmic logic and other theories heavily involved in theorem proving.

1.4.7 Methodology of my Guide

The methodology in creating this guide to Lean 4 involves the following:

1. Approaching type theory from a basic set theoretic perspective, introducing philosophies of mathematics by providing relevant examples (4).
2. Motivating the need for different data types by explaining their relevancy (3).
3. Effectively introducing Lean's types and their properties relative to my primary aim (4.1).
4. Introducing algorithmic logic through constructively built examples; such as the introduction to a Turing machine, with the intent to use this to explain computability, non-computable functions and non-computable real numbers (6.1.3).
5. The introduction of decidability was also fulfilled through the use the halting problem example, which linked back to Turing machines (6.1.2).
6. Demonstrating algorithmic logic through digestible examples which purposefully avoided the need for lambda calculus, such as providing Lean examples for decidability (5, 6.2.2)
7. I explained the termination of terms, avoiding referencing lambda calculus (4.1).
8. Illustrating the challenges faced due to derivability of DecidableEq (6.2.2).

Chapter 2

Background Maths

Theorem provers are a way of simulating mathematical proof. Before we simulate the proof on Lean, let's formalise the maths and derive the appropriate algorithms. In this chapter I will provide the algorithms used to find constructible points.

I will use the following algebraic expressions on $\Re^2$ to represent lines and circles respectively. Let the point (a,b) describe the centre of the circle and r the radius of the circle [8].

$$line : Ax + By + C = 0 \qquad circle : (x-a)^2 + (y-b)^2 = r^2$$

The general equation for a line is above [18], where A or B can be 0 but not both. For a more intuitive and visual understanding of the algorithms, I will be splitting up each line into vertical and non-vertical cases. The non-vertical case of $y = mx + c$, where the finite gradient is m and the y intercept c. A vertical line can be expressed by where the line crosses the x axis (u,0), $x = u$. Both algebraic equations respectively represent all possible points a in 2 dimensional Cartesian Space. The set of points express different "shapes" or "drawings" in this space. So two geographic representations would intersect when they share a common (x,y) coordinate pair. Geometrically, these points correspond to the locations where the graphs of the two polynomials intersect on the coordinate plane. We can determine when they intersect by solving algebraically.

I will refer to lines with a finitely measurable gradient as **non-vertical**.

2.1 Intersection of two lines

Consider that a point would be constructed when two lines intersected.

We have 4 cases to consider: Two non-parallel lines must cross [15].

1. Non-vertical non-parallel 2. Vertical and non-vertical 3. Vertical parallel 4. Non-vertical parallel

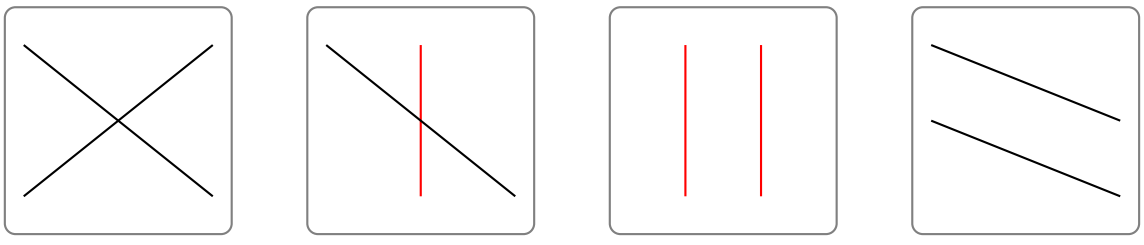

Figure 2.1: Intersecting Lines Scenarios

1. Since the gradient is finite, both lines can be expressed as follows.

$$y = m_1x + c_1, \quad y = m_2x + c_2, \quad m_1 \neq m_2$$

With these 2 simultaneous equations we may rearrange for the x intercept and substitute x into either line function to find the y intercept.

$$x = \frac{c_2 - c_1}{m_1 - m_2}, \quad y = m_1(\frac{c_2 - c_1}{m_1 - m_2}) + c_1 = m_2(\frac{c_2 - c_1}{m_1 - m_2}) + c_2$$

2. Let the red vertical line be expressed only by its constant x coordinate, a.

$$y = mx + c, \qquad x = a$$

They are not parallel so they must cross, rearranging gives us the coordinate of the intersection.

$$y = ma + c, \qquad x = a$$

3./4. The lines are parallel, so either they coincide at all possible points, or they don't intersect. In either case, no new point is constructed.

2.2 Intersections of a line and a circle

We have 4 cases to consider. When the circles do and don't intersect, and of those, when the lines are vertical. Let's first consider cases 1. and 2., where the equation for the line is in red and the equation

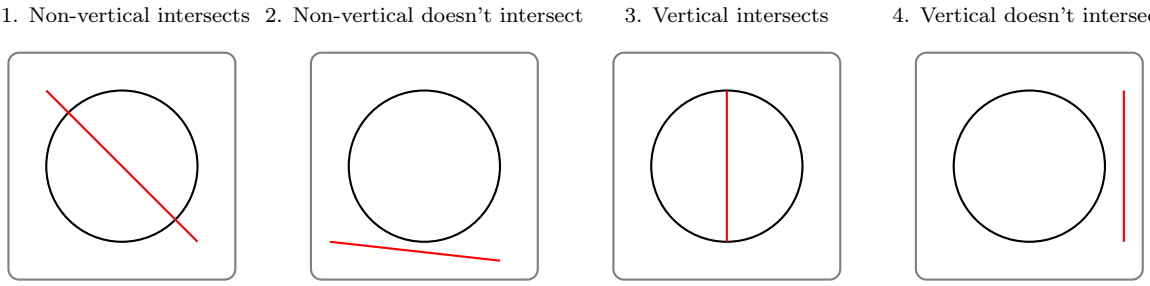

Figure 2.2: Line-Circle Intersection Scenarios

for the circle in black.

$$y = mx + c \qquad (x-a)^2 + (y-b)^2 = r^2$$

The associated constants m and c will be represented in red. Let's solve the simultaneous equation by substituting in y:

$$\begin{aligned}
(x-a)^2 + (mx + c - b)^2 &= r^2 \\
x^2 - 2ax + a^2 + (mx)^2 + 2m(c-b)x + (c-b)^2 &= r^2 \\
(1+m^2)x^2 + 2[m(c-b) - a] + a^2 + (c-b)^2 - r^2 &= 0 \\
\textit{Let } A = (1+m^2)x^2, \quad B = 2[m(c-b) - a] \quad C &= a^2 + (c-b)^2 - r^2
\end{aligned}$$

We can now solve for x using the quadratic equation:

$$\begin{aligned}
x &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \\
y &= m(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}) + c
\end{aligned}$$

Note that the discriminant Δ tells us when the line intersects. When $\Delta > 0$ the line must intersect at 2 points, when $\Delta < 0$ the line cannot intersect as we would get an imaginary solution and when $\Delta = 0$ the line is tangential and intersects at 1 point.
Now examine cases 3. and 4,, for a line represented by x = u, we substitute x and simplify to solve the equation.

$$\begin{aligned}
x = u \qquad (x-a)^2 + (y-b)^2 &= r^2 \\
(u-a)^2 + (y-b)^2 &= r^2 \\
(y-b)^2 &= r^2 - (u-a)^2 \\
y &= b \pm \sqrt{r^2 - (u-a)^2}
\end{aligned}$$

For exactly the same reasons as the Δ, the root must also tell us here how many intersections there are. Finally, we may evaluate x and y.

