Formalization of Hilbert's Geometry in Lean 4

Hankai Chen

A thesis submitted in partial fulfillment for the
degree of the Bachelor of Science in Computer Science

Supervised by :
Howard Straubing
Department of Computer Science

May 15, 2024

Contents

Abstract

Long mathematical proofs written in natural language frequently contain ambiguous definitions. It is also difficult and time-consuming to verify by hand that a proof is correct, making such arguments prone to errors. Formal verification systems are softwares that rely on computational methods to ensure the correctness of proofs of mathematical claims. Lean 4, developed by Microsoft Research, is one such theorem prover, with a kernel based on dependent type theory. The programming language aims to bridge the gap between interactive and automated theorem proving by providing an integrated development environment.[1] Theorem provers like Lean 4 ensure correctness and assist users with type checking and construction of well-typed definitions.

This thesis explores the usage of Lean and proposition-as-type framework, it is also a learning experience for those with no prior knowledge of typed systems. We introduce theorem-proving features of Lean 4 and how to define terms and construct proofs. As an application, we comment on the difficulties of transcribing a formal system from natural language to Lean, through formalizing portions of Hilbert's Foundations of Geometry.

Acknowledgements

I want to thank Professor Howard Straubing for supervising this project and starting this learning journey with me, I also want to thank Professor Amittai Aviram for referring me to Professor Howard, and Professor Carl Mctague for his guidance and knowledge in functional programming. I would also thank my father for inspiring my interest in formal systems.

Introduction

Theorem provers are formal systems for reasoning about mathematical objects and their relationships. Theorem-proving software is used to reduce proof checking for the users and develop a common language between computer programs and mathematical ideas. The particular theorem-proving system we explore, Lean 4, relies on the results of Calculus of Inductive Constructions [2], an extended theory of typed systems. Lean 4, is not only a proof assistance but is also capable of automated theorem proving. There has been developing research in automated theorem proving using statistical models, and this idea was emphasized by John McCarthy: "The combination of proof-checking techniques with proof-finding heuristics will permit mathematicians to try out ideas for proofs that are still quite vague and may speed up mathematical research." [3] Some notable automated theorem provers are *Otto* [4] and *ACL2* [5] . There has been development in recent years of automated theorem proving using transformer model from OpenAI[6]. The model used in this paper found new and shorter proofs for existing theorems in their math library. It demonstrates the potential of theorem provers utilizing deep learning systems to generate formal mathematics. The motivation for this project is to learn the foundations of interactive theorem proving to prepare for learning automated theorem proving in the future.

Introduction to Lean 4

The introduction of Lean 4 in this paper focuses on the necessary parts of interactive theorem proving for formalizing Hilbert's axioms. We will be referencing Hilbert's Foundations of Geometry [7] for the translation. The first half of this thesis will introduce the basic syntax and the proposition-as-type framework, and the latter half presents an approach for transcribing Hilbert's axioms and theorems. The source code for this project is on Github.

2.1 Basics

2.1.1 Terms and Types

Lean aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and support both mathematical reasoning and complex systems. [8]. The extension to a typed system equips Lean with types bound to terms. Every term in Lean has a type which are first-class values that denote the kind of the term, for example, the type of 1 is *nat*. Every term has a type, and every type also has a type. To check the type of a term, we can use the #check command to display its type in the vscode live server window and use #evals to evaluate, or compute, an expression.

```
#check 1 --- Nat
#check True --- Prop
#check true --- bool
#check 1 + 1 = 2 -- Prop
#eval 2 * 5 --- 10
```

What makes type system powerful is that we can construct types from existing types. We can construct product types with the symbols $- >$. For example:

```
#check Nat -> Bool -- Type
#check True -> False -- Prop

--same
#check Nat -> Nat -> Nat
#check Nat ->(Nat -> Nat)
```

2.1.2 Definitions

We use the *def* keyword to define a constant followed by its definition. Below we defined the *One* constant of type *Nat* and 1 as its definition.

```
def One : Nat := 1
```

Definition can also parameterize terms:

```
def Sum (a : Nat) (b : Nat): Nat := a + b
```

One can also define functions in Lean. Below we define a function that takes in a natural number n and construct the successor of n. The right arrow represents the return value of the function.

```
def plus_one : Nat := fun n:Nat => n+1
#eval plus_one 5 --- 6
```

2.1.3 Complex Definitions

Pattern matching and recursion are power keywords for constructing complex definitions. Pattern matching allows the user to specify the return values for different constructors, however, only closed recursions are legal in Lean, and the system will signal an error if not every recursive step is strictly decreasing. We can specify an explicit type for each parameter, and if parameters are implicit, the elaborator tries to infer the function body's type. [9] Below, we define the factorial function that matches the natural number

n to two cases, either n is 0, in which case the function returns 1 and terminates, or n has the form $(m+1)$, in which case we enter the recursive step. Even though this feature is not used during the formalization of geometry, this demonstrates the extend of Lean's ability to define complex mathematical notions.

```
def factorial (n:Nat) :=
  match n with
  | 0   => 1
  | m+1 => (m+1) * factorial m
```

