Reasoning on Figures of Theoretical Geometry Theorems

by

Eirini Vandorou

Submitted
in partial fulfilment of the requirements for the degree of

Master of Artificial Intelligence

at the

UNIVERSITY OF PIRAEUS

June 2021

Author ....Eirini Vandorou.............

II-MSc “Artificial Intelligence”

June 30, 2021

Certified by ....

Stasinos Konstantopoulos
Research Fellow
Thesis Supervisor

Certified by ....

Angelos Charalambidis
Research Fellow
Member of Examination Committee

Certified by ....

Ioannis Emiris
Professor
Member of Examination Committee

Reasoning on Figures of Theoretical Geometry Theorems

by

Eirini Vandorou

Submitted to the II-MSc "Artificial Intelligence" on June 30, 2021, in partial
fulfillment of the
requirements for the MSc degree

Abstract

One of the most basic problems scientist need and want to solve using computers is, the process of solving mathematical problems. While addition, subtraction and multiplication seem fairly easy calculations to do, as one dives into mathematics and the calculations advance, the problems become more and more thought depleting. In this sense, having a system with the ability to solve mathematical problems varying in discipline and structure, would be nice to have. For example, in geometry in most cases calculations are part base and a combination of functional and logic programming for the implementation. The first step, from our perception, was to create a structure in which we could describe the Euclid's theorems. After that, we used this structure to generate the theorems' figure constructions. Then we generated all possible premises needed for a relevant conclusion, given the construction.

Thesis Supervisor: Stasinos Konstantopoulos
Title: Research Fellow

Acknowledgments

Thank you to my supervisor Dr. Stasinos Konstantopoulos for his undoubtful help and advisory, as well as, Dr. Angelos Charalambidis and Professor Ioannis Emiris. I also want to thank my family and friends for supporting and believing in me. Of course, I wouldn’t be any close to achieving this without my mother, so thank you.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ATP	**A**utomated **T**heorem **P**roving, Automated deduction
FOL	**F**irst **O**rder **L**ogic
LP	**L**ogic **P**rogramming
FP	**F**unctional **P**rogramming

Chapter 1

Introduction

One of the most basic problems scientist need and want to solve using computers is the process of solving mathematical problems. While addition, subtraction and multiplication seem fairly easy calculations to do, as one dives into mathematics and the calculations advance, the problems become more and more thought depleting. To aid this problem humans thought of the idea using a computer to mimic the human brain and do these calculations. In our attempt to mimic the human brain the Artificial Intelligence (AI) field was created.

Through out the years, computer scientists try to solve all the more complex issues, that adequately require all the more mathematics. In this sense, having a system with the ability to solve mathematical problems varying in discipline and structure, would be nice to have. An attempt to creating such a solution is Automated Deduction.

With this thesis we attempt to use parts of this subfield of AI to construct and solve Theoretical Geometry's theorems. The first thing to consider here is creating the instructed figure using the basics shapes of a compass and ruler approach, using a computer instead. Then one would infer conclusions based on the shapes, their attributes and relations between one another, this we named generating facts and made the computer infer them for us. Last but not least, one would used these facts to deduct the proof of a theorem, we did so by using logic and logic programming.

This is a proposed new solution to an existing old problem. Maybe it has become such an old problem, that not many scientist seem to focus on anymore. This in our opinion should change. Automated deduction is an old field with still many perspectives and as new technologies and solving techniques emerge, this old problem can find new solutions.

While not as new as one would expect, is functional programming. We used functional programming to create a proposition-function relationship that can help address with ease the references between propositions of geometry. Thus, potentially cut many branches of the possible evolution of a construction and intrinsically come to the resulting shape and its inferred conclusions. However first we need to step back and examine all the parameters needed to fill the gaps in the definitions of all these.

1.1 Automated Proofs

To start with, a core computer science concept is automating procedures, while a core mathematical concept are proofs. Thus, an core interdisciplinary concept are automated proofs. As natural as that might have come as a conclusion, the challenge of finding such an automated proving system is an extremely difficult task. There can be one or more ways to prove a theorem, however, there might be none as well. Even if we had a suggested proof validating it also an almost equally difficult challenge. However a challenge it might be, scientist are still trying to find a suitable solution, since finding such a solution can help many parts of each science evolve further and aid the better understanding of it fundamentals.

1.2 Motivation

To study a fundamental part of any science we believe it is good practice to start with an even more core part of it. Euclid, the Greek mathematician, is often referred to as "the father of geometry", but we still lack to fully see his view of mathematics. On the other hand, functional and logic programming have been around from the start of computer science and are still a key part of it.

Many mathematicians and computer scientist have claimed that his work requires editing in order to fit in the mathematical logic's strict structure, which is partially true. There is however, a part where we fail to interpret Euclid's logic to the computer.

We considered this as an opportunity for experimentation. An approach that programmatically implements Euclid's thinking -from our perception and understanding. This study aims to implement a system to address the automated deduction problem in the field mathematics, specifically theoretical geometry. This is achieved using Euclid's Elements for the mathematical part base and a combination of functional and logic programming for the implementation. The first step, from our perception, was to create a machine readable representation in which we could describe the Euclid's theorems. After that, we used this representation to generate the theorems' constructions. Then we generated all possible premises needed for a relevant conclusion, given the construction.

1.3 Structure of thesis

Chapter 2 provides the required background for Logic Programming, Euclid's work and others' related approaches on the subject

Chapter 3 presents the proposed method by explaining the three parts of the combinational solution. Specifically, figure construction, valid fact extraction and reasoning with the generated facts.

Chapter 4 proves the validity of the method using the implementation and provides information for expanding from the current state.

Chapter 5 presents a set of conclusions for this work, discussion on the overall approach in relation to other approaches and future directions.

Chapter 2

Background

The following sections give the basic needed knowledge from logic programming, automated deduction and basics of theoretical geometry as seen in Euclid's Elements.

2.1 Automated Deduction

There are two main ways to solve or justify the solution of a problem, the first being through concrete evidence that lead to that solution and the second through assumptions of validity issuing from unreliable sources. In the former, solution is based on tangible evidence emerging from reliable verified sources and a mixture of those sources to make a conclusion on a related topic. The latter is a more common way that includes either unreliable sources or assumptions that are based on one's opinion instead of many's logic. What automated reasoning aims to do is to address the problem using logic. Logical reasoning is used in order to exploit the structure -instead of the content- to make an inference [10]. Putting forward the importance of valid structure, rather than the content meaning, creates a structured logic that is based on solid inference instead of ambiguous assumptions.

Automated deduction, also known as automated theorem proving (ATP), is based on automated reasoning and mathematical logic and serves to prove and/or verify problems and their solutions. There are roughly three things used for ATP. The first part needed are the axioms. Axioms are used to give a detailed description of the current knowledge of the world in which the problem belongs. The second part is the conjecture, which serves as the description of what needs to be solved. While the third part, is automated reasoning that combines this knowledge to logically conclude, to accept or reject a proof and give counter-examples. Thus, ATPs aim to determine if a conjecture is a logical consequence of a set of axioms[20]. Propositional Logic and First-order Logic are two most widely used for automated reasoning.

Propositional Logic is based on Boolean algebra. Boole in 1847 created an algebra, namely Boolean logic, that would structure logic in a way that would assist logic with ease of stating and understanding propositions and the calculations between them, much like mathematical algebra does in mathematics. Variables in this case are some truth-valued elements, meaning their values may be true or false. Logic propositions are then constructed using these truth-value variables and some predefined logical connectives, like "not", "and" and "or" [10]. Calculations between logical propositions were clearly defined. One can use this algebra to form a problem in such a way that it's easy to do calculations and derive conclusions based on the truth values of the variables in the problem.

On the other hand, First-order Logic (FOL) or Predicate Logic, an extension of propositional logic, is also often used for automated reasoning. In first-order logic there are predicates and terms. *Predicates* are used to declare relations between arguments. Arguments vary in number, starting from zero in which case the predicate is considered a propositional variable. Every predicate having more than one argument has the same *arity* as the count of the arguments.There are different approaches to addressing the matter of ATP using first order logic.

While the most obvious application areas for ATPs evolve around mathematics and logic there have been a lot more fields that automated deduction has been and even more that it can be applied [19]. In fact, there is not as much content in the applications of fully automated deduction in the area of mathematics [5]. The reason for that lies in most cases in the fact of numbers are difficult to interpret in most logic programming languages.

Logic programming

Logic programming (LP) is the programming of a computation using a restricted form of resolution [4].LP's aim is to solve goals by systematically searching for a way to derive the answer from the program using resolution. Resolution allows the inference of new propositions from other given propositions and is the primary rule of inference in LP.

There is a variety of LP languages, however, the most widely used is Prolog, the name stands for Programming in Logic. Prolog uses the inductive logic programming logical approach and is a programming language designed to help with solving and/or verifying problems that can be described using first-order logic. In general, inductive logic programming systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge and final descriptions are all described as logic programs[14, 7, 4]. In its essence a program in LP, consists of a set of axioms, which serve to prove a fact, using rules of inference. The set of axioms is a collection of universal truths or facts, expressed in the form of Horn clauses. Prolog, as well, uses Horn clauses and implements resolution via a depth-first strategy and a unification algorithm[18]. To better understand this we will explain some basic semantics of logic. Since logic programming is based on an extension of FOL, A key point to understanding Horn clauses, are propositions. Propositions can be atomic or compound. An atomic proposition is a statement or assertion that must be true or false. the atomic proposition is in its basic form is one word or letter. However atomic propositions can consist of a *functor* and an ordered list of parameters -the order matters- in which case it is called a compound term.

Atomic propositions	:	a
compound term	:	friend(pythagoras, crates)
functor	:	friend
ordered list of parameters	:	pythagoras, crates

TABLE 2.1: Examples of atomic propositions

A compound proposition on the other hand, is when two or more atomic propositions are connected be logical connectors or operators (Table 2.2) .

negation	:	$\neg$
conjuction	:	$\wedge$
disjunction	:	$\vee$
equivalence	:	$\Leftrightarrow$
implication	:	$\subset$ or $\supset$

TABLE 2.2: Logical connectors and operators

Then there is Clausal form, a standard form of propositions that consists of exactly one implication. The left part of the implication is called consequent and is a compound propositions that has only atomic propositions connected with disjunctions. The right part is called antecedent, and is a compound proposition that is formed as well by atomic propositions connected only by conjuctions. The clausal for expresses the fact that if all antecedents are true then at least one of the propositions of consequent must be true. So, a Horn clause is a restricted clausal form, that has zero or one atomic proposition in the consequent. Prolog uses Horn cluases as logical formulas like the following:

$$P : -Q_1, Q_2, ..., Q_n.$$

with P and the Q_i being atomic propositions. The above would translate in a Horn clause in the following way:

$$P \vee \neg Q_1 \vee \neg Q_2 \vee ... \vee \neg Q_n$$

2.2 Euclid's Elements

Automated deduction can be applied to many fields of mathematics, this study focuses on Geometry and more specifically on theoretical geometry, a huge part of which are Euclid's Elements [11]. The most widely known mathematician in geometry is Euclid form Ancient Greece and Euclid's most renowned work are the *Elements*, which he wrote at about c. 300 BC. The Elements consist of thirteen books, the first six being about geometry. In the first book Euclid starts by stating the perquisites for the rest of his books by declaring twenty three definitions, five postulates and nine axioms. After the axioms, the propositions and their proofs begin.

A proposition by Euclid consist of some given statements, the construction part and the proof. The given statements represent the base upon which he will built the construction and then the proof, so as to conclude to the proposition. A basic characteristic of Euclid's Elements is that each proposition once it is proven, it is considered as given knowledge and so it is used inside other later propositions.