2.3 Intersections between circles

I will refer to circles with the same y coordinates as **horizontal** from each other and circles with the same x coordinates as **vertical** from each other.
We have 3 cases to consider for circles:

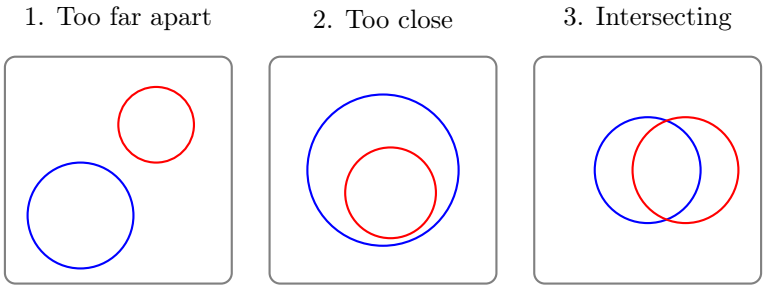

Figure 2.3: Intersecting Circles: Demonstrating Different Cases

We can verify if the circles intersect by comparing the distance between the centres to their radii. First let's consider case 1.
The reason they don't intersect is because the distance D between centres is greater than the sum of the radii [8].

$$D > r + s$$

Case 2. Here we consider when one of the circles is inside the other. This happens when one of the radii is greater than the sum of D and the other radius.

$$D + s < r \quad or \quad D + r < s$$

Intuitively you can think of these cases occurring when the circles are either too far apart or too close. So now in Case 3. we can guarantee that they actually intersect. Let's start by deriving the equations:

$$(x-a)^2 + (y-b)^2 = r^2 \qquad (x-u)^2 + (y-v)^2 = s^2$$
$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 = r^2 \qquad x^2 - 2ux + u^2 + y^2 - 2vy + v^2 = s^2$$
$$-2ax + a^2 - 2by + b^2 + 2ux - u^2 + 2vy - v^2 = r^2 - s^2$$
$$2(u-a)x + 2(v-b)y = r^2 - s^2 - a^2 + u^2 - b^2 + v^2$$

Case 3.1, $v \neq b$

$$y = \frac{-2(u-a)x + r^2 - s^2 - a^2 + u^2 - b^2 + v^2}{2(v-b)}$$
$$y = \frac{-(u-a)x}{(v-b)} + \frac{r^2 - s^2 - a^2 + u^2 - b^2 + v^2}{2(v-b)}$$

If the two circles overlap then this equation represents the supposed line that connects the two points from the circles' intersection. We can now use the previous formula for finding the intersection between a line and circle to determine where this line intersects with the circle to provide the points at which they intersect [8].

Be careful when the circles are **horizontal** from each other, the steps above would produce a vertical line. This is expressed in the equation when (v - b) = 0, the gradient would be undefined.

Case 3.2 u = b.

$$2(u-a)x = r^2 - s^2 - a^2 + u^2$$

We can confidently dismiss the scenario where $u = a$ since it would result in the circles sharing the same centre, thereby not sharing any new points, hence we proceed by dividing both sides by (u - v).

$$x = \frac{r^2 - s^2 - a^2 + u^2}{2(u-a)}$$

This expression represents the vertical line between the two points where the circles intersect.
Now we can retrieve the points of intersection between two circles which are **horizontal** from each other using the equations we have from intersections between a vertical line and a circle

2.4 Constructible numbers

The naturals are constructible; this the mathematically simple result that I have proved in Lean. Much of this dissertation will build up to why this simple result was a challenge.
I will give a very simple proof by induction to show they are constructible.
inductive step: let's assume that points (k,0) and (k+1,0) are constructible, for any k $\geq$ 0.

1. Draw a straightedge between these points on the x-axis.
2. Draw a circle with a compass from the points (k,0) and (k+1,0) with centre at (k+1,0).
3. Calculate that (k+2,0) is a point of intersection between the straightedge and the circle.

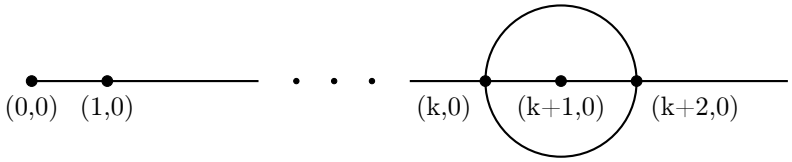

Figure 2.4: Constructing the Natural Numbers

We have shown that if any given points (k,0) and (k+1,0) are constructible, then so is (k+2,0). Since the base cases of (0,0) and (1,0) are already constructible by definition. By induction, we have proved that all natural numbers $n \geq 0$ are constructible. This argument is briefly verified by 'Example 10.1.2' in the 'Galois Theory' book [13]. Notice that this proof relies on only the line $y = 0$ and circles of radius 1 around natural numbered coordinates. We could use the rationals ($\mathbb{Q}$) to describe coordinates for the points; the gradient and y intercept of the lines; and the radius and coordinates of the centre of the circle. In calculating these intersections, we also do not require any square roots. The calculations described by the previous algorithms to find these particular intersections do not encounter irrational numbers.

We say a number n is constructible if the point (n,0) is constructible. If in the inductive stage we showed that (k-1,0) is constructible from (k,0) and (k+1,0), we have shown that the negatives are also constructible and so all integers are trivially constructible.

How I formalised this in Lean

Overall, I defined a point to be constructible if it was a member of another set of constructible points.

Suppose we created a function called '**one_step_construction**' which given a set of points, returns the set of all possible points that could be constructed from the first set of points. This function would first create the set of all possible lines and all possible circles from the given set of points. The function would

then find all the intersections between all lines and lines, all lines and circles, and all circles and circles. For example, applying one step construction to B would calculate the points of intersection between the circle around the origin, the circle around (1,0), and the line through the x axis. The line would intersect the two circles at two points each, at (-1,0),(0,0),(1,0) and (2,0). The circles would intersect at $(0.5, \frac{\sqrt{3}}{2})$ and $(0.5, -\frac{\sqrt{3}}{2})$, to see this please see the diagram below 3.1.

To summarise, **one_step_construction** $\{(0,0),(1,0)\} = \{(-1,0),(0,0),(1,0),(2,0),(0.5, \frac{\sqrt{3}}{2}),(0.5, -\frac{\sqrt{3}}{2})\}$.

Let another function, called '**n_step_constructions**', recursively invokes the previous function from the base set, then all the set of points returned by **n_step_constructions** would be the set of all constructible points after n iterations from the base set. The function would therefore be defined as:

- **n_step_constructions** (0) = {(0,0),(1,0)}
- **n_step_constructions** (n) = **one_step_construction** (**n_step_constructions** (n-1))

Hence define a natural number, n, as constructible if there exists a value k such that the point (n,0) belongs to **n_step_constructions** (k). To formally demonstrate this we express the theorem as follows.

$$\forall n \in \mathbb{N}, \exists k \in \mathbb{N} \; s.t. \; \{n, 0\} \in \textbf{n\_step\_constructions} \; (k)$$

Chapter 3

Construido

In order for the algorithm to determine whether a point is constructible in Lean, we need to define the mathematical object of a point. They are defined by their attributes, namely the x and y coordinates. Additionally, we would need to define what the attributes are by what they inhabit. The point (1,-1) is described by its x coordinate 1 ($1 \in \mathbb{N}$) and its y coordinate -1 ($-1 \in \mathbb{Z}$) .