Another example of pattern matching is the definition of *or* we implement here. Note that a and b are type $Bool$, not to be confused with $Prop$. The subtle difference here is that $Bool$ is a decidable type, in contrast with impredicative $Prop$, this has implications on what kinds of functions we are allowed to define with $Prop$, but it will not restrict us given the scope of this project.

```
def or (a b : Bool) :=
    match a with
    | true => true
    | false => b
```

2.1.4 Forming Proofs

To form proofs, we can use the *example* keyword and state an expression, followed by its definition. Lean's kernel relies on the results of Curry Howard correspondence to ensure the correctness of mathematical statements, by establishing equivalence between proof systems and computational models [10]. In other words, providing proofs for theorems is equivalent to constructing well-typed functions for programs.

```
example : factorial 3 = 6 := by
    unfold factorial
    unfold factorial
    unfold factorial
    unfold factorial
    simp
```

The *by* keyword enters us into tactic mode, which allows us to write tactics to manipulate proof states or complete proofs. The $unfold$ tactic expands the definition of $factorial$, and matches it with one of its cases, while the *simp* tactic solves expression once terms are fully expanded.

Instead of writing *example*, we can also write *theorem* or *lemma* for the same purpose. *rw* tactic changes our goal state by matching either side of an equivalent statement, the

theorem below is the exact one provided by the imported Mathlib library, a commonly used library for working with mathematics in Lean that provides mathematical types as well as useful tactics. One such tactic is *linarth*, a powerful tactic for proving properties of arithmetic.

```
theorem Add_comm_ex (a : Nat) : a + b = b + a := by
    rw [Nat.add_comm]

lemma leq_lemma (a b : Nat) : a ≤ a + b := by
    linarith
```

2.2 Proposition as Type

The primary type we will be working with is *Prop*. *Prop* in Lean is a kind of type that classifies evidence of its truth, and it can be interpreted as the universe of propositions. At the bedrock of the typed system, there is an infinite number of *Types* at the same degree, and *Prop* resides in this base level at the system, this results in the type being impredicative meaning types can quantify themselves, which means we can form more expressive statements of type *Prop*, also leads to inconsistency in our statements. [8]

We are equipped with logical operations like conjunction, and disjunction, as well as existential and universal quantifiers. Note the distinction between the term $P : Prop$ and $p : P$: the term p is understood as evidence for the proposition P.

```
#check And      -- Prop → Prop → Prop
#check Or       -- Prop → Prop → Prop
#check Not      -- Prop → Prop

example (P : Prop) (ph : P): P ∧ P := by
  constructor
  apply ph
  apply ph

example (P : Prop) (p : P): P ∨ ¬ P := by
  constructor
  apply p

example (P Q: Prop) (p:P) (ph : P → Q): Q := by
  apply ph p
```

In Lean, mathematics can be viewed both as pure computation or be described abstractly and masks the computational detail. This allows users to obtain a better conceptual understanding while not being distracted by the algorithmic information. While Lean accepts both functions and propositions as definitions, this affects "admitting mathematical theorems that are simply false on a direct computational reading"[1]. In Lean, *Prop* is an impredicative type, meaning terms of the type can be undecidable notions, and one is allowed to form well-typed statements that are not logically sound. *Prop* would be the primary type used for the transcription of geometry, and we need to ensure not to introduce false axioms that result in a logical explosion. How does Lean define falsity? False is a term of type Prop, and negation is defined as a function that takes in a term of type Prop and returns False. Logical explosion can be defined as every proposition is True following from False. We use the *absurd* keyword to assume the negation of a hypothesis, which allows us to prove the statement using contradiction.

```
example (p q : Prop) (hp : p) (hnp : ¬p) : q := by
  absurd hnp
  apply hp
```

2.3 Axioms

In addition to forming definitions and proofs, we can also introduce axioms with the *axiom* keyword followed by an expression. We have previously noted the possibility of introducing false axioms, below is one such example. The statement says that there exists a term of type A such that it is not equal to any term of the same Type. This is obviously false because it contradicts the reflexive property $A = A$. The result of this axiom allows us to prove any false statements.

```
axiom false_axiom : ∃ A : Type, ∀ B : Type, A ≠ B

theorem false_is_true : False = True := by
  cases' false_axiom with A B
  absurd B
  simp
```

We now have the necessary tools to construct a system to facilitate our transcription of geometry.