Although the Elements where considered flawless for many years, Euclid's work was not as clear concerning the purity of logic, instead there where many times in which he would use non-logically proven assumptions [10]. Whether that was because something came from observation or from assumption. This is where Hilbert[13] firstly stepped in and axiomatized Euclid's geometry by redefining the basis of Euclid's definitions, replacing and removing some axioms concerning the plane. There are also other known axiomatizations of the Elements such as Tarski's [22] and Birkhoff's [6].

2.3 Automated Deduction for Geometry Theorems

In the notion of theoretical geometry more often than not the proof to a proposition needs more than what can be calculated. This is because in most cases, the diagram plays a vital role to the solution. To add on top of that, the diagram is usually constructed given the initial assumptions, and then after the diagram itself provides the additional information needed to create the proof. However, it would be easier to use automated deduction and logic programming on geometry if the theory of geometry was transformed into axioms and then fed to a logic program and the proving fact transformed into an query, as described in Section 2.1. One step in that direction is creating a formal system of the Elements.

As observed the Elements use a form of logic in order to form the proofs, naturally many attempts have been made to create a formal system for them, one of these attempts was by Avigad et al.[1]. They made a formal system to model Books I to IV from Euclid's Elements. In order to address the matter of facts that Euclid takes as observations from the diagram, they set a list of axioms to replace the conclusions derived from the diagram Euclid constructs. These axioms are in their core rules that depict the diagram's purpose and part in the proof. They state that in some cases where Euclid reads geometric relations directly from the diagram, in their system this has to be met with one of the system's rules. These axioms are divided into categories according to the impact they depict from the diagram. First are "construction rules" these are the base of their rule system and are described as the "built-in theorems that are available from the start". The construction rules include rules about points, lines, circles and intersections of the above. Then they define the "diagrammatic inference" axioms, which replace the diagram's purpose. Then are the rules about "transfer inferences" on the diagrams, their intention is to depict how the changes on a diagram, as Euclid describes them, apply on their facts. The last rules they define are about "superposition inferences", these rules are about how one diagram relates to another and the rules that apply to the former apply to the latter as well. They conclude this section by explaining the "direct consequence" notion, that they based their approach on. However, while Euclid takes some parts of the construction as granted, in their system they claim that more specificity is needed in order to make use of their rules. In one of the examples they give is with Proposition I.9., where Euclid generates a triangle dfe (as seen in Figure 2.1), they claim that the

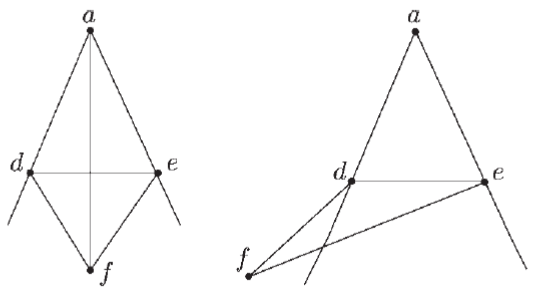


Figure 2.1: Two cases for Proposition I.9 considered by Avigad et al. [1].

f point needs more clarification for its placing in the diagram because it may fall out of the *dae* angle, which is not true. If one follows Euclid's direction for the construction the f point does cut the angle in half and it's impossible to be placed in the position they claim. Euclid's directions state that to place the f point one must construct an equilateral triangle using his proposition I.1 with *de* as a base. Creating an equilateral triangle would place point f exactly in a positions precisely cutting the *dae* angle in half when one connects f and a. Another example the use is the one of proposition I.35 (as seen in Figure 2.2), in this case there is a lot more to be discussed. While all three cases that they present as figures can be generated solely from the proposition's statement, the construction steps of the proposition eliminate the second where there is one point less. How-

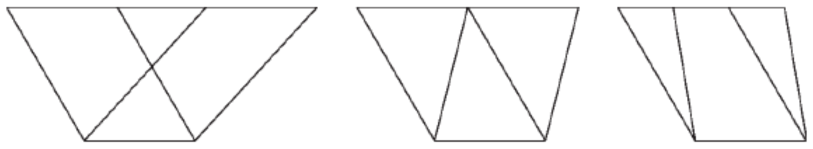
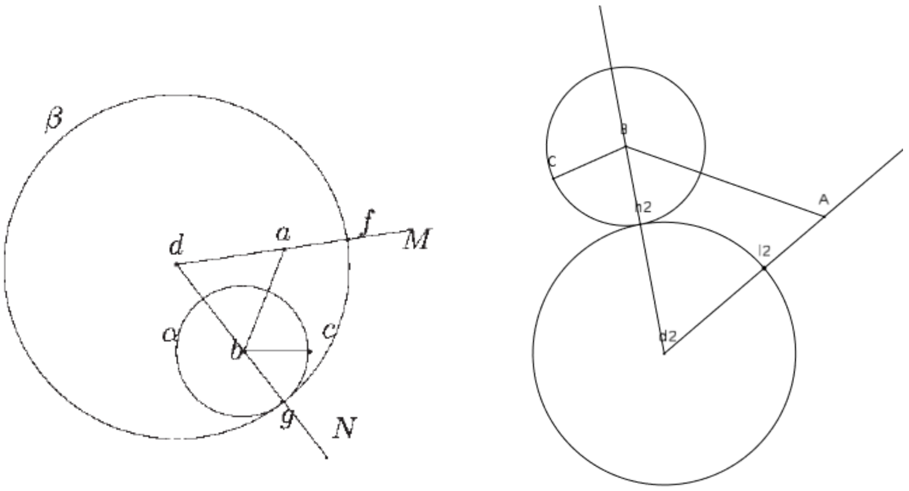


Figure 2.2: Three cases for Proposition I.35 considered by Avigad et al. [1].

ever, Euclid's aim is to prove the proposition, and while the proof might not be exactly the same for all the cases considered in Figure 2.2, the proposition is true for all which is of most importance. Euclid is vague enough where he should be, since his intend is to reuse the propositions through out his work, should he have been more precise other proofs that use this one might not be provable. It appears that Euclid is making use of the open world assumption, which makes his work all the more attractive to potential representation. Last but not least, I will refer to their statement of Euclid not stating enough information in his constructions. They take as an example proposition I.2, claiming that point a is may not be inside circle β and a clarification of $da < dg$ should be made. This is again misinterpreted since if one follows Euclid's steps to the construction, even if a fell out of circle the entire proof still stands true. To prove this, one can consider the case where $ab > bc$ for which constructed Figure 2.3b. In this case the circle with center B or circle α, would intersect with line segment $Bd2$ (or db in Figure 2.3a) in two points. When one takes the point that is between $d2$ and B to name $h2$ and forms circle with center $d2$ and radius of $d2h2$ or β. Then line segment $Ad2$ intersects with circle β at point $l2$ and Euclid's proof stands exactly the same, even in this case, since again the proposition's aim was to create a line segment from A equal to BC and that is achieved.



(A) Proposition I.2 as formed by Avigad et al. [1].

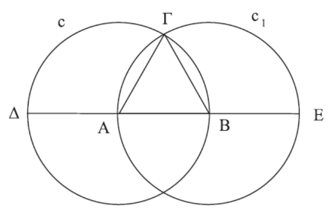
(B) Another case for Proposition I.2

FIGURE 2.3: Different representations of Proposition I.2

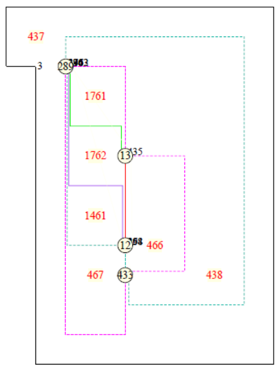
The latest approach on Euclid seems to be the on of Beeson et al. [3] where they approach the matter of "Proof-checking Euclid". In their approach they used a custom designed representation of Euclid's Elements. In this representation they merged every axiom, definition, postulate, lemma and proposition into respective letters or combination of such. The most natural thing in doing so, following Euclid, is to name variables with one character. Then, they used two-character names for the relations since they exceed 24, and used AN, OR and NO for conjuctions, disjunctions and negations respectively. In their approach they do not follow Euclid's ways of proving and also do not follow the same order as his. They seem to approach superposition as equality between figures, and as they mention in section Checking the proofs in Coq, they had to adapt to the tool and reused theorems as implicit assumptions for each lemma. They claim that their approach is more complete and adds to Euclid's work and that they are the first to do a non-paper-and-pencil formalization that proves correct and valid Book I of Euclid and their "corrected proofs of those propositions, close to Euclid's ideas". However, they too do not interpret superposition in the way Euclid does and try to find other ways around this in the construction of propositions and their proofs. For example, when they say that proposition I.9 cannot be proved using I.1 and they try to work around this.

2.4 Diagrammatic Inference

On the other hand, there are systems that act on diagrams like the one of Miller [16]. In his approach Miller creates a collection of rules that are divided into categories according to the implication of the rule on the diagram, similar to the ones of Avigad et al.. First are the construction rules, inference rules follow and then transformation rules. He treats superposition as "lemma incorporation", for which he defines and proves a theorem. Miller's Lemma Incorporation Theorem is used in his system to identify the forms that a diagram may take supposing an ancestor diagram. This becomes useful in the sense of using this lemma to get the superposition effect by isolating the diagram requiring superposition from the parent diagram, applying is to a smaller "environment" and using the result in the parent diagram. Miller concludes this section with the presentation of the system CDEG that makes use of all the above. CDEG diagrams are not as conservative and not easy to read, an example diagram is the one of Figure 2.4 which is the result of CDEG on proposition I.1 of Euclid. The main issue with this approach is that it does not fully take advantage of what Euclid essentially



(A) General representation of diagram from proposition I.1



(B) Resulting image of Proposition I.1 from Miller [16].

FIGURE 2.4: Different representations of Proposition I.1

aims at, which is also the major issue that most publications have difficulty interpreting.

Euclid does indeed have some more vague points, on the contrary to what these publications suggest, work best through his work, since the vague points are not important to the application of a proposition, rather opportunities to apply them even more times. The vague point need no more clarification, if ones does specify in more detail the context of each proposition might become more logically complete, but it will lose the point of versatility.

One approach acts directly on diagrams [16] the other solely with logic [1, 3] to tackle the problem. However, Euclid relies both on diagrams and logic in his proofs. To encode so much information may be challenging, which means that, isolating one or the other requires even more effort in designing and applying such a method.

2.5 Functional Programming

The notion of Functional Programming (FP) one would say is to transform a problem into a set of steps one should execute, with the intent to focus on the computation of the problem. Thus, instead of creating multiple paths for execution at a time, functional programming focuses on one main goal and each function is a step towards this goal.

We aim that in this process, every step is a function, which itself is a part of the solution to a greater goal. A *function* in FP takes a number of arguments, that it will use to compute the respective output.

2.5.1 Clojure

One of the first FP languages was Lisp[27], which was then used as the base of Clojure[12, 8]. Lisp is the second oldest high-level programming language [21]. That makes Clojure, even as a fairly new language, a very interesting one as well. Offering many different available approaches to a subject.

One of these ways that was also considered for this work, is metaprogramming. In metaprogramming the program is written with the intent to generate code, to create an other program. One way to take advantage of this method for this work, was to create a function that would use this automatic generation of code, is to let the program generate a figure based on custom queries for constructions and that would also be a very quick way to do so. It was one of the reasons this a Lisp family language was selected, its transform and adapt abilities. However since this approach was intuitively considered too user related and very time consuming to implement, while the aim of this project is more demonstrative and computational.