Since a constructible point is defined as being constructed from other constructible points, we must also define all preceding points that it is constructed from. In this chapter I will introduce and explore some of the limitations of these mathematical 'containers' to describe the x and y coordinates of each point.

We know points are constructible when they are found between the intersections of drawings from other constructible points. In fact, in order for the algorithm to determine whether a point is constructible, we must also properly define the lines and circles created between points. Hence attributes like the gradient of a line or radius of a circle must also inhabit a proper 'container' (or datatype).

3.1 Rational

It is demonstrated that the point (0.5,0) is constructible by bisecting the line between points 0 and 1:

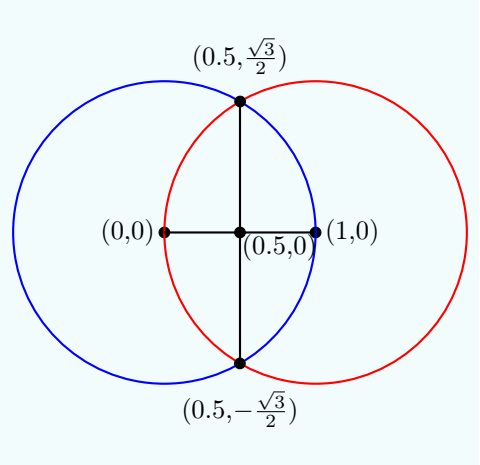

Figure 3.1: Constructing 0.5

However, when finding the intersections between the two circles we encounter the radical $\sqrt{3}$. This is not in the set of rational numbers $\mathbb{Q}$.
In fact this is also true for all natural numbers n and m, n ≤ m. Let $d = |n - m|$ is the distance between the points. Yet still able to prove the natural numbers are constructible. The points of intersection between the circles would be irrational. We can see this by the algorithms described in chapter 2. We simply cannot evaluate the **constructible** point (n + d/2, 0) exclusively in terms of the rationals, and hence we are limited to proofs on the x axis with all points described by the rationals.

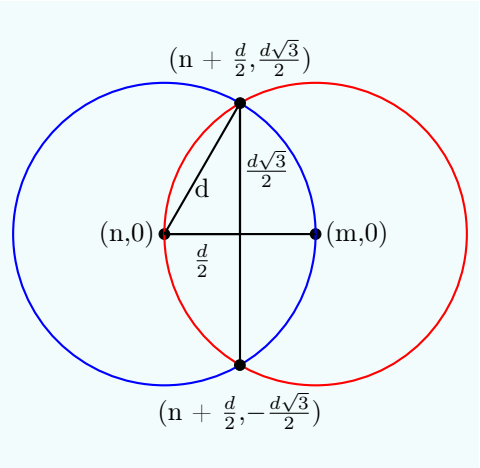

Figure 3.2: Bisecting the Line

3.2 Real

One might also consider evaluating constructible points in the reals since they are complete [23]. Loosely speaking, meaning that there are no 'gaps' or 'missing points' on the real number line. As opposed to the rationals where there are many gaps, namely the irrational numbers ($\sqrt{2}, \pi, e...$), as I will explore in chapter 6, the fact that most real numbers cannot be computed makes certain proofs difficult.

3.3 Float

The **Float** datatype in Lean is consistent with the IEEE 754 binary64 format, analogous to double in C or f64 in Rust [2]. Floats are represented with a finite precision approximation. Hence we cannot associate with them in general [17], to illustrate this let a,b,c be some combination of very large and small Float numbers, they are just an approximation limited by the computing capabilities of the program, so for some values a + (b + c) $\neq$ (a + b) + c. Although proofs might work for small numbers, we couldn't guarantee that an algorithm using Floats would arrive at the actual point, only that the computed points is 'approximately' close to what it should be. If we employ two competing algorithms to arrive at the same constructible point, there is no guarantee that their results would arrive at the same floating point number.f For these reasons the floats are a bad 'container' for proofs.

Additionally, multiple values are stored under the same value '`NaN`' (Not a Number). Numbers like $+\infty, -\infty, \sqrt{-1}$ are all stored under the same value, so `NaN` $\neq$ `NaN`. Hence equality of the floats is not reflexive, for a float x, we do not have that x = x. For these reasons, any algorithms which relies on a proper notion of equality will encounter challenges.

3.4 Construido

In order to prove that the rationals or that some surds are constructible, we require a new 'container' or datatype that I will name **Construido**. Let's suppose Construido contains all constructible points built from the base set B, with a notion of equality (under a reduced form) and decidable equality (6.2.2).

Thanks to Corollary 10.1.8 [13], we have a description for how we might want to create Construido. However, the formalisation of this mathematical object which encompasses all constructible numbers (with the appropriate properties for the proof) is an exercise for future projects. As of this dissertation, I haven't been able to formalise this mathematical object in Lean.

Chapter 4

Introduction to Lean's type theory

In order to justify that we can prove the statement 'the naturals are constructible' (in Lean), we must justify that the system can support the method of its proof, and we must also justify the creation (and use) of rational numbers in this proof. This chapter will explore some of the basic theory behind these justifications, such as what a type is and what a constructive proof is.

In mathematics we consider the real numbers $\mathbb{R}$ to be a set and 54321 to be an element of that set. Lean uses **type theory** as a foundation for mathematics; the theory was principally developed to cope with paradoxes found in set theory [4]. A common example is **Russell's paradox** [28]. A paradox arises when we consider the set R of **all** sets that do not contain themselves. If the set R is not contained in R, then by definition, it must be in R. If the set R is contained in R, then again by how we constructed R it must not be in R. Or formally, $\{R : R \notin R\}$ does not exist.

4.1 Martin-Löf's type theory

One of Lean's fundamental ideas is **Martin Löf's intuitonistic type theory** [14], the theory provides a possible foundation for mathematics through the use of computational processes. In Martin Löf's own words:

> "We shall think of *mathematical objects* or *constructions*. Every mathematical object is of a certain kind or *type* [... and] is always given together with its type. ... A type is defined by describing what we have to do in order to construct an object of that type. ... Put differently, a type is well-defined if we understand ... what it means to be an object of that type. Thus, for instance N→N [functions from N to N] is a type, not because we know particular number theoretic functions like the primitive recursive ones, but because we think we understand the notion of number theoretic function *in general*. "[25]

Informally, in Lean we say that a **type** plays the role of a 'set', and a **term** plays the role of its elements. The expression `t : T` means that: the term `t` is of type `T`, `t` is an instance of `T`, `t` is an example of `T` and `t` is proof that the type `T` exists. As we will later see, Lean uses these types to describe mathematical objects [9]. In fact, all types `T` describe a proposition and all propositions are a type, the type `T` describes the proposition 'is this type inhabited' and a proposition is the type of all proofs of that proposition.

One possible application is as follows: Let the proposition be the Pythagorean theorem, then let the type `PT` be the type of all proofs of the Pythagorean theorem (to which there are many [24]). Any term `pt` of `PT` (`pt : PT`) would then be a proof of the theorem. There are many proofs of the theorem and they would all be evidence of the type `PT`'s existence and hence proof that `PT` is inhabited and proof that the theorem is true.

A more trivial example in Lean would be the statement `0 : Nat`, which means `0` inhabits the type of the natural numbers. The term `0` is also a proof that `Nat` is inhabited. Similarly for `true : Bool` where `true` is the boolean true value and `Bool` is the type of all boolean values. All `types` are a term of some larger type. Some of the types that Lean allows us create are as follows

- The type of all propositions `Prop`

- Propositions p, p : Prop, for example PT : Prop.
- Proofs h, which are terms whose types are propositions (h : p), if pt were the algebraic proof of the Pythagorean theorem then pt : PT.
- Functions, if a function's input is of type A and output of type B then f : A → B.