Formalization of Geometry

3.1 Past Formalizations of Geometry

One of the earliest works of axiomatized geometry was Euclid's Elements [11]. Without denying the importance of the Elements, by the end of the nineteenth century, the common attitude among mathematicians and philosophers was that the appropriate logical analysis of geometric inference should be cast in terms of axioms and rules of inference. This view was neatly summed up by Leibniz more than two centuries earlier:

> " ... it is not the figures which furnish the proof with geometers, though the style of the exposition may make you think so. The force of the demonstration is independent of the figure drawn, which is drawn only to facilitate the knowledge of our meaning, and to fix the attention; it is the universal propositions, i.e. the definitions, axioms, and theorems already demonstrated, which make the reasoning, and which would sustain it though the figure was not there"[12]

An example of the work's reliance on diagrams is the proof of the first postulate in the Elements, which describes the construction of an equilateral triangle through the construction of two intersecting circular lines, however, no claim supports the fact that this construction of circles yields intersecting points.

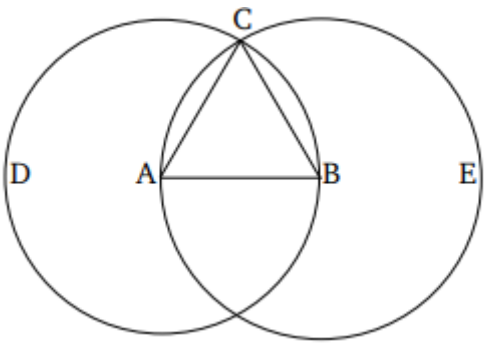

FIGURE 3.1: Proposition 1 of Euclid's Elements

This paper that examines the formal system of Euclid's Geometry [13] points out that occasionally, important details are only represented in the diagram and not the text. They comment on the implicit treatment of the line bisecting segment in *Proposition*10. "Bisect" the segment ab expects d to lie on the same line as a and b, and to lie between a and$b$, and the length of the segment ab is equal to the length of the segmentbd. However, only the last part of the claim is made explicit in the proof text; the other two facts are implicit in the diagram.

Proposition I.10. *To bisect a given finite straight line.*

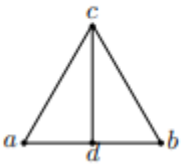

FIGURE 3.2: Proposition 1 of Euclid's Elements

Foundation of Geometry, written by David Hilbert in the late 19th and early 20th centuries, aims to axiomatic geometry in a manner that separates its formalization from diagrams. One of the distinguishing features of Hilbert's geometry is its flexibility. By modifying certain axioms or adding new ones, one can create different geometries with distinct properties. For instance, by altering the parallel postulate, one can derive non-Euclidean geometries, such as hyperbolic or elliptic geometry. There have also been past attempts to formalize Hilbert's Geometry. The first formalization using the Coq proof assistant[14] was proposed by Christophe Dehlinger, Jean-Fran¸cois Dufourd, and Pascal Schreck [15] . There has also been a formalization in Isabelle [16]. These sources serve as a good benchmark for this project's investigation. The most modern counterpart to Hilbert's geometry is Tarski's geometry. Unlike Hilbert and Euclid's system, Tarski worked non-constructively in first-order theory, as a result, it is more concise and elegant, but also harder to translate into constructive language. There have been attempts

to translate Tarski's axioms into a constructive language, and the GeoCoq [17] project provides the translation in Coq, another functional programming language. Out of the three past attempts, the Hilbert system contains the most axioms and contains the richest description. We will focus our translation on the first three groups of axioms and investigate the process of translating Hilbert's system.

3.2 Axioms of Incidence

3.2.1 Definition of Point and Line

Hilbert introduces points, lines, and planes as three sets of distinct objects. For our purpose, we omit planes and introduce the Point and Line as primitive types, and we will be working with elements of line geometry and elements of plane geometry.

```
def Point := Type
def Line := Type
```

3.2.2 Point Line Relations

We define the relation of point and line incidence as a statement of type Prop. We use the axiom keyword here because we want the relation to be of type *Prop* so it satisfies the parameter of logical operators. Our initial approach defines the relation as a *True* statement of type Prop, but this results in logical explosion since it allows us to construct incidence relation for any points and lines.

```
axiom lies_on_L (a : Point) (l : Line) : Prop
```

3.2.3 Axioms Translation

3.2.3.1 Axiom I.1

I.1: *"For any two points there exists a straight line passing through them."*

```
axiom line_exists{a b : Point} (nab : a ≠ b) :
∃ l : Line, lies_on_L a l ∧ lies_on_L b l
```

The above line of code is a direct translation of the axiom. Note that the translation does not mention that the two points are distinct, which we need to explicitly state in our translation. In addition to the translation, we also define a function that takes in the

same parameter and returns the type Line. This relation "axiom" simplifies the proving process and allows us to refer to a line from two distinct points instead of using the axiom to produce a line and its hypothesis.

```
axiom line {a b: Point} (nab : a ≠ b) : Line
```