Lisp and Clojure as a consequence, have a very distinctive syntax. To begin with, everything is enclosed within parenthesis "()" and everything is function call at it core.

Chapter 3

Method Description

The method is split in three parts; the figure construction based on the description of Euclid, the extraction of facts based on the construction and the reasoning part. Each part assists the process in a different way and its equally important for the achieving the goal of the prover. In a brief description, the first part of figure construction sets the base for determining what we need, a figure; to visually understand and check the process and the crucial part of *seeing* as a human. The second part of valid fact extraction, is extracting the above knowledge to computer readable and interpretable. The last part the one of reasoning, is using all the built knowledge from the previous steps and a combination of Euclid's instructions, reason upon them and conclude to facts that can be stated in relation to the above. The facts may include, but are not limited to, the proof. In this section we describe each part and its role in the process.

3.1 Figure Construction

This part aims to create the construction according to Euclid's construction steps and generate a figure representing the figure. The first step to consider in this part, are the "Definitions" of Euclid, specifically the ones that are also used to create and describe the rest. These are points, line segments and circles.

Through out the figure construction we used the two-dimensional coordinate system to guide the computer in creating the figures. An example figure is Figure 3.1 for proposition 2 from Book I. Points are defined using x,y coordinates, and a name to later be used in the next parts (see Section 3.2). Line segments or lines [1], are defined using two points. Circles are defined using their center and the line whose size was used as radius.

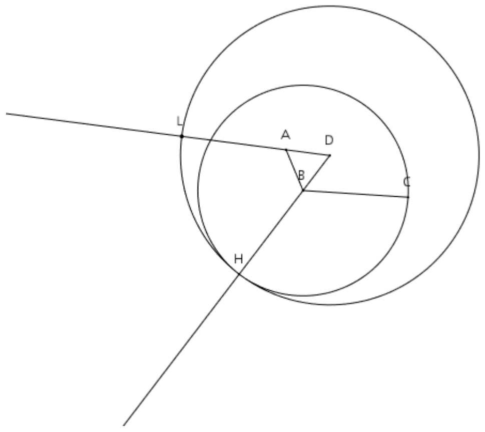


FIGURE 3.1: A proposed figure for Proposition I.2 using our approach

After defining the shapes, the next step would normally be to start implementing the Propositions, but this is not the case here. Starting off implementing propositions was attempted at this point, but, then we stumbled upon the "intersections".

Intersections between circles and lines are fundamental when it comes to implementing Euclid's construction steps. Euclid through out his work, takes intersections as given since his approach is paper-and-pencil. In our approach, this does not come as naturally, since the code needs some help *seeing* the intersections. To aid this need, we used analytical geometry *restrictively* to make the application see. By restrictively, we mean that the use of analytical geometry was not used for anything else other than make the computer see what the human eye would in a paper-and-pencil approach. This means the following methods:

- methods to seek points on shapes:
  - fetching a random point from upon a shape and
  - checking that a point is on a shape.
- methods to determine intersections between shapes:
  - line to line intersection

[1]We use only line segments through out the constructions, thus we call them lines for ease.

- line to circle intersection
- circle to circle intersection

The initial coordinate numbers are random for the points needed and said to be given from the proposition. This randomness leads to different images every time a propositions figure is constructed. An example of this are Figures 3.3 and 3.4. They are the same proposition and still generate vastly different figures in every step. After creating a construction, the next part is to extract valid facts from it.

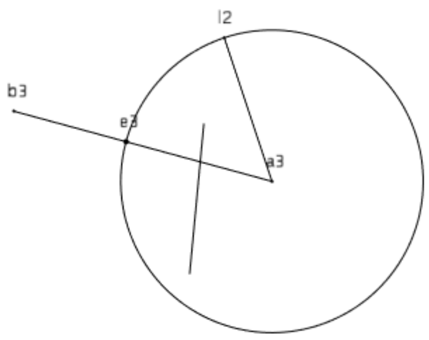


FIGURE 3.2: A generated figure for Proposition I.3 using our approach

A key part of this method is the ability to reuse any previously defined proposition inside a later one. That should be done by giving the same parameters as one would give to solely create the former proposition. This means that each proposition should be independent from its descendants, but can and should depend on its ancestors in cases of avoiding repeatability and achieving reusability. What is more, this solutions aims at ease of expandability. In this notion, each step of the construction should be clearly defined and only depend on previous steps. To keep this balance, the order in which each step is conducted is very important.

The proposed method uses a proposition-function approach. Using a function to describe each proposition, gives the advantage of reusing a proposition simply by referring to it by its name as Euclid does. To aid the ease of writing new propositions, all the functions should follow the same naming protocol to be easily discoverable, a template will be used that should only need the construction steps to be filled. The printing part should have the option of choosing what one wants to put in the generated image, while keeping it simple enough to follow.

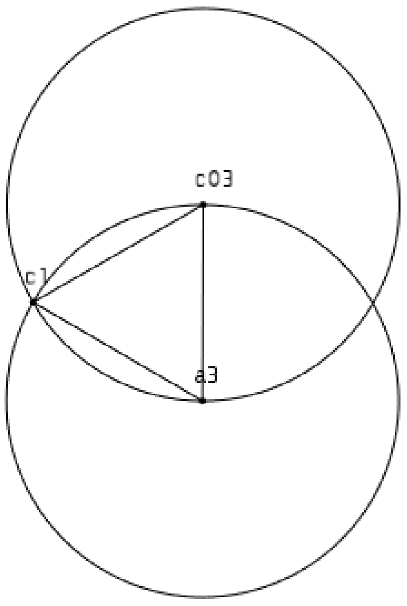
3.2 Valid Fact Extraction

The key step in the valid fact extraction is actually to extract the relations between the shapes created in the construction. For this step we reused the previous Section's proposition-function method by appending to the function template a part were the we choose what will be considered for te fact generation.

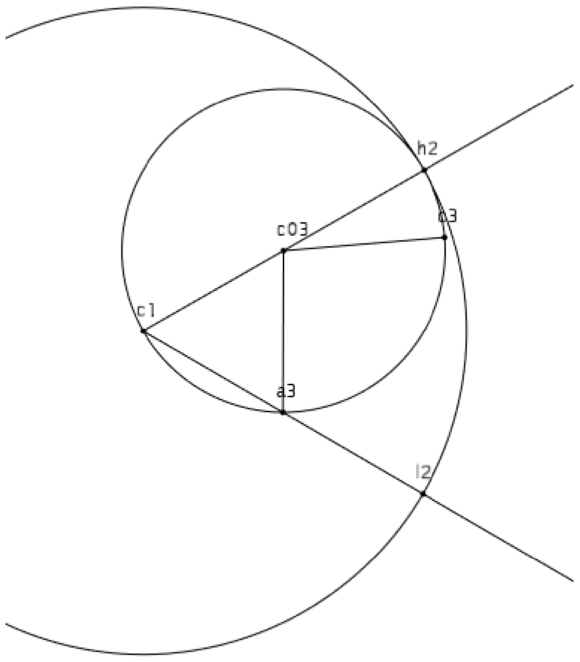
Facts are essentially properties of the construction, that are translated into Prolog predicates. To achieve this, we gathered the created points from each construction and wrote files that contained for each proposition every possible shape. By *every possible* shape, we mean that after following the steps for each construction, we end up with a collection of shapes, included in the figure of the proposition. These shapes have connecting points from the way they have been constructed, depending one to the other. For example, a point A and another point B, are used for constructing the line AB, by connecting A to B. A full example of generated facts on Proposition I.3 can be seen in Figure ?? Another example, is when a rectilinear is created using lines with points in common. Connections as such had to be translated to facts as well, since they are a key point to creating a proof.

We consider a connection as a fact, when a shape interacts with another inside the propositions construction and only them. Interactions that appear but have no defined relation to the constructions are not considered at all. For example, Figure 3.2 is a figure generated for Proposition I.3, one can see that the line c intersects with line connecting $a3$ to $e3$. It is not a wrong construction, the numbers used were random and these accidents or coincidences can happen. On the other hand, if we let these coincidences define the turn out of the proof the proof would be meaningless. Thus, there is no reference to this types of coincidences in the facts.

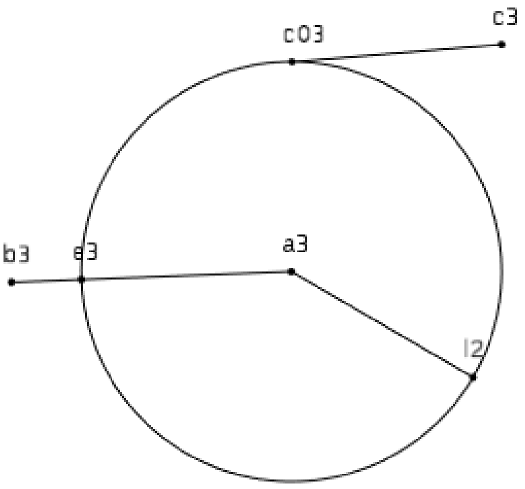
Extracting a proposition's facts is essential, but itself would not be enough. Euclid makes his propositions use one another for a reason that should not be overlooked. Skipping this connection and the property inheritance they provide, leads to making a problem harder to solve or even unsolvable.



(A) Proposition I.1 generated for assisting Proposition I.3

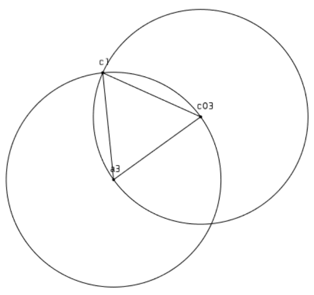


(B) Proposition I.2 generated for assisting Proposition I.3

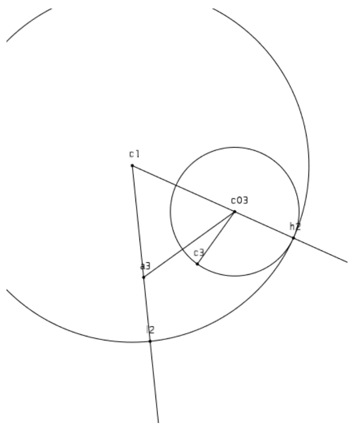


(C) Proposition I.3 with a line-circle intersection coincidence.

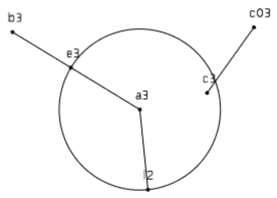
FIGURE 3.3: A full example of the steps to construct I.3



(A) Proposition I.1 generated for assisting Proposition I.3



(B) Proposition I.2 generated for assisting Proposition I.3



(C) Proposition I.3 with a line-circle intersection coincidence.

FIGURE 3.4: Another full example of the steps to construct I.3

To better understand this, consider Figure 3.4, displaying the process of constructing proposition I.3. Figure 3.4c is the final figure and outcome of I.3. The point of this proposition is to subtract from line *a3*-*b3* a line equal to line *c03*-*c3* (line *c*). To create this according to Euclid, first one needs to use point *a3*, to create an equal line to line *c*, using Proposition I.2. This caused Figure 3.3b a step before completing I.3 to be generated. However to construct I.2 Euclid makes use of I.1 (Figure 3.4a) to construct the needed equilateral triangle. Every one of those figures' constructions is translated into Prolog facts. Since they contain key information on the shapes and their connections, there is no way to get the equalities of the goal proposition without them.