Another example is $(\exists(x : \mathsf{Nat}), x \geq 0) : \mathsf{Prop}$, Nat is the type of natural numbers. Since all types have a type, Nat : Type and Prop : Type. Referring back to Russel's paradox, Lean provides a hierarchy of types to avoid the paradox. Each type is an instance of a greater universe of types [**?**]. The lowest level of this hierarchy is Type, it contains the most fundamental types that are used most commonly in proofs. A lot of details have been overlooked, but this is what begins to give way to a large degree of mathematical expressivity.

In Martin-Löf's Type Theory: a term is said to terminate when it's computational process reaches it's conclusion and stops in a finite number of steps, leading to a reduced or normal form [26]. Normalisation is a key concept to the theory, it employs a series of strict typing rules to ensure each well-typed term terminates. This helps validate the confidence of the system. Lean, however, allows the creation of some non-terminating terms, we must mark them as non-computable and apply different logical constraints.

4.2 Constructivism

Lean is a theorem prover and programming language that can be used in constructive settings, but it is not inherently constructive in the same way as Martin-Löf Type Theory. Constructivism describes a philosophy of maths where one must provide an example of a mathematical object in order to prove that it exists. In a **constructive proof**, each mathematical object can be effectively constructed through a defined procedure (or algorithm) [31]. For example, the proposition (or mathematical object):

$$\exists n \in \mathbb{N},\ n^2 - 4n + 4 = 0$$

is **constructively** proved by providing the evidence that 2 solves the expression, $2^2 - 4 * 2 + 4 = 0$.

Linking back to the proof

Constructive proofs can be seen as the recipe behind a verified algorithm which creates a mathematical object. Recall that in section 2.4 we stated that a point was constructible if that point was found within the set of constructible points (after a certain number of steps). We prove it by providing the evidence that our point satisfies the conditions needed for it to be in that set. One can see that the proof is indeed constructive. In order to demonstrate to Lean that this point is in the set, we must also construct types to describe the points, lines and circles used in our algorithms.

4.3 Dependent Type

Lean is based on a form of dependent type theory [9], which can naturally express many concepts from constructive mathematics, known as the Calculus of Constructions [12] with inductive types [27].

A **dependent type** is a type which is defined to depend on a value [30]. A typical example arises when considering the type List a in Lean; section 5.4.7 shows how the list is inductively defined in Lean. List a is dependent because because the type depends on its parameter 'a', allowing distinction between List Float and List Bool.

Functions can also be **dependent**, let insert describe a function which inserts an element to the head of a list. It would expect parameters for the type (E : Type), an element (e : E) and a list (ls : List E). insert Nat 1 [4, 5] would be an instance of List Nat, whereas (insert Bool true [false, false]) : List Bool. The the function's type clearly **depends** on its parameter.

Chapter 5

Introduction to Lean's Types

As said in the previous chapter, Lean uses type theory to provide access to constructive proofs. This chapter aims to demonstrate how Lean creates the types relevant to the proof. And to introduce some concepts to help the reader understand my code. We will observe how Lean builds different datatypes and in particular begin to explore the properties of Float, Real, Rat and Finset.
This introduction presents concepts from the book 'Theorem Proving in Lean 4' [20] and code from the mathlb documentation [3].

5.1 Prop

In Lean, Prop is the type encompassing logical formulas. Here is an example:

```
(∀ (x y a b: ℕ), a > 1 ∧ b > 1 → x^a - y^b = 1 → x = 3 ∧ a = 2 ∧ y = 2 ∧ b = 3 ) : Prop
```

Listing 5.1: Catalan's conjecture

Lean supports the declaration of multiple terms of a type, x y a b : ℕ means x : ℕ, y : ℕ, a : ℕ and b : ℕ. ℕ is an abbreviation for Nat, the type of natural numbers. Also known as the Catalan's conjecture, the proposition above has type Prop.

Suppose we created a function which describes the property of a natural number being constructible, as described in section 2.4. The final theorem I proved would look like this:

```
(∀ (n : ℕ), constructible (n)) : Prop
```

Listing 5.2: Constructible proposition

For now, it is just a proposition, Lean doesn't know if it is true or not.

5.2 Structures

A **structure** is a kind of type which holds a record of **fields**, to be packaged, represented and later accessed.

One way to think of it is that structures are used to create mathematical objects by describing a finite number of 'things' needed to create it (its fields).

```
structure <name> <parameters> <parent-structures> where
    <constructor-name> :: <fields>
```

Listing 5.3: General syntax for Structures

Where anything in between '<' and '>' is to be filled in by the user. Most parts aren't mandatory, but it is important to understand that the **fields** describe the mathematical object's attributes or the things needed to create it. They are the pieces of data that we would like to package under this new record.

If we wanted to create the type of a **point** in 2D space we could have the fields to associate to the x and y coordinates. We must also declare the type that the fields themselves have.

```
structure Point where
  x : Float
  y : Float
```

Listing 5.4: Point structure using Float

For now we will use the built in datatype of Float to introduce this concept. They function similarly to the floats in C or C++.

naturally we would like to create elements of this Point structure. We could use the **constructor** to do this, by default the constructor's name is **mk**.

Using the constructor means we would have to remember the order in which we defined the fields, this can be cumbersome and make parsing code difficult, but I will demonstrate both anyway.

Below is the general syntax for how to create instances of a structure.

```
{ <field-name-1> := <expr-1>, <field-name-2> := <expr-2> , .... : structure-type }
    or
{<field-name-1> := <expr-1>, <field-name-2> := <expr-2>, <field-name-3> := <expr-3>, ... }
```

Listing 5.5: General Structure syntax

Lean is often able to infer the structure type given the fields it expects, but there is no harm in explicitly stating it.

We will now create some **instances** of Point and evaluate their types.

```
#check {x := 10, y := 2 : Point} -- {x := 10, y := 2} : Point
--Lean tells us the instance {x := 10, y := 2} has type Point no surpises here.

#check Point.mk 0 1 -- {x := 0, y := 1} : Point
```

Listing 5.6: Point Instances

The expression #check is used in Lean to evaluate an expression's type. Comments begin after the characters '- -' and all code after it is ignored up to the end of the line. Typically in the comments will contain the evaluation from Lean infoview. The Lean infoview gives me information about the code, such as the type of an expression, definition of a function or the evaluation of the function.

5.2.1 Fields

As previously stated, fields can be thought of as properties of structures, where the expected type of each field must be stated. We may check the type of a field or evaluate its stored value as follows:

```
#check {x := 10, y := 0 : Point}.x -- {x := 10, y := 0 : Point}.x : Float
#eval {x := 1, y := 2 : Point}.y -- 2
```

Listing 5.7: Field Evaluations

The expression #eval evaluates the expression given. While structures provide a neat way of collecting a fixed set of fields, we often encounter an arbitrary number of elements that we would like to collect inductively under the same datatype, such as the natural numbers.

5.3 inductive Datatypes

In short, an **inductive** type is specified by a list of **constructors**. Each constructor declares a different method on how to create the inductive type in question. The constructor could even require an instance of the original inductive type as part of the method. Recall that structures only had one non-recursive constructor, namely the one that generated its fields.

```
inductive Indv <parameters> where
  | constructor₁ : ... → Indv
  | constructor₂ : ... → Indv
  ...
  | constructorₙ : ... → Indv
```

Listing 5.8: General Syntax for inductive Types

The character '|' is used for pattern matching. Pattern matching decomposes data types according to their constructor, the way the data is structured facilitates the conditional execution. The following examples may help us see this.