3.2.3.2 Axiom I.2

> **I.2:** *"There exists only one straight line passing through any two distinct points."*

```
axiom same_line {l : Line} {a b : Point} {nab: a ≠ b} {h: lies_on_L a
l ∧ lies_on_L b l} : l = (line nab)
```

To simplify the statement, an equivalent statement can be constructed by stating the unicity of lines, which can be established if there is an axiom to determine when two lines are equivalent. Therefore, two lines are equivalent if two distinct points lie on both lines. Note that this axiom can be used for both representations of a line from axiom $I.1$.

3.2.3.3 Axiom I.3

> **I.3:** *At least two points lie on any straight line. There exist at least three points not lying on the same straight line.*

```
def collinear := fun a b c: Point => ∃ l: Line, (lies_on_L a l) ∧
(lies_on_L b l) ∧ (lies_on_L c l)

axiom points_on_line (l:Line) : ∃ a b : Point, (lies_on_L a l) ∧
(lies_on_L b l)

axiom non_collinear : ∃(a b c: Point), a ≠ b ∧ a ≠ b ∧ b ≠ c ∧ ¬
(collinear a b c)
```

The sentence can be broken up into two parts: the existence of points on a line and the existence of three distinct points that are not collinear. This axiom requires the definition of collinear relation, which we define as a function that takes in three points and returns a proposition that asserts the existence of a line on which the three points lie. Note that *collinear a b b* and *collinear a b b* are always $True$, and noncollinear points are always distinct.

Alongside these axioms, we also proved lemmas that describe permutation and transitive properties of collinear and noncollinear points. Because collinear points do not assume distinct points, the application of collinear transitivity requires the uniqueness of at least two out of four collinear points. We also explore some basic relationships between betweenness and collinearity, one can prove that betweenness implies collinearity, and non-collinearity implies non-betweenness. Here are some examples below, and we omit their proofs here.

```
lemma col_perm1 (A B C: Point) :
(collinear A B C) → collinear A C B

lemma col_trans {A B C D : Point}(h1 : collinear A B C) (h2 :
collinear B C D) (nBC : B ≠ C):
collinear A B D ∧ collinear A C D:= by

lemma between_collinear {a b c : Point} :
  between a b c ->
  ( a ≠ b ∧  a ≠ c ∧ b ≠ c ∧ collinear a b c )

lemma not_collinear_imp_not_between {a b c : Point} :
  ¬collinear a b c -> ¬ between a b c := by
```

3.2.4 Theorem 1

"Two lines either have one point in common or none at all."

```
theorem Theorem1 (l1 l2: Line) (A B: Point):
  (l1≠l2) → (lies_on_L A l1) → (lies_on_L B l1) → (lies_on_L A l2)
→ (lies_on_L B l2) → A = B := by
  intro h1 h2 h3 h4 h5
  by_contra h'
  have k : l1=l2 := by
    apply unique_line_from_points h' h2 h3 h4 h5
  exact h1 k
```

We present the translation and proof of Theorem 1 above. Parts of Theorem 1 describe the unique intersecting point between two lines, which we can state by saying if point A B both lie on lines $l1$ and $l2$, then $A = B$. This theorem is proved by contradiction from the assumption that two lines are distinct. The lemma *unique_line_from_points* is an application of Axiom I.2 and proves that $l1$ and $l2$ are indeed the same line, in which case we reach a contradiction. The *have* keyword asserts a hypothesis in the current context and asks the user to prove that hypothesis.

3.3 Axioms of Order

3.3.1 Between Relation

```
axiom between : Point → Point → Point → Prop
```

The second group of axioms introduces the between relation, we follow the same pattern and use product type to specify the parameters of the relation. We define a relation axiom that takes in three points and returns a *Prop*. The axioms of this group focus primarily on between relations, which expresses the ordering of points on a line.

Hilbert also introduces the definition of segment and defines it as the set of points that lies between two endpoints. While the Pasch Axiom (II.4) mentions segments, we do not make use of its definition and simplify its notion to just between relations. Moreover, Hilbert's definition of segment requires second-order logic because of the quantification over sets, which we intend to avoid. For this group of axioms, the necessary notion requires us to define points lying on the same side and on different sides of a plane when proving Theorem 8. The construction of a ray, or half-lines can be realized when formalizing Axioms of Congruence, by defining what it means for two points on a line to be on the same side and on different sides of the ray.