3.3 Reasoning on Generated Facts

This part contains the needed logic for proving and making logic conclusions on the theorem. The prover itself is a set of clauses that include the definitions, postulates and common notions that Euclid made in his first book. The definitions, postulates and common notions are translated into definite clauses, along with the notion of subtraction of lines.

```
circle(a3, line(a3, c03)).
circle(a3, line(a3, l2)).
circle(c03, line(a3, c03)).
circle(c03, line(c03, c3)).
circle(c1, line(c1, h2)).
line(a3, b3).
line(a3, c03).
line(a3, e3).
line(a3, l2).
line(b3, e3).
line(c03, c3).
line(c1, a3).
line(c1, c03).
line(c1, h2).
line(h2, c03).
on(circle(a3, line(a3, c03)), point(c03)).
on(circle(a3, line(a3, c03)), point(c1)).
on(circle(a3, line(a3, l2)), point(e3)).
on(circle(a3, line(a3, l2)), point(l2)).
on(circle(c03, line(a3, c03)), point(a3)).
on(circle(c03, line(a3, c03)), point(c1)).
on(circle(c03, line(c03, c3)), point(c3)).
on(circle(c03, line(c03, c3)), point(h2)).
on(circle(c1, line(c1, h2)), point(h2)).
on(circle(c1, line(c1, h2)), point(l2)).
on(line(a3, b3), point(a3)).
on(line(a3, b3), point(b3)).
on(line(a3, b3), point(e3)).
on(line(a3, c03), point(a3)).
on(line(a3, c03), point(c03)).
on(line(a3, e3), point(a3)).
on(line(a3, e3), point(e3)).
on(line(a3, l2), point(a3)).
on(line(a3, l2), point(l2)).
on(line(c03, c3), point(c03)).
on(line(c03, c3), point(c3)).
on(line(c1, a3), point(a3)).
on(line(c1, a3), point(c1)).
on(line(c1, c03), point(c03)).
on(line(c1, c03), point(c1)).
on(line(c1, h2), point(c03)).
on(line(c1, h2), point(c1)).
on(line(c1, h2), point(h2)).
on(triangle(line(a3, c03), line(c1, a3), line(c1, c03)), point(a3)).
on(triangle(line(a3, c03), line(c1, a3), line(c1, c03)), point(c03)).
on(triangle(line(a3, c03), line(c1, a3), line(c1, c03)), point(c1)).
triangle(line(a3, c03), line(c1, a3), line(c1, c03)).
```

FIGURE 3.5: Generated facts for Proposition I.3

For this part to work one has to consult both the prover file that contains Euclid's statements and the file generated from the proposition described in Section 3.2 In this way, this part can assist in deriving more than just the theorem, but also other valid statements about the construction.

Consider the example of proving the equality of two lines based on them both being radii of the same circle. For the purpose of this example we will use proposition's I.3 figure (Figure 3.3c). Inside this proof one has to prove that `line(a3,e3)` and `line(a3,l2)` are equal. We will first translate Euclid's Definition 15 into a Horn Clause (Figure 3.6).

Definition 15. A circle is a plane figure contained by a single line -which is called a circumference-, (such that) all of the straight-lines radiating towards -the circumference- from one point amongst those lying inside the figure are equal to one another.

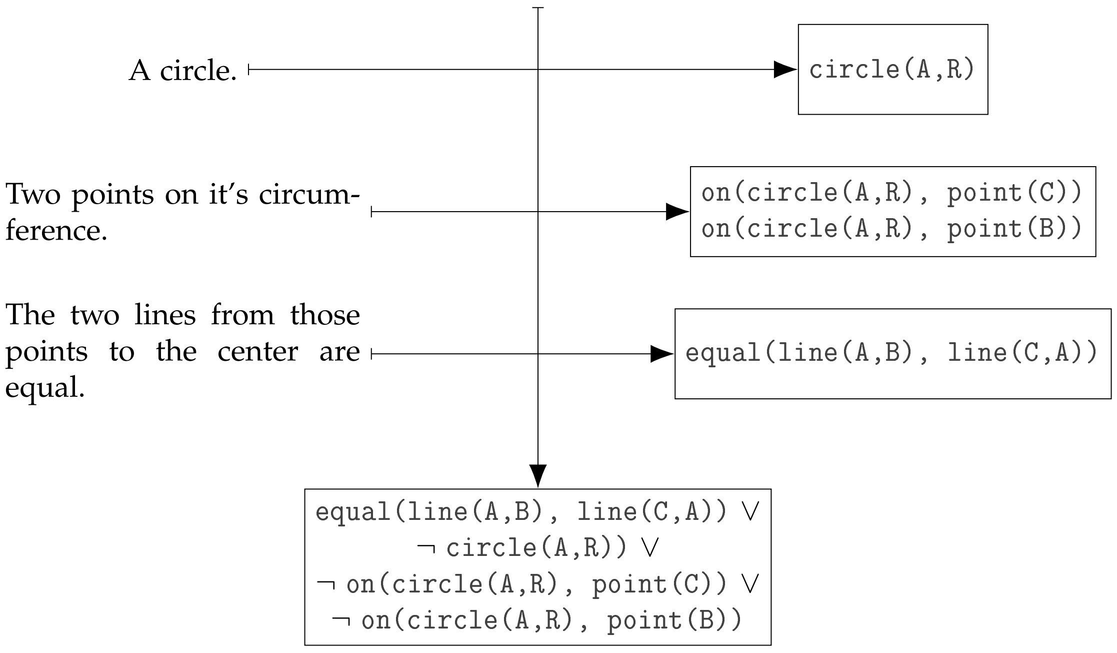

A circle.
circle(A,R)
Two points on it's circumference.
on(circle(A,R), point(C))
on(circle(A,R), point(B))
The two lines from those points to the center are equal.
equal(line(A,B), line(C,A))
equal(line(A,B), line(C,A)) ∨
¬ circle(A,R)) ∨
¬ on(circle(A,R), point(C)) ∨
¬ on(circle(A,R), point(B))

FIGURE 3.6: Translating Definition 15 to a Horn Clause

Then we will translate that `line(a3,e3)` and `line(a3,l2)` are equal (Figure 3.7) and take the needed background knowledge from the generated facts for Proposition I.3

Based on the Horn clause of Figure 3.6 the suggested way to resolve 3.7 would be as seen in Figure 3.9.

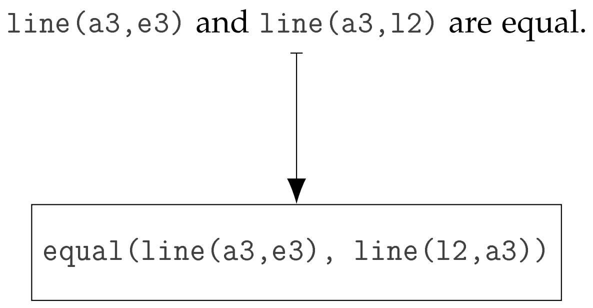

line(a3,e3) and line(a3,l2) are equal.
equal(line(a3,e3), line(l2,a3))

FIGURE 3.7: Observation to be explained

```
       circle(a3, line(a3, l2))
on(circle(a3, line(a3, l2)), point(l2))
on(circle(a3, line(a3, l2)), point(e3))
```

FIGURE 3.8: Background knowledge (taken from Proposition I.3)

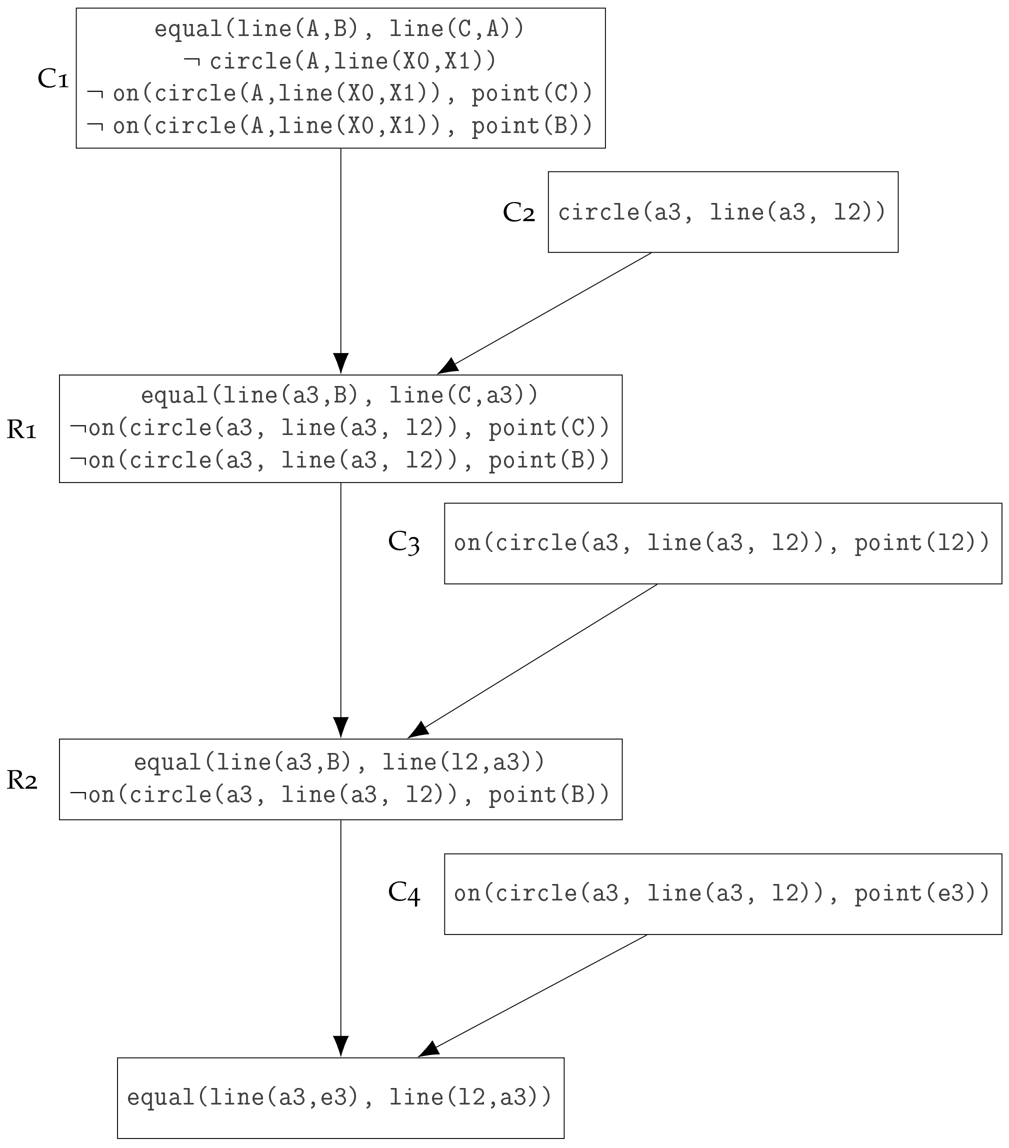

C1
equal(line(A,B), line(C,A))
¬ circle(A,line(X0,X1))
¬ on(circle(A,line(X0,X1)), point(C))
¬ on(circle(A,line(X0,X1)), point(B))
C2
circle(a3, line(a3, l2))
R1
equal(line(a3,B), line(C,a3))
¬on(circle(a3, line(a3, l2)), point(C))
¬on(circle(a3, line(a3, l2)), point(B))
C3
on(circle(a3, line(a3, l2)), point(l2))
R2
equal(line(a3,B), line(l2,a3))
¬on(circle(a3, line(a3, l2)), point(B))
C4
on(circle(a3, line(a3, l2)), point(e3))
equal(line(a3,e3), line(l2,a3))

FIGURE 3.9: Horn Resolution for line equality based on them both being radii of the same circle

Chapter 4

Implementation

This section describes the tools used for implementing the method of Chapter 3.