5.4 Built in datatypes

The relevant code for how Lean builds these datatypes is sourced from the 'Mathlib' documentation [3]. However, some of the finer details have been removed to facilitate the explanation of these types.

5.4.1 Bool

Our first inductive datatype gives us the data type for boolean logical truth values, true and false. It requires no parameters.

```
inductive Bool where
  | false : Bool
  | true : Bool
```

Listing 5.9: Bool Type

This means that the only way to construct an instance of type Bool is by providing a boolean true or false instance. Much like structure, we can create an instance of Bool by it's constructor. Bool.true : Bool and Bool.false : Bool. Since Bool instances are common, Lean abbreviates Bool.true to true and Bool.false to false. The true and false boolean instances should not be confused with the True and False instances of propositions. Here are some uses for Bool:

1. Storing truth values.
2. Evaluating logical operators like || (boolean or), && (boolean and) or == (boolean equality).
3. To pattern match.
4. For programming applications.

5.4.2 Nat

The natural numbers can be defined inductively from 0 or a successor of another natural number thanks to the Peano axioms [16].

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Listing 5.10: Nat Type

Nat has two constructors. The smallest natural number is Nat.zero, normally written 0. Any other natural number is Nat.succ n, normally written succ n, where n : Nat. For constructor succ to be used, it requires an example of another natural number n as a parameter to be the successor of (succ n = n + 1). This is how the type Nat is created; Lean then creates functions and theorems to apply the Peano axioms and determine how zero or succ n interact.

5.4.3 Int

The Int type represents the integers. They are built inductively from the naturals.

```
inductive Int where
  | ofNat (n : Nat) : Int
  | negSucc (n : Nat) : Int
```

Listing 5.11: Int Type

The constructor ofNat is used to assert that any natural number is an integer. The constructor negSucc is used to assert that the negation of a successor is an integer, where the negation of succ n is -(n + 1). So -1 is an abbreviation for Int.negSucc 0. Since Lean has an algorithm to compute Nat, the system also guarantees that ofNat and negSucc are implemented properly, this is done thanks to the fact that the integers are described by the set of negative and positive natural numbers.

5.4.4 Rational

The rationals (Rat) are defined as a structure. Rat is implemented by coupling an Int and a Nat, the numerator and denominator. The denominator is positive and must be coprime to the numerator.

```
structure Rat where
  num : Int                  --num is the numerator.
  den : Nat := 1             --den is the denominator.
  den_nz : den ≠ 0 := by decide
  reduced : num.natAbs.Coprime den := by decide
```

Listing 5.12: Rat Type

The expression t : T := v means that for a term t of type T, we have initialised the term t with the value v. For example, den : Nat := 1 means 1 is assigned by default to the field 'den', meaning that if no value is given for the denominator, it is assumed to be 1. So in this case we could say that $\frac{3}{1}$ is represented by Rat.mk 3 in Lean.

The first 2 fields should make sense by now. Intuitively, the last two make sure the Rat in question is stored (in some reduced form) as a fraction whose numerator and denominator are co-prime.

1. den_nz : $den \neq 0$:= by decide
   - den_nz : den ≠ 0; declares that den_nz is a variable of type den ≠ 0.
   - := by decide; decide is a proof tactic of Lean. Lean is trying prove den ≠ 0, if the proof fails then Lean is unable to assign any variable to den_nz. If nothing can be assigned to this field then the structure cannot be created and Lean is unable to assign a type to its term. It is a cleaver way of guaranteeing that the denominator is not zero. For example: Rat.mk 1 0 would fail to create an instance.
2. reduced : num.natAbs.Coprime den := by decide
   - reduced : num.natAbs.Coprime den; declares a new variable of type num.natAbs.Coprime den.
   - := by decide; does the same thing as above. Where by decide is trying to prove that num and den are coprime, and so fails when the aren't. For example Rat.mk 2 4 would also fail.

Again, Lean then uses this type and creates functions and theorems to provide algorithms for division, multiplication, addition.... etc. This ensures that Rat is implemented properly.

5.4.5 Real

The equivalence class between rational Cauchy sequences constructs the type Real in Lean.

```
structure Real where ofCauchy ::
  cauchy : CauSeq.Completion.Cauchy (abs : ℚ → ℚ)
```

Listing 5.13: Real Type

5.4.6 Float

As explained before in section 3.3. A float is just an approximation to a number. Computers have finite resources. This can lead to complications.

```
structure Float where
  val : floatSpec.float
```

Listing 5.14: Float Type

5.4.7 List

A List is an inductive type, it is either an empty or not. A non-empty List is described by it's head and tail. The tail is another List and the head is just the first element of the list.

```
inductive List (α : Type) where
  | nil : List α
  | cons (head : α) (tail : List α) : List α
```

Listing 5.15: List Type

The expression (α : Type) denotes the type's parameter. The constructor 'cons' requires 2 parameters, If a : α and ls : List α, then a list that starts with the element 'a' and has ls as the rest of the body is constructed by cons a l. The empty list [ ] is constructed by nil. A List is **ordered** because we know where each element is located relative to each other. The list [1, 2, 3] would be created as below.

```
#eval List.cons 1 (List.cons 2 (List.cons 3 [])) -- [1, 2, 3]
```

Listing 5.16: List Example

Lean is able to infer the type parameter Nat here.

5.4.8 Finite Set

A finite set of elements having Type α. Is defined as an **un-ordered** list (a Multiset) with no duplicates. We do not know the location of any element.

```
structure Finset (α : Type) where
  val : Multiset α
  nodup : Nodup val
```

Listing 5.17: Finset Type

I used Finset to create the set's of points I used in my proof due to their expressive nature and the resources available on mathlib to create functions and prove propositions with them.

5.5 Function tutorial

When given function name, its parameters, the return type and the function body, the keyword def is used to define a function.

```
def function_name <list_of_parameters> : <return_type> := <function_body>
```

Listing 5.18: General Structure of a Function

```
def eg_funct : Nat := 1 -- assigns the value of 1 to eg_funct.
def dos := eg_funct + eg_funct -- Lean can often infer the return_type of the function.
                               --now dos has value 2.
def mul3 (n : ℕ) : ℕ := n * 3 -- this function multiplies is parameter by 3.
```

Listing 5.19: Function Example

Inputs to the function are applied to the right. If the function requires multiple inputs, they must be fed into the function in the correct order.

```
#eval eg_funct -- 1
#eval dos      -- 2
#eval mul3 dos -- 6

#check
```

Listing 5.20: Function Evaluation 1

Recall that we have created a datatype called Point in the structures section. Let's make some points, and create a function which is able to output the distance between any 2 points using Pythagoras's theorem.

```
def origin  : Point := { x := 0.0, y := 0.0 }
def origin1 : Point := { x := 1.0, y := 0.0 }
def distancepp (p1 : Point) (p2 : Point) : Float :=
    Float.sqrt ( ( (p2.x - p1.x)^2 ) + ( (p2.y - p1.y)^2 ) )
```

Listing 5.21: Distance Between two Points

However, this function has Point parameters whose fields are of type Float. Hence the method for the square root is relative to floats.

```
#eval origin.x -- 0.0000
#eval origin1.x -- 1.0000
#eval origin.x == origin1.y -- true
#eval distancepp origin (Point.mk 3 4) -- 5.0000
```

Listing 5.22: Function Evaluation 2

Where == was used to evaluate a Bool equality between the values given.