3.3.2 Axioms Translation

II.1 *"If a point B lies between points A and C, B is also between C and A, and there exists a line containing the distinct points A, B, C."*

```
axiom between_sym {a b c : Point} : between a b c → between c b a

axiom between_distinct_points {a b c : Point} :
  between a b c →
  a ≠ b ∧ a ≠ c ∧ b ≠ c

axiom between_line_exists {a b c : Point}:
  between a b c →
  ∃ l : Line, lies_on_L a l ∧ lies_on_L b l ∧ lies_on_L c l
```

This axiom can be described in three parts. The first part can be stated as *between a b c* implies *between c b a*. The second part states that if b is between a and c, then a b c are all distinct from one another. This also implies that if point b lies in segment a c, then a b c are distinct, and the endpoints of a segment are not in the segment. The third part

asserts if we have Between *a b c*, then there exists a line through all three points, which is precisely the collinear relation.

3.3.2.1 Axiom II.2

II.2 *"If A and C are two points, then there exists at least one point B on the line AC such that C lies between A and B."*

```
axiom between_extend {a b : Point} (h: a ≠ b):
  ∃ c: Point , between a b c
```

This axiom states that given two points, there exists a third point on the line that is extended from the previous two points. The application of this axiom allows us to construct a betweenness relationship by asserting the existence of an extended point, this axiom is applied many times in Theorem 3 for the realization of intersecting points.

3.3.2.2 Axiom II.3

II.3 *"Of any three points situated on a line, there is no more than one which lies between the other two."*

```
axiom exist_unique_point_between {a b c : Point} :
  between a b c  →
  ¬ between a c b ∧ ¬ between b a c
```

This statement can be translated using logical connective. Moreover, the reverse implication, with an additional condition that *abc* are collinear points, is stated in Theorem 4. It is interesting to note that in the Geocoq approach of Tarski to Hilbert's formalization [17], the reverse implication is baked into this axiom. Whereas in the Foundation of Geometry, Hilbert provides proof for the statement.

3.3.2.3 Axiom II.4

II.4 *"Let A, B, C be three points not lying in the same line and let a be a line lying in the plane ABC and not passing through any of the points A, B, C. Then, if the line passes through a point of the segment AB, it will also pass through either a point of the segment BC or a point of the segment AC."*

```
    axiom Pasch {A B C: Point} {l: Line} :
    ¬(collinear A B C)→ (intersect_line_segment l A B)
    → ¬(lies_on_L C l) → ((intersect_line_segment l A C) ∨
(intersect_line_segment l B C))
```

This axiom is also known as Pasch's axiom, it is a statement that is not provable nor disprovable from Euclid's Axioms. As stated previously, we omit the plane and the definition of segments. In addition, we define the line segment intersection relation that takes in a line and two points and asserts that there exists an intersection point. The unique line segment intersection can be proven as a lemma. Note the statement asserts that if a line intersects one side of a triangle, it also intersects one of the other two sides. But to prove that a particular line that enters through one side of a triangle exits from a particular side, we need to prove that it does not exit through other side.

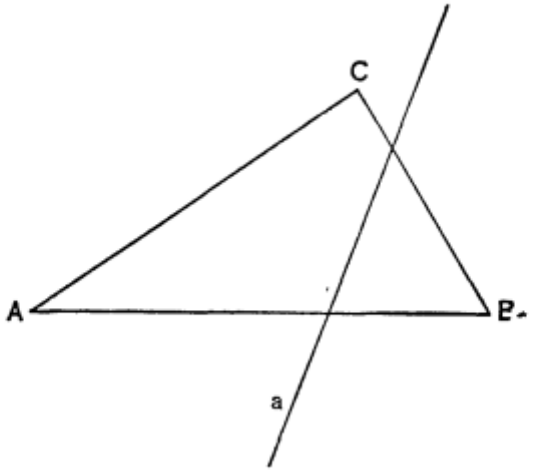

FIGURE 3.3: Pasch's Axiom

3.3.3 Theorem 3

"For two points A and C there always exists at least one point D on the line AC that lies between A and C."

```
theorem Theorem3 (A C : Point): A ≠ C → ∃D : Point, between A D C
```

This theorem is proved through the construction of a triangle using the noncollinearity axiom and betweenness extension axiom, followed by an application of the Pasche axiom yields the intersecting point that lies between A and C. While the book relies on pictorial diagrams for the application of Pasch's axiom, it is not obvious that the second

intersecting point does not lie on the opposite side of the triangle, and that it is not the case that the line goes through point A or C. These conditions require proof through contradiction for the unique application of the Pasch axiom. This theorem also asserts that segments are non-empty sets because there is at least one point in a segment.