4.1 Implementation Part: Figure Construction

Figure Construction was implemented using Clojure [8] programming language and specifically the Quil [17] library. It produces images based on the logic described in Section 3.1.

4.1.1 Infrastructure

The decision on Clojure was made upon the facts that it had to be able to generate images, remain simple to add new propositions and combine various propositions again in a fairly easy way. An approach was also made to create the figures using Prolog, but it proved to be a complicated procedure to add new ones, even more to combine them.

4.1.2 Structure of Code

The parts to consider in the implementation of this part are the following:

1. Declaration of the basic shapes, as in Euclid's definitions. This is done in the `base.clj` file.

2. Implementation for what an eye can see, but the computer can not. That is limited to the intersections between basic shapes. To do this we used Analytical Geometry. Implemented in the `core.clj`.

Fact	Example
Point	`point(c1).`
Line	`line(c1, h2).`
Circle	`circle(c1, line(c1, h2)).`
Triangle	`triangle(line(a3, c03), line(c1, a3), line(c1, c03)).`
Attribute	`on(line(c1, h2), point(c03)).`
	`on(circle(c1, line(c1, h2)), point(l2)).`

TABLE 4.1: Examples of generated facts.

3. Propositions from Euclid's Book I. The implemented propositions are included in the `propositions.clj`.

Each proposition is a function in this system. This means that it can be used again, especially inside other propositions, only by giving the needed parameters.

4.1.3 Completeness

On the part of Figure Construction and valid fact generation six propositions where implemented. Example generated images for each are displayed in Figure 4.1.

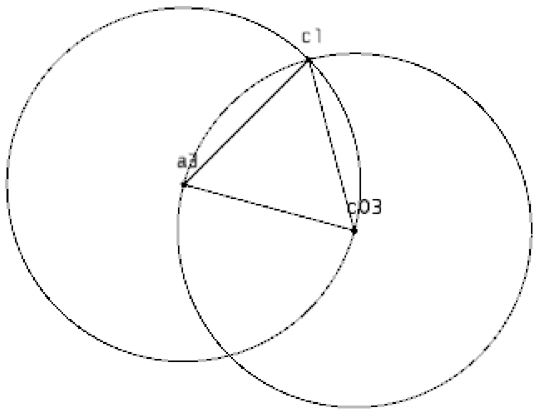
4.2 Implementation Part: Valid Fact Extraction

When each proposition is constructed, every shape created is appended as a Prolog fact to a *.pl* file [23]. The attributes currently generated are basic shapes and their relations. Basic shapes include points, lines, circles, triangles and angles, while relations are mostly of intersection type denoted as *on* attributes.

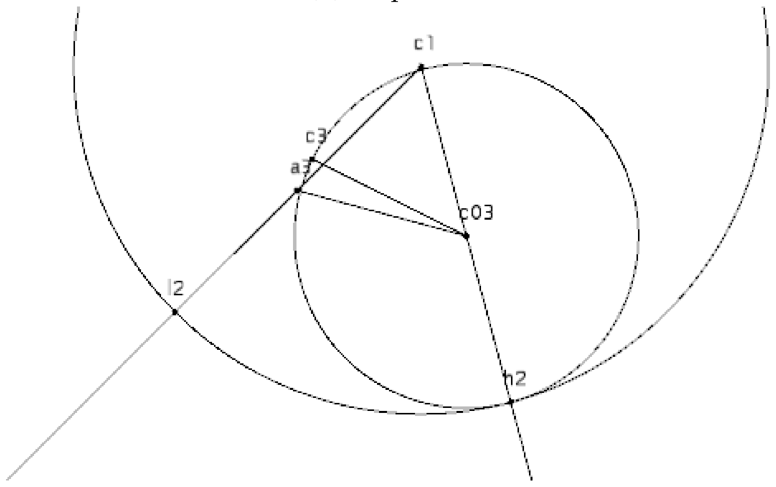
4.2.1 Structure of Code

For this part I added more functionality in the code described in Section 4.1.2. That is the combinations part as well as the printing of the facts to a `.pl` file [1]. Examples of the facts can be seen in Table 4.1.

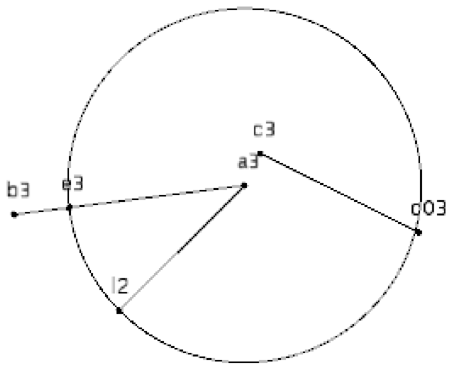
[1]https://www.swi-prolog.org/pldoc/man?section=projectfiles



(A) Proposition I.1

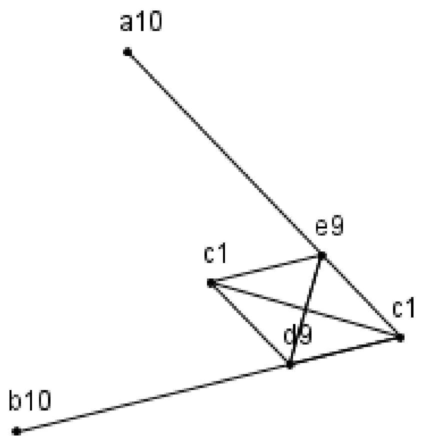


(B) Proposition I.2

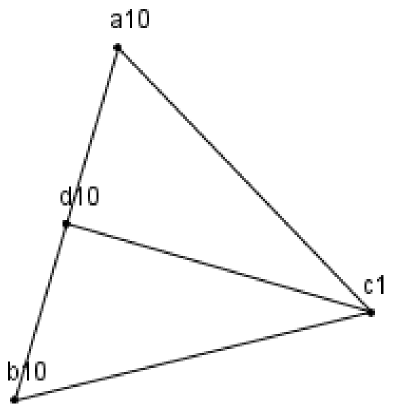


(C) Proposition I.3

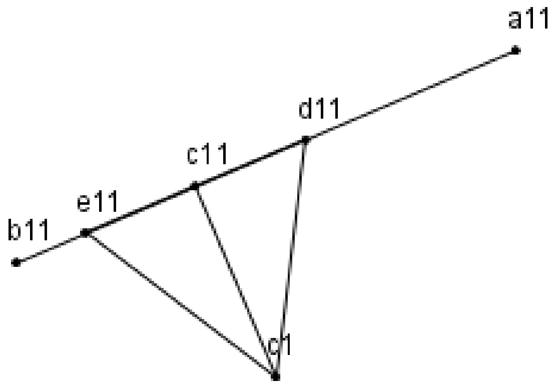
FIGURE 4.1: Figures of propositions as generated from the program



(D) Proposition I.9



(E) Proposition I.10



(F) Proposition I.11

FIGURE 4.1: Figures of propositions as generated from the program

4.3 Implementation Part: Reasoning

For the reasoning part I used Prolog and specifically SWI-Prolog [25]. In this part, I wrote a set of facts as Euclid describes them to create the solving environment. An example is given in Code 4.3.

Listing 4.1: Euclid's Definition 15.

```
equal(line(A,B), line(C,B)):- circle(B,X),
on(circle(B,X), point(C)),
on(circle(B,X), point(A)).
```

This is a translation that all lines from the center of a circle to points of the circumference are equal, which corresponds to Euclid's Definition 15 (as seen in [9]):

> 15. A circle is a plane figure contained by a single line [which is called a circumference], (such that) all of the straight-lines radiating towards [the circumference] from one point amongst those lying inside the figure are equal to one another

4.3.1 Completeness

For the Reasoning part the basis was set for most of these and more. At this state of the implementation, valid statements may be equality between the lines and type of a triangle. Though these may seem limited functionalities, they are key to solving most theorems. It requires more work to be fully complete for all propositions though. The example described in Section 3.3, this time using Prolog.

Listing 4.2: Example described in Section 3.3 using Prolog

```
?- equal(line(a3, e3),L).
L = line(a3, b3)-line(b3, e3);
L = line(l2, a3);
```

4.4 Adding more propositions

A key characteristic of this structure is the ability to add more propositions with ease, even completely custom ones. This requires appending a function inside the `propositions.clj` file, as explained in the following paragraphs.

To create a specific proposition one has to know the steps of its construction. These steps can then be translated to code, using a combination of predefined operations and shapes.

4.4.1 Predefined Shapes and Operations

By predefined we mean that are already implemented and available to use in the way described for each one below.

Predefined Shapes Each shape has a set of attributes that are needed for its creation.

Point. Every point has the following attributes, that should be passed in the order indicated:

1. *name* - the name of a point should be begin with a lower case character. It will be used both for the generated images and the fact extraction.
2. *x* - the coordinate x of this point, though we used a coordinate system for the design of the generated images, the coordinates cannot and are not used anywhere else.
3. *y* - the coordinate y of this point, though we used a coordinate system for the design of the generated images, the coordinates cannot and are not used anywhere else.

Line. For the purpose of this approach lines are in their core line segments, thus, they consists of two points. It does't matter which is in the left or right point of the line.

1. *p1* - the first point of this line. Should be of type `Point`..
2. *p2* - the second point of this line. Should be of type `Point`..

Circle.

1. *center* - the center of the circle. Should be of type `Point`..
2. *radius* - the radius in this approach is defined with the line that was used as a guideline for the circle. This is essentially a line connecting the center and a point in the perimeter of the circle. Should be of type `Line`.

Triangle. Three lines connecting three points. In this case one should be aware of the linking between the line points, since it very likely that they will be used in later stage.

1. *l1* - Should be of type `Line`..
2. *l2* - Should be of type `Line`..
3. *l3* - Should be of type `Line`..

Additional shapes where defined in the code that were not used for the implemented propositions and for this reason they are not described.

Predefined Operations

- `point-distance` - returns the distance between two points.
- `point-on` - can either take 1 or two parameters. When given one parameter it returns a point on this shape. When used with two parameters it the first can again be any shape, but the second parameter should be a point and this returns whether the point is on the former shape.
- `intersection-points` - takes two parameters that can be of type circle or line and returns any intersecting points.
- `extend-line-segment-to` - taks two parameters, the line to extend and the point of this line towards which we want to extend.