Chapter 6

Properties for the Proof

Now that we understand how the relevant types are created in Lean, we must then be able to prove propositions made using them. In this chapter, I will explain the algorithmic logic behind a decidable proposition and demonstrate how Lean applies this logic using type classes. This discussion will help to illustrate the challenges I faced.

6.1 Algorithmic logic

Lean algorithmically proves propositions. We would like an algorithm to derive truth or falsity from said proposition. Before we can make decisions on a proposition, lets define the concept of an 'algorithm' through the use of a Turing machine.

6.1.1 Turing Machine

A Turing machine [11] is a theoretical computational model introduced to help formalise the notion of **computation** or of a general **algorithm**. It has the following components:

- A **limitless memory-tape**. It is divided into cells or squares where each square may be blank or only contain a single symbol or character from some finite alphabet (possibly '0' or '1'). This means the tape applies no limit to the number of characters expressed on it.
- A **scanner**. It is able to read and write a symbol onto the cell it is above, it then moves left or right by one cell at a time or stops moving. The scanner contains mechanisms that enables it to read, write and move in the chosen direction.
- The **current state**. There could be (finitely) many states that the machine can cycle through. This is what dictates the current instructions to the scanner, possibly also updating the current state after it's use. There must be at least one start state and at least one stop state. For example, state 1 could be 'If a 0 is read, then write 0 and move left; otherwise, do not write anything and move left' and state 2 could be 'if 1 is read, then stop; otherwise move right' and so on...

In summary:

1. The scanner reads the character at the current cell.
2. Based on the current state and the symbol, the scanner is instructed to do some combination of 'write something', 'move', 'change state' or 'stop'.
3. The process repeats until the machine stops, or halts.

Let's consider the '**Halting Problem**' as a proposition: given the source code for a Turing machine and an its input, can we determine whether the program will terminate on that input, or go into an infinite loop?

Thanks to Turing's theorem we know that there is no algorithm to solve the halting problem for all algorithms [29]. From this we know that not all propositions can be proved or disproved. To avoid this, Lean requires us to first give evidence. While proving a proposition in Lean, it would be difficult to prove anything if we had to first provide evidence that the proposition is provable or disprovable every time. As we will soon see, Lean groups these provable propositions so that the proof assistant can automate certain goals and streamline the proof.

6.1.2 Decidable

If there is a finite algorithm which can always solve a proposition and provide evidence of its truth or falsity, we say the proposition is **decidable** [32].

The **halting problem** isn't decidable. There are propositions, p : Prop, where we cannot say p or ¬p is true. We cannot prove or disprove them.

Lean derives individual instances of the concept of **decidability** for different types of propositions [19]. For example, one might want to prove the proposition n = 0 : Prop is decidable (for some n : Nat). The algorithm in question would break up n according to the definition of Nat. We would now be left with 2 cases, when n is 0 and when n is the successor of another natural number. By the definition of Nat the first case is true and the second case is false. Later we will briefly see how one might formalise this proof the code snippet.

6.1.3 Computable

We say function f is called **computable** if there exists an algorithm (or Turing machine) that can compute the function's value for any valid input in a finite amount of time [29]. For example, the function which checks primality for a given number n is computable. An algorithm can be given by checking if the numbers less than n divide it (excluding 1 of course).

There are also functions that are not computable. Let H be a function on a pair of inputs (p,i).

- Let p be a Turing machine.
- Let i be the input to that Turing machine.

Now define H(p,i) as follows:

$$H(p,i) = \begin{cases} 0, & \text{if the program p halts on the given input i} \\ 1, & \text{otherwise} \end{cases}$$

Again, thanks to Turing's theorem, we know that we cannot decide if p halts on any given input, so there is no algorithm which can compute H. It is **non-computable**.

A **real** number is considered computable if there exists a computable function that can compute its digits to any desired level of precision [6]. For example the numbers 1, π *or* e all have specific algorithms to generate their decimal expansions. However, most real numbers are non-computable because they are uncountable [7]. An example of a non-computable number is Chaitin's constant [10]. It is a real number between 0 and 1. The constant is formed by summing the probabilities of a program halting for all possible inputs.
Hence any function which uses a real number cannot be guaranteed to terminate, which implies that any functions (or terms) in Lean which use real numbers cannot be guaranteed to terminate. We must mark them as non-computable. We cannot, by definition, use non-computable terms in constructive proofs. Other logical constrictions must be used in poofs with them.

6.2 Type Classes

In Lean, any family of types can be labeled as a **type class** [20]. Any element element of a type class can be seen as an instance. For example, we might want to create a canonical effect of addition where instances of this type class would behave similarly over Nat, Int, Float and many others. We then use instances to individually derive the desired effect, as we will now see with the type class for decidability.

6.2.1 Decidable

From an algorithmic perspective, the Decidable type class allows for the derivation of a procedure that irrefutably determines the truth of a proposition [20]. It does this when given a proof that p is true or p is false. We derive Decidable on different propositions by matching proofs h of the proposition p or ¬p (not p). In the standard library, Decidable is defined formally as follows:

```
class inductive Decidable (p : Prop) where
  -- Prove that ‘p‘ is decidable by supplying a proof of ‘¬p‘
  | isFalse (h : Not p) : Decidable p
  -- Prove that ‘p‘ is decidable by supplying a proof of ‘p‘
  | isTrue (h : p) : Decidable p
```

Listing 6.1: Decidable

From a logical perspective, possessing an element t : Decidable p represents a stronger condition than possessing an element t : p ∨¬p. As a result, Decidable supports computational strategies of propositions as long as they are possible. Given p : Prop, the expression ‘if p then X else Y’ is effectively used when we are able to verify p's decidability. Hence the type class is to used automate proofs of decidable propositions.
Below is an example of proving decidablility, I create an instance of n = 0 is decidable by splitting up n into the cases of Nat. Any n : Nat is either 0 or succ n. If the natural number n is zero then it is trivially true that zero = 0, the tactic rfl shows this. Any successor of a natural number is also not equal to 0, succ_ne_zero is proof of this. When we define this function, Lean asks us to provide it's algorithm and show that it terminates. We do this by using match to split up n into it's cases. Recall that isTrue and isFalse are abreviations for Decidable.isTrue and Decidable.isFalse, respectively. The symbol ‘=>’ is used to return the expression to the right of it.

```
def eqn_0 (n: Nat) : Decidable (n = 0) :=
  match n with
  | 0 => isTrue rfl -- isTrue rfl is evidence of Decidable (0 = 0)
  | succ n => isFalse (succ_ne_zero n) -- isFalse (succ_ne_zero n) : Decidable (succ n = 0)
```

Listing 6.2: Decidable example

The function eqn_0 is now an instance of of Decidable (n = 0). We could now, inefficiently, use this to evaluate if (5 = 0) then true else false. Lean of course provides decision algorithms to solve and automate many propositions, such as n = m for n m : Nat.

6.2.2 Decidable Equality

The type class DecidableEq on some type T asserts that for all a b : T, the equality ‘a = b’ is **decidable**. It allows us to evaluate the boolean value of the proposition ‘a = b’, given a proof of its decidability. For example, **DecidableEq** exists for **Bool**:

```
def booleq (a b : Bool) : Decidable (a = b):=
  match a,b with
  | true, true => isTrue rfl
  | true, false => isFalse Bool.noConfusion
  | false, true => isFalse Bool.noConfusion
  | false, false => isTrue rfl

def DecBEq : DecidableEq Bool := booleq
```

Listing 6.3: DecidableEq on Bool

Bool.noConfusion is proof that true ≠ false and false ≠ true. Thus DecBEq proof of an algorithm which can derive boolean equality.