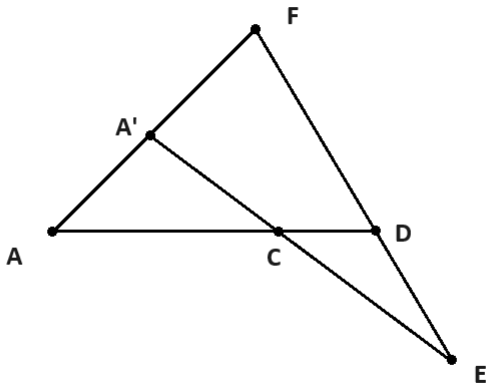

FIGURE 3.4: Theorem 3 Construction

3.3.4 Theorem 4

"Of any three points A, B, C on a line there always is one that lies between the other two."

```
theorem Theorem4 ( A B C  : Point)
  (nAB: A ≠ B) (nAC: A ≠ C) (nBC: B ≠ C)
  (colABC: collinear A B C ):
   (between A C B) ∨ (between A B C) ∨ (between B A C)
```

This theorem can be translated as given the case of three distinct collinear points, one point is between the other two. Proving this theorem requires the longest proof out of all the theorems. Due to the repeated application of Pasch's axiom, we are required to introduce additional hypotheses on point distinctness and noncollinearity to complete the proof through contradiction. The length of this proof suggests that it is necessary to introduce projection from a defined geometric structure to better fulfill the conditions for Pasch's axiom.

3.3.5 Theorem 8

"Given a line that bisects the plane, for any point not on the line, it determines a point on the line such that if they are on the same side, there exists a point of intersection between the segment formed from the two points and the bisecting line, and if they are not on the same side, the line and segment do not intersect."

This theorem introduces the notion of a plane bisecting line, and the definition of points being on the same side and different sides of the line. We represented the cases by defining two points as being on the same side as there does not exist a point on the bisecting line that is between the segment formed by the two points. The case that two points are on the same side of the bisecting point is exactly the *intersetct_line_segment* relation.

```
def points_same_side (l:Line)(a b: Point) :=
¬ lies_on_L b l ∧ ¬ lies_on_L a l ∧ (forall c: Point, lies_on_L c l →
¬ between a c b )

theorem Theorem8_same_side (l:Line)(a b: Point):
points_same_side l a b -> ¬ intersect_line_segment l a b
```

To prove this theorem, we only need to prove that the points on the same side do not have an intersecting point with the bisecting line, then we can prove that all points not on the line fall into either case.

We went a step further and attempted to prove that if two points are on different sides of a third point, then the two points are on the same side.

```
theorem diff_diff_same (l: Line) (a b c: Point) :
intersect_line_segment l a b → intersect_line_segment l a c
→ points_same_side l b c
```

We could not provide proof for the statement, because in the case that points B and C are on the same side, and point A is on a different side from B and C, we apply Pasch's Axiom on segment AB and AC, which requires us to prove that intersection point is not in segment BC. We could not prove this through contradiction by extending the segment BC and applying the unique intersection lemma, since line BC can be parallel to the bisecting line, therefore not intersecting. Because we have yet to introduce the notion of parallel lines, this statement is not provable in such case.

3.4 Axioms of Congruence

3.4.1 Congruence Relation

```
axiom cong (a b a' b' : Point): Prop
```

We define congruence relations on segments and angles as proposition types. Hilbert describes the construction of segments and rays in relation to betweenness, which is introduced in the first group of axioms, but we did not rely on this definition in proving previous theorems. However, the relationship between points and rays and points and segments is explicitly used in axiom III.4, which briefly states that every angle can be constructed on a given side of a given ray in a uniquely determined way. To simplify this axiom, we only define the condition when a point lies in a ray. We also use the endpoints of segments for the parameters of segment congruence, this is similar to the Geocoq Project approach when translating from Tarski to Hilbert [17].

3.4.2 Axioms Translation

3.4.2.1 Axiom III.1

III.1 : *"If A, B are two points on a line a, and A' is a point on the same or on another line a' then it is always possible to find a pointB' con a given side of the line a' through A' such that the segment AB is congruent or equal to the segment $A'B'$. In symbols $AB \cong A'B'$. "*

```
axiom cong_exist (a b o: Point) (l: Line) (nab: a ≠ b) :
lies_on_L o l → ∃ a' b':Point , lies_on_L a' l ∧ lies_on_L b' l ∧
between a' o b' ∧ cong a b o a' ∧ cong a b o b'

axiom cong_permute (a b c d: Point) : cong a b c d → cong a b d c
```

This axiom describes the possibility of constructing congruent segments; interestingly, this axiom does not imply segment equivalence is reflexive or symmetric. The uniqueness of segment congruence is also provided as a theorem that is proven with the aid of axiom III.2, but its proof is not provided in the project, and we only accept its intuitive result. Above we defined the construction of congruence relation as well as its implied permutation property since the order of construction is not specified.