4.4.2 Breaking Down a Proposition-Function

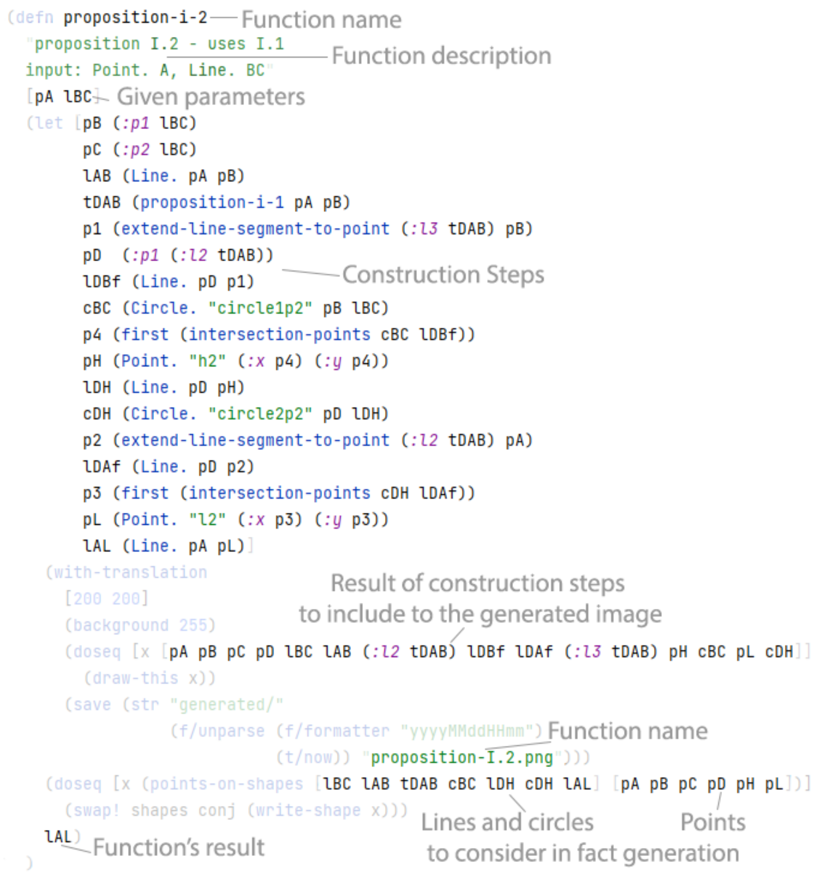
Before we are able to add a proposition it is important to undesrtand the structure of a propositions function. Such an example is the following. Each construction step is formed by two parts. The left part is a reference name of the result of the right part. The right part is a combination of predefined operations and shapes defined earlier.

For example, in Figure 4.2 we see the code of Euclid's Proposition I.2. The faded parts are Clojure code that is the same in every proposition-function and for this reason we will not explain them. The first thing of interest in Figure 4.2 is the *Function name*, that is the name we wish to give to this function and it should indicate the proposition it is for, since this name is the way we can call this proposition for late usage. Then can write a few words about this proposition in the *Function description*, this is optional, but helpful for future users. In square brackets we give the *Given parameters* for this proposition, in proposition I.2 those are point A (pA) and a line BC (lBC). These are the base of the proposition, upon these we will build the rest of the proposition.

The main part of the proposition-function is the part where we construct the figure and that is done through the *Construction Steps*. We will go through some steps of Figure 4.2 to better explain the usage of Section 4.4.1. The first two lines of the construction steps are `pB (:p1 lBC)` and `pC (:p2 lBC)`, this code says that from line BC I will take `p2` and assign it to point C (`pC`). These are done to define more clearly all the points we will use, it is optional for ease. Then we define a new line AB, for this we define a new `Line.` with the parameters as explained in Section 4.4.1, which we write as `lAB (Line. pA pB)`. This defines line AB (`lAB`), a line from point A to point B. After that, we use proposition I.1 to create an equilateral triangle as required for this construction. Proposition I.1 takes as given parameters two points. After the definition of the first circle, whose center is point B (`pB`), and has a radius equal to the size of line BC (`lBC`), there is a more peculiar point definition `p4 (first (intersection-points cBC lDBf))`. This makes use of the predefined operation `intersection-points`. However, since the intersection of a line and a circle may return up to two points, we give the keyword `first` to get the first and in some cases only point of intersection.

After the construction steps, Figure 4.2 shows a *Result of construction steps to include to the generated image*. This part we simply need to write all the reference names of the shapes we want to include in the generated image. Nothing is mandatory, however the outcome might seem odd if key parts are missing.

Another important part of Figure 4.2, is the last indicated part namely *Function's result*. This is the reference name of the shape we wanted to take from this proposition, usually the last shape created in the construction steps. As we explained earlier in this section each proposition can be reused inside other propositions, that is done so that when we need to construct something that can be constructed via another proposition we do not need to re-write all the lines. This means though, that we want to take the construction result, that is why we use the *Function's result* part.

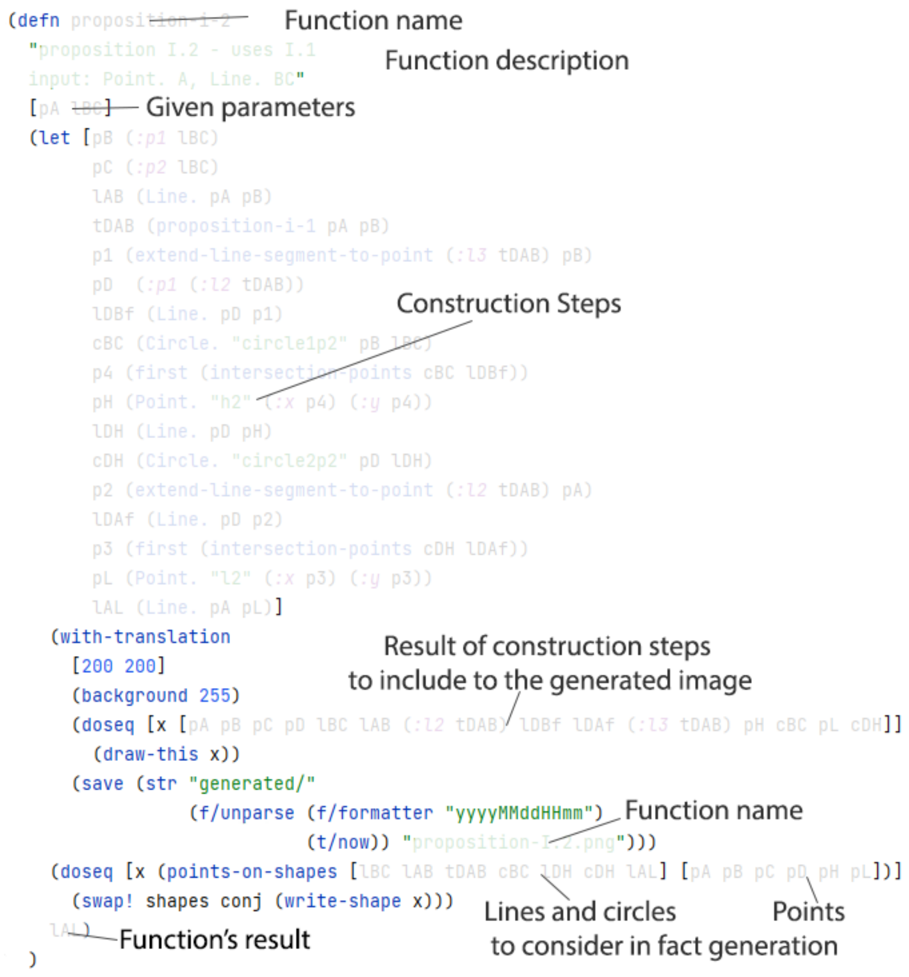


```
(defn proposition-i-2
  "proposition I.2 - uses I.1
  input: Point. A, Line. BC"
  [pA lBC]
  (let [pB (:p1 lBC)
        pC (:p2 lBC)
        lAB (Line. pA pB)
        tDAB (proposition-i-1 pA pB)
        p1 (extend-line-segment-to-point (:l3 tDAB) pB)
        pD  (:p1 (:l2 tDAB))
        lDBf (Line. pD p1)
        cBC (Circle. "circle1p2" pB lBC)
        p4 (first (intersection-points cBC lDBf))
        pH (Point. "h2" (:x p4) (:y p4))
        lDH (Line. pD pH)
        cDH (Circle. "circle2p2" pD lDH)
        p2 (extend-line-segment-to-point (:l2 tDAB) pA)
        lDAf (Line. pD p2)
        p3 (first (intersection-points cDH lDAf))
        pL (Point. "l2" (:x p3) (:y p3))
        lAL (Line. pA pL)]
    (with-translation
      [200 200]
      (background 255)
      (doseq [x [pA pB pC pD lBC lAB (:l2 tDAB) lDBf lDAf (:l3 tDAB) pH cBC pL cDH]]
        (draw-this x))
      (save (str "generated/"
                 (f/unparse (f/formatter "yyyyMMddHHmm")
                            (t/now)) "proposition-I.2.png")))
    (doseq [x (points-on-shapes [lBC lAB tDAB cBC lDH cDH lAL] [pA pB pC pD pH pL])]
      (swap! shapes conj (write-shape x)))
    lAL)
  )
```

Figure 4.2: The code for Proposition I.2. The faded parts of the figure are generic and the same in all functions/propositions.

4.4.3 Adding a New Proposition

To add a new proposition we need to replace all the parts analyzed in the previous Section, these are pointed out in Figure 4.3. The entire proposition's function is then appended inside the `propositions.clj` file.



```
(defn proposition-i-2
  "proposition I.2 - uses I.1
  input: Point. A, Line. BC"
  [pA lBC]
  (let [pB (:p1 lBC)
        pC (:p2 lBC)
        lAB (Line. pA pB)
        tDAB (proposition-i-1 pA pB)
        p1 (extend-line-segment-to-point (:l3 tDAB) pB)
        pD  (:p1 (:l2 tDAB))
        lDBf (Line. pD p1)
        cBC (Circle. "circle1p2" pB lBC)
        p4 (first (intersection-points cBC lDBf))
        pH (Point. "h2" (:x p4) (:y p4))
        lDH (Line. pD pH)
        cDH (Circle. "circle2p2" pD lDH)
        p2 (extend-line-segment-to-point (:l2 tDAB) pA)
        lDAf (Line. pD p2)
        p3 (first (intersection-points cDH lDAf))
        pL (Point. "l2" (:x p3) (:y p3))
        lAL (Line. pA pL)]
   (with-translation
     [200 200]
     (background 255)
     (doseq [x [pA pB pC pD lBC lAB (:l2 tDAB) lDBf lDAf (:l3 tDAB) pH cBC pL cDH]]
       (draw-this x))
     (save (str "generated/"
                (f/unparse (f/formatter "yyyyMMddHHmm")
                           (t/now)) "proposition-I.2.png")))
   (doseq [x (points-on-shapes [lBC lAB tDAB cBC lDH cDH lAL] [pA pB pC pD pH pL])]
     (swap! shapes conj (write-shape x)))
   lAL)
  )
```

Function name
Function description
Given parameters
Construction Steps
Result of construction steps to include to the generated image
Function name
Lines and circles to consider in fact generation
Points
Function's result

FIGURE 4.3: The code for Proposition I.2. The faded parts of the figure indicate that they need to be replaced to generate a new proposition.

Chapter 5

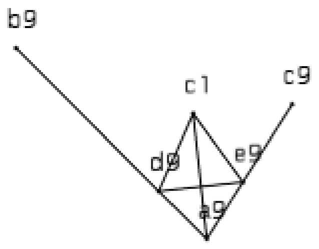
Conclusions

This thesis focused on the automated deduction problem over the field of theoretical geometry. Using Euclid's Elements as the base for our data, we tried to technically approach the problem from a different perspective than previous attempts. As discussed in Sections 2.3 and 2.4, these attempts' methodologies vary from using entirely mathematical logic to using only diagrammatic inference. However, they were all based on Euclid's work and evolved around making it logically complete.

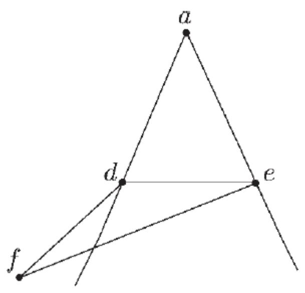
5.1 Discussion

Our method and implementation aims to create a solution that does not follow this pattern. The first step in creating this solutions was to follow the exact steps of Euclid. Many of the approaches discussed in Section 2 claimed that to implement Euclid's Elements one had to give a considerable amount of effort into overcoming logic obstacles that Euclid did not consider. The most critical of which, straying away from Euclid's logic, not reusing previously defined propositions for construction of later ones.