Nat

DecidableEq exists on Nat thanks to the Peano axioms[16]. We know for all n m : Nat, n = m is decidable because the natural numbers are either 0 or the successor of another natural number. The algorithm which decides if they are equal or not is to compare the number of successions used in n and the number of successions used in m.

Int

Int is built from 2 cases of Nat, positive and negative. For two integers x and y, the algorithm that decides if x = y could check if the integers have the same sign and same magnitude, where the magnitude is found by taking the absolute value of the integers (|x|, |y|). If x and y do not have the same sign then they are not equal, and if two numbers have the same sign then we can apply the decidable equality algorithm from the natural numbers on |x|=|y|.

Rat

Two rational numbers $\frac{a}{b}$ and $\frac{c}{d}$ are equal when a*d = b*c, so we can use the algorithm for equality on the integers to decide if $\frac{a}{b}$ and $\frac{c}{d}$ are equal when a*d = b*c. Hence we can derive DecidableEq on Rat.

Real

The reals do not have decidable equality, which is to say that there is no algorithm which can determine whether any two real numbers are equal or not. This is because most of them are non-computable (6.1.3). For a b : Real, there is no way to compute a - b and justify a - b = 0.

In Lean's case, they are defined by a rational Cauchy sequence. There may be two completely different Cauchy sequences with the same limit, and yet no algorithm that can decide if they are equal.

Float

We can't however derive DecidableEq using the Float type, the datatype isn't native to Lean and it isn't associative. There are some computations using floats that arrive to the 'wrong' answer due to the limitations of a float being an approximation , as described in section 3.3. From Lean's perspective: given a b c : Float, we cannot provide a reasonable algorithm to decide if a + (b + c) = (a + b) + c is true or false without knowing a lot more about a, b, c and how they are computed. Essentially, floating-point equality can be unreliable because even identically computed values might differ due to minute differences in computation paths, hardware, or even compiler behavior.

Membership of Finset

Let F : Finset B, for some type B, and e : B. Then $e \in F$: Prop asks if e is a member of F. Evaluating $e \in F$ depends on deriving DecidableEq for the underlying type B. This is because an element e is in a finite set F if and only if there is some element i in the finite set such that e = i. This means that e and i must be of the same type, B. If we want e ∈ F to be decidable for any e, then e = i must also be decidable. Hence, deriving DecidableEq on B, provides a computational strategy to use e ∈ F (for e : B and F : Finset B).

6.3 Linking Back to My Code

DecidableEq can be derived on Point, Line, and Circle thanks to their structure.

```
structure Point  where
  x : ℚ
  y : ℚ
deriving DecidableEq
```

Listing 6.4: My Point

Rat has DecidableEq (6.2.2). From this we are able to derive a procedure which determines equality between individual fields because all of Point's fields are instances of Rat. Suppose p q : Point; we may derive a procedure to determine p.x = q.x and p.y = q.y. Thus DecidableEq derivable for Point.

```
structure Line  where
  m : ℚ
  c : ℚ
  vert : ℚ := 0
  isvert : Bool := false
deriving DecidableEq
```

Listing 6.5: My Line

An instance of DecidableEq Bool is derived in section 6.2.2. Hence, for l j : Line an instance of DecidableEq Line is derived by the algorithms that decide if l.m = j.m, l.c = j.c, l.vert = j.vert and l.isvert = j.isvert. All of these equalities are decidable, these individual procedures are deducible, We can derive DecidableEq on Line 6.2.2.

```
structure Circle where
  a : ℚ
  b : ℚ
  r : ℚ
deriving DecidableEq
```

An instance of DecidableEq Circle is derivable. For c k : Circle, as before, the procedure exists because the there is DecidableEq on the fields' type.

6.3.1 Finset Result

Instances of DecidableEq for Point, Line, and Circle are derivable. Suppose F_p : *Finset Point*, F_l : *Finset Line*, *and* F_c : *Finset Circle*. Let p : *Point* , l : *Line*, *and* c : *Circle*. It is now the case that the following are decidable propositions $p \in F_p$, $l \in F_l$, *and* $c \in F_c$. Lean is able to infer computational processes to evaluate these propositions in proof.

Chapter 7

Conclusion and Results

In this chapter I will review the results of my code and evaluate the challenges I faced.

7.1 Results

I used structures to describe the points, lines and circles. I was able to derive DecidableEq on my structures because their fields had only Bool or Rat types. I used the algorithms, described in chapter 2, to create functions to find the point(s) of intersection between any drawings. In order to create a function to represent the concept of a points being constructible: I created the following functions:

1. linexline, this function calculated the point of intersection between two lines.
2. linexcircle, this function calculated the point(s) of intersection between a line and a circle.
3. circlexcircle, this function calculated the point(s) of intersection between two circles.
4. points_fom_lines_from_points, this function calculated the set of all possible lines from the inputted set of points and then returned the set of the intersections between all of those lines.
5. points_from_circles_from_points, this function calculated the set of all possible circles from the inputted set of points and then returned the set of the intersections between all of those circles.
6. points_from_linexcircle_from_points, this function calculates the set of all possible lines and the set of all possible circles from the inputted set of points. Then it returned the set of the intersections between all line, circle pairs from the two previously caculated sets.
7. one_step_construction, this function combines the previous functions to find the set of all intersection between all drawings from points in the inputted set.
8. n_step_construction, this function has input parameter n : Nat and iterates over one_step_construction to create the set which encompasses all constructible points after n iterations.
9. **constructible**, this function creates the proposition which defines conductibility of a natural number n. Being that a number n is constructible if there exists a k : Nat were the point (n,0) is a member of n_step_construction (k).

I used a logical structure presented in chapter 2 and aggregated my functions to recursively build the desired sets of points. These functions were accurate, efficient, and made the conceptual aspects of the project achievable and clear. In addition, I fulfilled the primary objective of the project, as described in chapter 1.4. I gaining a deep understanding into Lean and the logic behind computability (6.1.3), constructibility (4.2), type theory (4.1),and decision algorithms (6.1).
Finally, the approach taken to both recognise and resolve the challenges along the way was especially successful. A lot of the obstacles were due to nuances of using theorem provers, and so a strength of the project was in how I found these and re-adapted to continue on a fruitful course of discovery.

7.2 Conclusion

Proving the final theorem in Lean was made expeditiously easier through the use of Finset membership (6.2.2), I was able to effectively automate and evaluate the membership of points from other sets of points. The use of Finset and evaluating its membership relied on a type which could derive DecidableEq. Decidable equality on Rat is easily derived by Lean (6.2.2). The type Rat supports the construction of all terms of Point, Line and Circle needed for the proof (3.1). DecidableEq is derivable on Point, Line, and Circle for this reason. Suppose p : Point, l : Line, and c : Circle. Then we have that $p \in FinsetPoint$, $l \in FinsetLine$, and $c \in FinsetCircle$ are all decidable propositions. These are the reasons why Rat was chosen for the formal proof.

Chapter 8

Critical Evaluation

In this chapter I will evaluate the project's technical approach, criticize my results, describe the areas of improvement and comment on the possibility for future projects.

8.1 Primary Result

My project's primary result was to formalize the constructibility of the natural numbers under Euclidean constructions.