3.4.2.2 Axiom III.2

III.2 : *"If a segment $A'B'$ and a segment $A''B''$ are congruent to the same segment AB, then the segment $A'B'$ is also congruent to the segment $A''B''$, or briefly if two segments are congruent to a third one they are congruent to each other."*

```
axiom cong_transitive1 (a b a' b' a'' b'': Point) :
  cong a b a' b' → cong a b a'' b'' → cong a' b' a'' b''

axiom cong_transitive2 (a b a' b' a'' b'': Point) :
  cong a b a' b' → cong a'' b'' a b → cong a' b' a'' b''
```

This axiom appears to be the transitive property of congruence, but the language suggests that the order of how we apply the segment to this axiom does not matter, therefore we divide the axiom into two cases. These axioms imply symmetry and reflexivity, as well as the "actual" transitive property.

3.4.2.3 Axiom III.3

III.3 : *"On the line a let AB and BC be two segments which except for B have no point in common. Furthermore, on the same or on another line a' let $A'B'$ and $B'C'$ be two segments which except for B' also have no point in common. In that case. if $AB \cong A'B'$ and $BC \cong B'C'$, then$AC \cong A'C'$. "*

```
def disjoint := fun a b c d=> ¬ ∃ p, between a p b ∧ between c p d

axiom add_segment (a b c a' b' c' : Point) :
  between a b c →
  between a' b' c' →
  disjoint a b b c →
  disjoint a' b' b' c' →
  cong a b a' b' ∧ cong b c b' c' →
   cong a c a' c'
```

This axiom describes the additivity of segments. Because the segments that are added together have to be disjointed or do not share any point in common that lies in both segments, we need to introduce the notion as a definition. This axiom is not used for theorems proved in this project, but regardless, we present their translations.

3.4.2.4 Axiom III.4

III.4 : *"Let $Angle(h,k)$ be an angle in a plane α and a' a line in a plane α' and let a definite side of a' in α' be given. Let h' be a ray on the line a' that emanates from the point O'. Then there exists in the plane α' one and only one ray k' such that $Angle(h,k)$ is congruent or equal to $Angle(h',k')$ and at the same time all interior points of $Angle(h',k')$ lie on the given side of a'. Symbolically, $Angle(h,k) \cong Angle(h',k')$. Every angle is congruent to itself, ie. $Angle(h,k) \cong Angle(h,k)$ is always true."*

```
    axiom congA (a b c a' b' c' : Point) : Prop
    axiom angle_reflex (a b c : Point) (h : ¬ collinear a b c): congA a b
c a b c
    axiom angle_comm (a b c : Point) (h : ¬ collinear a b c): congA a b c
c b a

    -- symmetry is also implied
    axiom congA_symm {a b c a' b' c':Point} :
      congA a b c a' b' c' →
      congA a' b' c' a b c

    -- permutation from different construction with rays
    -- implied in the definition
    axiom congA_perm2 {a b c a' b' c':Point} :
      congA a b c a' b' c' →
      congA c b a c' b' a'
```

To break up this axiom into parts, the code above provides the definition of angle congruence and the construction of angle from uniquely determined rays, it is also mentioned that every angle is congruent to itself. Hilbert points out that the construction of angles is dealt with precisely as the construction of segments, and that besides the possibility of constructing angles, uniqueness, transitivity, and additivity can all be proven. By the end of this project, we accept these properties as results. It is also important to note that Hilbert's axiom does not allow a null angle and that there exists at least one point in the interior of the angle. With these in mind, we present a set of axioms that follows a similar approach used in the GeoCoq Project, which defines the existence and unicity of congruence angle through the notion of points lying on the same side of a given half line.

```
def same_side_ray := fun o a b : Point =>
between o a b ∨ between o b a ∨ ( o ≠ a ∧ a = b)

axiom angle_exists :
   ∀ a b c a' b' c',
     ¬ collinear a b c → ¬ collinear a' b' c' →
     ∃ c'', congA a b c a' b' c'' ∧ same_side_plane' c c'' b' a' ∧ ¬
collinear a' b' c''

-- determining angle from ray
axiom angle_rays (a b c x y z a' c' x' z' : Point) :
    congA a b c x y z →
    same_side_ray b a a' → same_side_ray b c c' →
    same_side_ray y x x' → same_side_ray y z z' →
    congA a' b c' x' y z'

axiom angle_unique (a b c a' b' c' p p' : Point):
     ¬ collinear a b c → ¬ collinear a' b' c' →
     congA a b c a' b' p → congA a b c a' b' p' →
     same_side_plane' c' p a' b' → same_side_plane' c' p' a' b' →
     same_side_ray b p p'
```

3.4.2.5 Axiom III.5

III.5 : *If for two triangles ABC and A'B'C' the congruences* $AB \cong A'B'$, $AC \cong A'C'$, $Angle(B, A, C) \cong Angle(B', A', C')$ *hold, then the congruence* $Angle(A, B, C) \cong Angle(A', B', C')$.

```
axiom congA5 {a b c a' b' c':Point} :
  ¬ collinear a b c →
  ¬ collinear a' b' c' →
  cong b a b' a' →
  cong a c a' c' →
  congA b a c b' a' c' →
  congA a b c a' b' c'
```

From the definition, it is implied that every angle is congruent to itself. Therefore we can construct angles from noncollinear points, and angle congruence is reflexive. It is also obvious but not explicitly proven that there for any angle, there exists an angle that permutes the order of ray construction such that two angles are congruent.