In our implementation we did not step into those obstacles, because of the solution's design of proposition-function (Section 3.1). The way this works is that each proposition is defined as a function its implementation depends only on previously defined propositions, as Euclid did throughout his work. In many other attempts to implement Euclid's work, re-defining and inserting more propositions and definitions was declared a necessity. Our approach is more of a digital alternative of Euclid, we do not claim to have fixed anything from his work. Instead of re-writing his method, we used the method he suggested. That



(A) Generated figure for proposition I.9 from our approach



(B) Possible error displayed in Proposition I.9 displayed by Avigad et al.

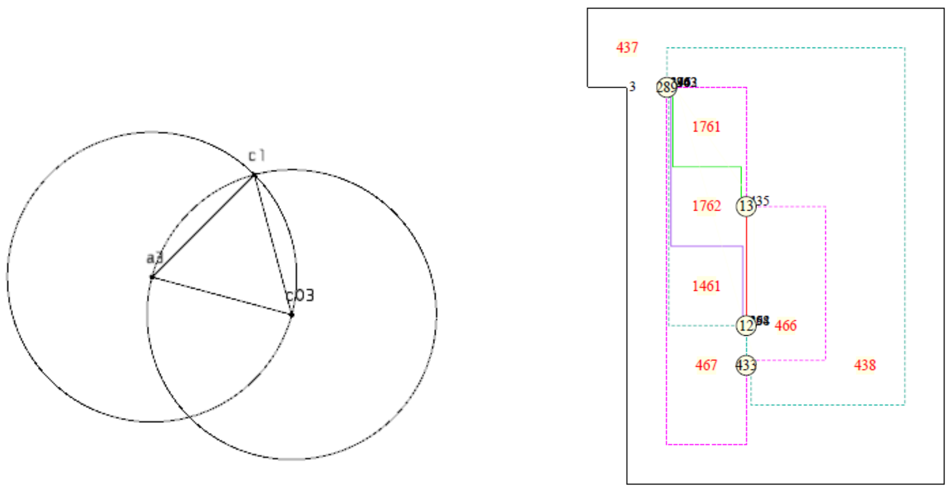
FIGURE 5.1: Comparison between our approach on Proposition I.9 and the possible error displayed from Avigad et al.[1]

being said, we did create workarounds, not for the methodology of Euclid, but for the gap between a human eye and a computer "vision", as pointed out in Section 3.1.

One example where the above design helps is during the construction of Proposition I.9.Avigad et al. [1] suggested that a figure like Figure 5.1b could be conducted by following Euclid's directions. Using our construction that does not stand true, because the *de* line, according to Euclids directions, is used as the base of an equilateral triangle (Proposition I.1). This means that point f will naturally fall between points *d* and *e*. As seen in the construction from our implementation (Figure 5.1a), where *f* is seen as *c1*, *d* is *d9*, *e* is *e9* and *a* is *a9*. While Avigad et al. did not fully go according to Euclid, they did achieve the formalization of his work. The most core difference from this approach is the simplicity in ours. We tried to keep the thought process strictly in the modeling of the solution and not in the knowledge transfer from Euclid to the implementation.

What is more, the images generated are actually human readable and complete representation of the construction. Fully designed by the same function that extracted the generated facts for reasoning and yet easily understandable by a human in contrast to Miller [16], as demonstrated in Figure 5.2. Which also makes the diagrams a helpful tool to validate the outcome of reasoning.

Another key part of this approach, is the ability to add more propositions with ease as explained in Section 4.4. This includes but is not limited to Euclid's propositions of Book I. Since the design of the code is such that allows the creation of custom propositions as well.



(A) Generated figure for proposition I.1 from our approach

(B) Resulting image of Proposition I.1 from Miller.

Figure 5.2: Comparison between our approach on Proposition I.1 and one from Miller [16]

Last but not least, an important note is a comparison between our approach and systems like Coq[2] or Theorema[26]. The latter are based on, proof checking and automated theorem proving respectively. As seen in a boarder comparison of theorem provers done by Wiedijk[24], Coq is mostly a proof checking tool to which we have a part that is related, since we need the construction of the figure to reason upon. However, we then leave the prover to make the conclusions, which is more relatable to Theorema - being an automated theorem prover. As with Theorema, our approach uses a mixture of input and predefined logic to be able to derive conclusions, in contrast with Coq that uses user input solely. At a higher level of abstraction, our approach aims to create a system that follows the concept of a human while solving a problem, meaning we take an input describing the data available, and derives conclusions based on these. The process of creating a complete input is not a concern of the user though, since the user's job is to provide his construction and it the system's job to generate all the facts that can be derived from this construction. This apart from saving time, is also a much needed logic since there is no room for *missing* trivial or not facts.

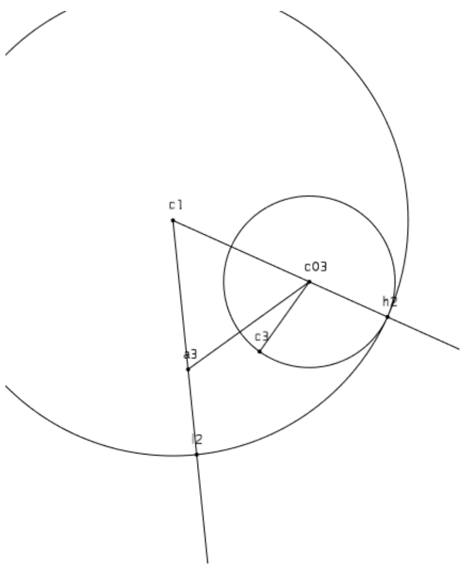
5.2 Future Work

In the scope of this work, we have developed limited operations like line equality and determining the type of a triangle. There is more content already included but is not yet in use and has not been tested. This includes definition and type determination of angles, surfaces, quadrilaterals and polygons. Undoubtedly we would like to add more propositions at least from the first books of Euclid's Elements and even test or prove custom theorems.

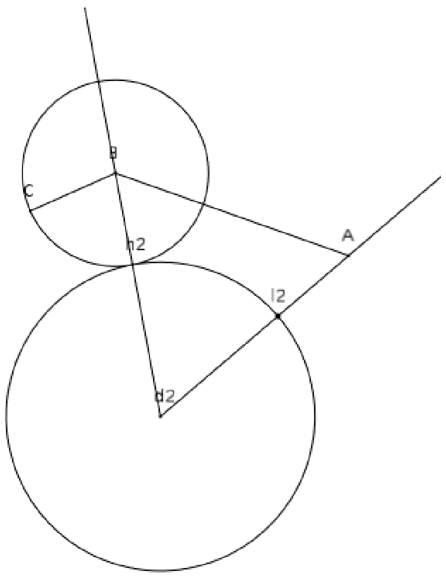
One idea of expansion is to not have to give the goal statement to the prover instead we would only need to provide the construction. Then, the prover would automatically generate all the possible facts that are considered valid for this construction. While this would ease the process of the proof, it would also be a good way to see any other possible fact trivial or not to the proof. Which would also give important information about the entire theorem.

Another interesting angle to consider would be the generation of coincidental facts in separate file. As explained in Section 3.2, since the figures are created using as initial points random coordinates, there might occur several different shapes and figures, all describing the same proposition (Figures 5.3). Extracting facts that would not be included by default since they are not clearly made by the steps of the construction, that are simply a coincidence, are the ones here called coincidental. Considering the coincidental facts could lead the proof in different paths, for example to a completely new theorem. The main idea behind this is to create a separate file that would contain all the coincidental facts. We would use a separate file because we want to clearly state the difference from the original generated facts and still keep the consistency and structure of the facts untouched. Since we are using a separate file we would have to also insert and load this one also to the reasoning part. Making use of the previous idea as well, generating facts without the need to explicitly state the goal, could lead to the generation of an entirely new theorem.

What is more, an interesting application would be in Computer-Aided Design (CAD). As explained by Martin [15], in CAD it is of interest to be able to observe geometric properties that are implications of the geometric constraints used on a design.



(A) First case Proposition I.2



(B) Second case Proposition I.2

FIGURE 5.3: Examples of possible cases for Proposition I.2

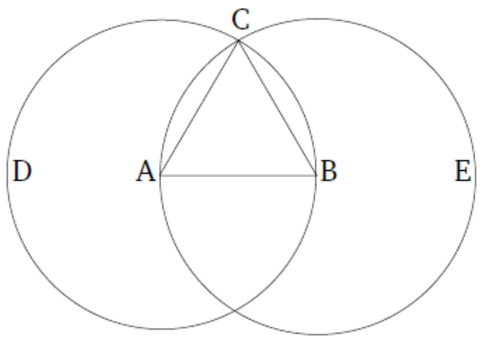
Appendix A

Referenced Euclid Propositions

In this Appendix we display the propositions implemented as seen in [9].

Proposition 1

To construct an equilateral triangle on a given finite straight-line.



Let AB be the given finite straight-line.

So it is required to construct an equilateral triangle on the straight-line AB.

Let the circle BCD with center A and radius AB have been drawn [Post. 3], and again let the circle ACE with center B and radius BA have been drawn [Post. 3]. And let the straight-lines CA and CB have been joined from the point C, where the circles cut one another,† to the points A and B (respectively) [Post. 1].

And since the point A is the center of the circle CDB, AC is equal to AB [Def. 1.15]. Again, since the point B is the center of the circle CAE, BC is equal to BA [Def. 1.15]. But CA was also shown (to be) equal to AB. Thus, CA and CB are each equal to AB. But things equal to the same thing are also equal to one another [C.N. 1]. Thus, CA is also equal to CB. Thus, the three (straight-lines) CA, AB, and BC are equal to one another.

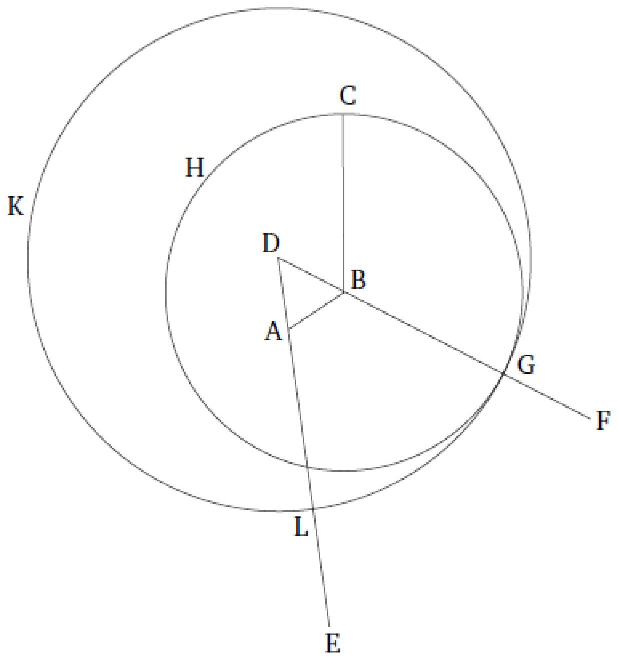
Thus, the triangle ABC is equilateral, and has been constructed on the given finite straight-line AB. (Which is) the very thing it was required to do.

Proposition 2

To place a straight-line equal to a given straight-line at a given point (as an extremity).

Let A be the given point, and BC the given straight-line. So it is required to place a straight-line at point A equal to the given straight-line BC.