8.1.1 Analysis of Results

The primary result was a somewhat trivial proposition that proposed challenges pertaining to DecidableEq. This lead me to endeavour on a deeper understanding about theorem provers, the algorithmic logic behind them, and the type theory used. Some examples include: non-computable terms cannot be used in constructible proofs because they do not terminate, how decidability is used in constructive maths to provide a transition to constructive logic, decidability of membership of Finset depends on the underlying type of the set. My understanding is also made evident by the quality and structure of proving my primary result. I simplified and deconstructed the primary theorem into it's necessary components to provide clear and concise goals to work towards the final proof.

8.1.2 Limitations

The framework and structure of my code in generating the necessary functions leaves room for improvement. Poor application of programming principles lead to poor adaptability for updates and advancements in my knowledge through the project. Such as the lack of an API to avoid unnecessary restructuring and programming. In order to apply Construido to future proofs, a lot of re-programming must take place. Additionally, I was this some of this project's novelty was limited by the fact that I had only proved that the natural numbers were constructible.

8.1.3 Personal Reflection

I have learnt lessons on decidability and it's possible applications of constructive logic through the pairing between DecidableEq and Finset. I am immediately ready to apply these lessons to proving that the integers are also constructible. Given the appropriate creation of Construido, I would be ready to apply these lessons to prove that $\mathbf{Q}$ are constructible under Euclidean constructions. I would do this by ensuring that a proper notion of equality exists on Construido, meaning I would have to figure out some kind of unique normal form representation.

8.2 Results of my Guide

Practically, I intend for this dissertation to provide as an effective guide to help the reader avoid using the incorrect types in this situation. Before embarking on proving a theorem, it is advised to understand the difference between constructive and classical logic and understand which one applies to your proof.

8.2.1 Limitations

The theory and logic behind theorem provers is extensive and challenging, yet it is easy to solve some theorems using the automation tools provided. For this reason, one could postpone learning key concepts that limit future achievements. Thus a limitation of the guide is that even though I have explained the challenges I faced, many unexpected complications could arise in using theorem provers. This guide is by no means an extensive overview on the whole theory behind theorem provers.

A lot of the theory behind theorem provers is restricted behind some understanding of lambda calculus, algorithmic logic and type theory. For this reason, this guide is also limited by the difficulty in explaining some topics to readers without the necessary background knowledge.

Bibliography

[1] GitHub - 18oscarm/Euclidean-constructions: The formalisation of Euclidean constructions in Lean 4 — github.com. https://github.com/18oscarm/Euclidean-constructions.git. [Accessed 09-05-2024].

[2] Init.data.float, 2024. [Online; accessed 30-April-2024]. URL: https://leanprover-community.github.io/mathlib4_docs/Init/Data/Float.html#Float.

[3] Mathlib, 2024. [Online; accessed 30-April-2024]. URL: https://leanprover-community.github.io/mathlib4_docs/.

[4] John L. Bell. *Higher-Order Logic and Type Theory*. Cambridge University Press, March 2022. URL: http://dx.doi.org/10.1017/9781108981804, doi:10.1017/9781108981804.

[5] Benjamin Bold. *Famous problems of geometry and how to solve them*. Dover Publications, 1982.

[6] Paola Bonizzoni, Vasco Brattka, and Benedikt Lowe. *The nature of computation. Logic, algorithms, applications 9th Conference on Computability in Europe, CIE 2013, Milan, Italy, July 1-5, 2013. proceedings*. Springer, 2013.

[7] George Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. Cambridge University Press, 2007.

[8] David A. Brannan, Matthew F. Esplen, and Jeremy J. Gray. *Introduction: Geometry and Geometries*, page 1–4. Cambridge University Press, 2011.

[9] Mario Carneiro. The type theory of lean, 2019. Master thesis. URL: https://github.com/digama0/lean-type-theory/releases.

[10] Gregory. J. Chaitin. *Algorithmic Information Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[11] B. Jack Copeland. *The essential turing: Seminal writing in Computing, Logic, philosophy, artificial life plus the secrets of Enigma*. Oxford University Press, 2004.

[12] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. URL: https://www.sciencedirect.com/science/article/pii/0890540188900053, doi:10.1016/0890-5401(88)90005-3.

[13] David A. Cox. *Galois Theory*. John Wiley Sons, 2012.

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015. URL: https://doi.org/10.1007/978-3-319-21401-6_26, doi:10.1007/978-3-319-21401-6_26.

[15] Israel M. Gelfand and Tatiana Alekseyevskaya. Geometry. 2020. URL: https://dx.doi.org/10.1007/978-1-0716-0299-7, doi:10.1007/978-1-0716-0299-7.

[16] Donald Gillies. *Frege, Dedekind, and Peano on the Foundations of Arithmetic (Routledge Revivals)*. Taylor and Francis, 2013.

[17] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991. URL: https://doi-org.bris.idm.oclc.org/10.1145/103162.103163, doi:10.1145/103162.103163.

[18] Richard O. Hill. Introduction to linear equations and matrices. *Elementary Linear Algebra*, Not available:1–80, undefined 1986. URL: https://dx.doi.org/10.1016/b978-0-12-348460-4.50004-8, doi:10.1016/b978-0-12-348460-4.50004-8.

[19] Soonho Kong Jeremy Avigad, Leonardo de Moura and with contributions from the Lean Community Sebastian Ullrich. Theorem proving in lean 4. [Online; accessed 30-April-2024]. URL: https://leanprover.github.io/theorem_proving_in_lean4/.

[20] Soonho Kong Jeremy Avigad, Leonardo de Moura and with contributions from the Lean Community Sebastian Ullrich. *Theorem Proving in Lean*. 2014. [Online; accessed 30-April-2024].

[21] Lean Prover Community. About Lean. https://lean-lang.org/about/. [Online; accessed 30-April-2024].

[22] Lean Prover Community. Lean Prover Community. https://leanprover-community.github.io/. [Online; accessed 30-April-2024].

[23] S. C. Malik. *Principles of Real Analysis*. New Academic Science Limited, 2011.

[24] Eli Maor. Princeton University Press, 2010. URL: https://ieeexplore-ieee-org.bris.idm.oclc.org/document/9453307.

[25] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73 Proceedings of the Logic Colloquium*, pages 73–118. Elsevier, 1975.

[26] Rob Nederpelt and Herman Geuvers. *Type theory and formal proof*, Nov 2014. doi:10.1017/cbo9781139567725.

[27] F. Pfenning and C. Paulin-Mohring. *Inductively defined types in the calculus of Constructions*. Carnegie-Mellon University. Department of Computer Science, 1989.

[28] Michael Potter. 21Collections. In *Set Theory and its Philosophy: A Critical Introduction*. Oxford University Press, 01 2004. arXiv:https://academic.oup.com/book/0/chapter/354159909/chapter-ag-pdf/44417660/book_41751_section_354159909.ag.pdf, doi:10.1093/acprof:oso/9780199269730.003.0003.

[29] Michael Sipser. *Introduction to the theory of Computation*. Cengagae Learning, 2013.

[30] Boro Sitnikovski. Introduction to dependent types with idris. 2023. URL: https://dx.doi.org/10.1007/978-1-4842-9259-4, doi:10.1007/978-1-4842-9259-4.

[31] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier Science Limited, 2006.

[32] Alfred Tarski and Alfred Tarski. *Undecidable theories*. Elsevier, 1953.

[33] The mathlib community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381, 2020. doi:10.1145/3372885.3373824.