3.4.3 Theorem 11

```
theorem Theorem11 (a b c : Point) :
  ¬ collinear a b c →
  cong b a c a →
  congA a b c a c b
```

Theorem 11 is the last theorem we prove for this project. The statement briefly states that the opposite base angles of an isosceles triangle are congruent. The proof follows directly from axioms III.2 and III.5.

Conclusion

In conclusion, theorem provers are vital tools in modern mathematics, offering a formal framework for reasoning about mathematical objects and relationships. We introduce the basic usage of Lean 4 as a proof assistance, as well as working with the proposition-as-type framework. We explore the process and difficulties in translating Hilbert's Geometry in Lean 4, and note the similarities and differences with past formalizations. Some of the challenges in proving Hilbert's postulates involve proving the uniqueness of points and their collinearity. While it can be assumed intuitively and from the diagrams that the points in a certain arrangement are evidently unique, those assumptions often arise from our innate familiarity with the physical planar space. This is most evident when proving Theorem 3 and Theorem 4 because it requires many construction of intersecting points and their implicit non-intersectionality with one of the three sides of the triangle. The application of the Pasch's Axiom is cumbersome, but given the required conditions of collinearity and point uniqueness, a potential solution might be reducing our model to finite projections of points and lines. It might be desirable to define the construction in Theorem 3 and Theorem 4 in general because the relation between the points and lines are known and all possible permutations of collinearity can be inferred. The next direction of this project is incorporating automated theorem proving features into our system, and simplifying our proofs with custom tactics.

Bibliography

[1] Soonho Kong Jeremy Avigad, Leonardo de Moura and Sebastian Ullrich. Theorem Proving in Lean 4. URL https://lean-lang.org/theorem_proving_in_lean4/title_page.html.

[2] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. *HAL*, 11 Dec 2014 2014. URL https://inria.hal.science/hal-01094195/document.

[3] John McCarthy. Computer programs for checking mathematical proofs. *American Mathematical Society. In Proceedings of the Fifth Symposium in Pure Mathematics of the American Mathematical Society*, pages 219–227, 1961.

[4] W. McCune. Otter and Mace2. 2003. URL ttps://www.cs.unm.edu/~mccune/otter/.

[5] ACL2: a computational logic for applicative common lisp. URL http://www.cs.utexas.edu/users/moore/acl2/.

[6] Ilya Sutskever Stanislas Polu. Generative Language Modeling for Automated Theorem Proving. *arXiv*, Sep 7, 2020. URL https://arxiv.org/abs/2009.03393.

[7] David Hilbert. The Foundations of Geometry. *The Open Court Pub. Co*, pages 1–20, 1910.

[8] J. Avigad F. van Doorn J. von Raumer L. de Moura, S. Kong. The Lean Theorem Prover (System Description). *Microsoft Research*. URL https://lean-lang.org/papers/system.pdf.

[9] Sebastian Ullrich Leonardo de Moura. The Lean 4 Theorem Prover and Programming Language (System Description)). *Microsoft Research*. URL https://lean-lang.org/papers/lean4.pdf.

[10] Philip Wadler. Propositions as Types. *University of Edinburgh*, pages 1–4. URL https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf.

[11] Giovanni Euclid, Hypsicles. Euclid's Elements. *Venice : Erhard Ratdolt, 1482-05-25*, 1475–76.

[12] G Leibniz. New Essays Concerning Human Understanding. *La Salle, IL: Open Court Publishing.*, 1949.

[13] John Mumma Jeremy Avigad, Edward Dean. A Formal System For Euclid's Elements. *The Review of Symbolic Logic*, 2009.

[14] Coq development team. The Coq Proof Assistant Reference Manual, Version 8.3. *TypiCal Project. (2010).*

[15] Higher-Order Intuitionistic Formalization and Proofs in Hilbert's Elementary Geometry. *In: ADG'00. Volume 2061 of LNAI., Springer-Verlag*, 2000.

[16] LI Meikle and JD Fleuriot. Formalizing Hilbert's Grundlagen in Isabelle/Isar. pages 319–334, 01 2003.

[17] Julien Narboux Gabriel Braun. From Tarski to Hilbert. pages 89–109, 2012.