For let the straight-line AB have been joined from point A to point B [Post. 1], and let the equilateral triangle DAB have been been constructed upon it [Prop. 1.1]. And let the straight-lines AE and BF have been produced in a straight-line with DA and DB (respectively) [Post. 2]. And let the circle CGH with center B and radius BC have been drawn [Post. 3], and again let the circle GKL with center D and radius DG have been drawn [Post. 3].



Therefore, since the point B is the center of (the circle) CGH, BC is equal to BG [Def. 1.15]. Again, since the point D is the center of the circle GKL, DL is equal to DG [Def. 1.15]. And within these, DA is equal to DB. Thus, the remainder AL is equal to the remainder BG [C.N. 3]. But BC was also shown (to be) equal to BG. Thus, AL and BC are each equal to BG. But things equal to the same thing are also equal to one another [C.N. 1]. Thus, AL is also equal to BC.

Thus, the straight-line AL, equal to the given straight-line BC, has been placed at the given point A. (Which is) the very thing it was required to do.

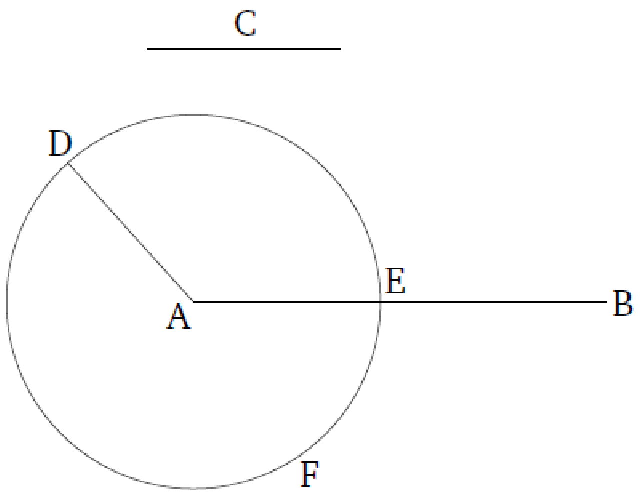
Proposition 3

For two given unequal straight-lines, to cut off from the greater a straight-line equal to the lesser.

Let AB and C be the two given unequal straight-lines, of which let the greater be AB. So it is required to cut off a straight-line equal to the lesser C from the greater AB.

Let the line AD, equal to the straight-line C, have been placed at point A [Prop. 1.2]. And let the circle DEF have been drawn with center A and radius AD [Post. 3].

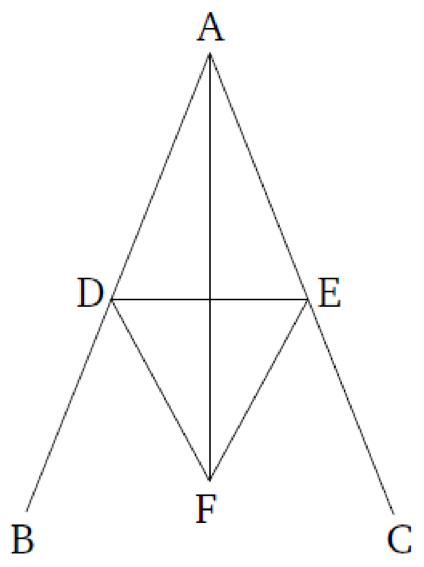
And since point A is the center of circle DEF, AE is equal to AD [Def. 1.15]. But, C is also equal to AD. Thus, AE and C are each equal to AD. So AE is also equal to C [C.N. 1].



Thus, for two given unequal straight-lines, AB and C, the (straight-line) AE, equal to the lesser C, has been cut off from the greater AB. (Which is) the very thing it was required to do.

Proposition 9

To cut a given rectilinear angle in half.



Let BAC be the given rectilinear angle. So it is required to cut it in half.

Let the point D have been taken at random on AB, and let AE, equal to AD, have been cut off from AC [Prop. 1.3], and let DE have been joined. And let the equilateral triangle DEF have been constructed upon DE [Prop. 1.1], and let AF have been joined. I say that the angle BAC has been cut in half by the straight-line AF.

For since AD is equal to AE, and AF is common, the two (straight-lines) DA, AF are equal to the two (straight-lines) EA, AF, respectively. And the base DF is equal to the base EF. Thus, angle DAF is equal to angle EAF [Prop. 1.8].

Thus, the given rectilinear angle BAC has been cut in half by the straight-line AF. (Which is) the very thing it was required to do.

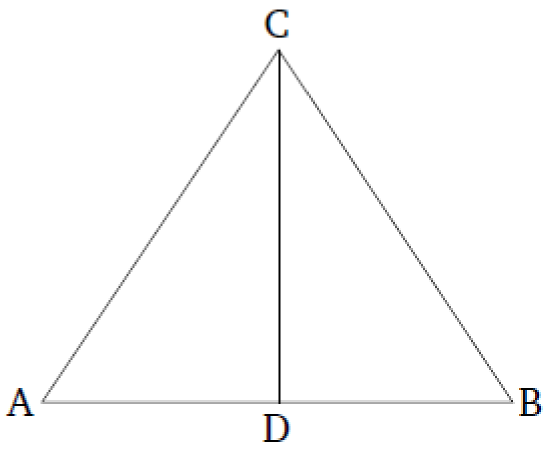
Proposition 10

To cut a given finite straight-line in half.

Let AB be the given finite straight-line. So it is required to cut the finite straight-line AB in half.

Let the equilateral triangle ABC have been constructed upon (AB) [Prop. 1.1], and let the angle ACB have been cut in half by the straight-line CD [Prop. 1.9]. I say that the straight-line AB has been cut in half at point D.

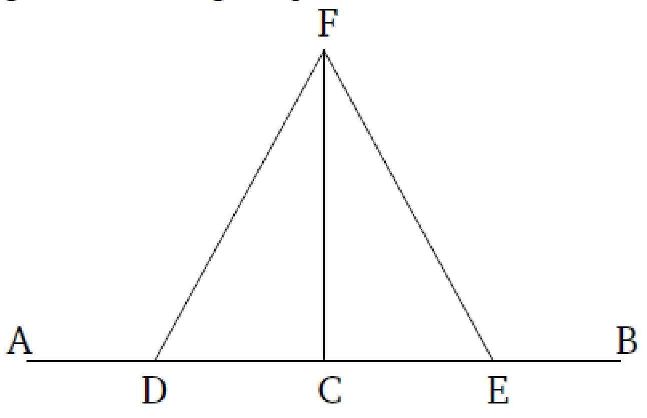
For since AC is equal to CB, and CD (is) common, the two (straight-lines) AC, CD are equal to the two (straight-lines) BC, CD, respectively. And the angle ACD is equal to the angle BCD. Thus, the base AD is equal to the base BD [Prop. 1.4].



Thus, the given finite straight-line AB has been cut in half at (point) D. (Which is) the very thing it was required to do.

Proposition 11

To draw a straight-line at right-angles to a given straight-line from a given point on it.



Let AB be the given straight-line, and C the given point on it. So it is required to draw a straight-line from the point C at right-angles to the straight-line AB.

Let the point D be have been taken at random on AC, and let CE be made equal to CD [Prop. 1.3], and let the equilateral triangle FDE have been constructed on DE [Prop. 1.1], and let FC have been joined. I say that the straight-line FC has been drawn at right-angles to the given straight-line AB from the given point C on it.

For since DC is equal to CE, and CF is common, the two (straight-lines) DC, CF are equal to the two (straight-lines), EC, CF, respectively. And the base DF is equal to the base FE. Thus, the angle DCF is equal to the angle ECF [Prop. 1.8], and they are adjacent. But when a straight-line stood on a(nother) straight-line makes the adjacent angles equal to one another, each of the equal angles is a right-angle [Def. 1.10]. Thus, each of the (angles) DCF and FCE is a right-angle.

Thus, the straight-line CF has been drawn at right-angles to the given straight-line AB from the given point C on it. (Which is) the very thing it was required to do.

Bibliography

[1] Jeremy Avigad, Edward Dean, and John Mumma. "A Formal System for Euclid'S Elements". In: *The Review of Symbolic Logic* 2.4 (2009), 700–768. ISSN: 1755-0211. DOI: 10.1017/s1755020309990098. URL: http://dx.doi.org/10.1017/S1755020309990098.

[2] Bruno Barras et al. "The Coq proof assistant reference manual: Version 6.1". PhD thesis. Inria, 1997.

[3] Michael Beeson, Julien Narboux, and Freek Wiedijk. "Proof-checking Euclid". In: *Annals of Mathematics and Artificial Intelligence* 85.2 (2019), pp. 213–257.

[4] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012.

[5] W. Bibel and P.H. Schmitt. *Automated Deduction - A Basis for Applications Volume I Foundations - Calculi and Methods Volume II Systems and Implementation Techniques Volume III Applications*. Applied Logic Series. Springer Netherlands, 2013. ISBN: 9789401704373.

[6] George D Birkhoff. "A set of postulates for plane geometry, based on scale and protractor". In: *Annals of Mathematics* (1932), pp. 329–345.

[7] Max Bramer. *Logic Programming with Prolog*. Springer Science & Business Media, 2013.

[8] *Clojure Programming Language*. Accessed February 2021. Clojure.org. URL: https://clojure.org/.

[9] Richard Fitzpatrick. *Euclid's elements of geometry*. Euclidis Elementa, 2007.

[10] John Harrison. "Handbook of practical logic and automated reasoning". In: Cambridge University Press, 2009. Chap. 1.1, 2.

[11] T.L. Heath. *The Thirteen Books of Euclid's Elements*. Dover classics of science and mathematics. Dover Publications, 1956. ISBN: 9780486600888.

[12] Rich Hickey. “The Clojure programming language”. In: *Proceedings of the 2008 symposium on Dynamic languages*. 2008, pp. 1–1.

[13] Dr David Hilbert. *Grundlagen der Geometrie*. 1899.

[14] *Logic Programming*. 2006. URL: http://www.doc.ic.ac.uk/~cclw05/topics1/index.html.

[15] RR Martin. “Geometric reasoning for computer-aided design”. In: *Artificial Intelligence in Design*. Springer. 1991, pp. 47–60.

[16] N. G. Miller. “A Diagrammatic Formal System for Euclidean Geometry”. In: 2001.

[17] Quil (mix Clojure Processing). *Quil*. http://quil.info/. [Online; accessed 7-October-2020].

[18] J Michael Spivey. *An introduction to logic programming through Prolog*.

[19] G. Sutcliffe. “The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0”. In: *Journal of Automated Reasoning* 59.4 (2017), pp. 483–502.

[20] C. Suttner. “SPS-Parallelism + SETHEO = SPTHEO”. In: *Journal of Automated Reasoning* 22 (2004), pp. 397–431.

[21] Michael Swaine. *Functional Programming: A PragPub Anthology: Exploring Clojure, Elixir, Haskell, Scala, and Swift*. Pragmatic Bookshelf, 2017.

[22] Alfred Tarski. “What is elementary geometry?” In: *Studies in Logic and the Foundations of Mathematics*. Vol. 27. Elsevier, 1959, pp. 16–29.

[23] *The project source files*. Accessed February 2021. URL: https://www.swi-prolog.org/pldoc/man?section=projectfiles.

[24] Freek Wiedijk. “Comparing mathematical provers”. In: *International Conference on Mathematical Knowledge Management*. Springer. 2003, pp. 188–202.

[25] Jan Wielemaker et al. “SWI-Prolog”. In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.

[26] Wolfgang Windsteiger, Bruno Buchberger, and Markus Rozenkranz. “Theorema”. In: *The seventeen provers of the world*. Springer, 2006, pp. 96–107.

[27] Patrick Henry Winston and Berthold K Horn. “Lisp”. In: (1986).