Automated Diagrammatic Reasoning

A proof checker for the language of *E*

Benjamin J. Northrop

Masters Thesis

Department of Philosophy, Carnegie Mellon University

Acknowledgements

I'd like to thank: John Mumma for connecting me with the *Elements* and with *E*; Jeremy Avigad for never giving up on me during this long process, his patience and encouragement were boundless; Wilfried Sieg for his thoughtful, helpful feedback; my wife for making this thesis a priority even during busy, trying times; and my parents for their love and support.

Introduction

For over two thousand years, Euclid's *Elements* stood not just as the canonical reference on geometry, but as the paragon of mathematical rigor. Through meticulously argued proofs, carefully grounded in a set of definitions, postulates, and common notions, the *Elements* delivered new mathematical theorems in thirteen different books, covering both geometry and basic mathematics. These results were studied for centuries, and at large, not found wanting. Despite its stability, however, in the 19th century a new school of mathematicians, led by David Hilbert, began to question exactly that for which the *Elements* represented: its rigor. The bar for mathematical certainty had been raised; rigor demanded the complete axiomatization of all underpinnings (abstractions, modes of inference, etc.) and although Euclid's *Elements* came close to this ideal, there was a significant gap: the use of diagrams in proofs.

Euclidean proofs all maintained a similar structure: specific geometric configurations (e.g. points, lines, circles, etc.) were constructed on a diagram, and then inferences were made based on this configuration, to the eventual outcome of some new proposition. Some of these methods of inference mapped directly to definitions, postulates, or propositions previously established, but others appealed to an intuitive diagrammatic reasoning which the reader was presumed to have had. Euclid would simply "read off" properties from the diagram, and assume that the reader would trust that this inferential leap was indeed generally valid (and not a specious artifact of the specific diagram that was drawn). To mathematicians of this new generation, this was a damning flaw; arguments should not depend on the imprecision of drawings, but rather should be locked down in a set of sentences, each sentence either itself a premise, or derived via a sound inference rule from a previous sentence. The *Elements*, therefore, because of its reliance on "fuzzy" diagrammatic reasoning, could not necessarily be fully trusted to deliver mathematical certainty. And although the *Elements* was later recast in different, fully axiomitized systems, notably by Hilbert [2], Tarski [13], and Pasch [12], the original work, has since remained a revered, but questioned relic of a prior era in mathematics.

Recently, however, philosophers have sought to revisit Euclid's *Elements*, specifically the diagrammatic argumentation that had been so assailed. In separate work, Ken Manders, Nathanial Miller, and John Mumma ,via different, but complimentary approaches have defined and classified the use of diagrams in Euclidean proofs, and asserted reliable methods for diagrammatic inference. Most recently, building on this research, the cohort of Jeremy Avigad, John Mumma, and Edward Dean [1] have established a formal axiomatic system, *E*, which defines precisely the diagrammatic reasoning used in the *Elements* based in a simple set of relations, construction rules, and inference rules. They have proved that this system

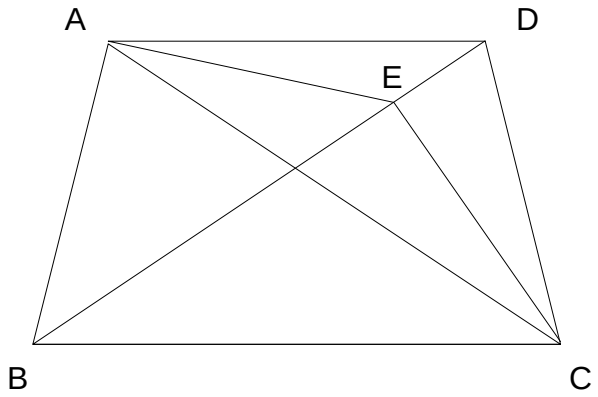
not only is both sound and complete, but also remains faithful to the essence of the Euclidean proof. In short, it re-asserts the once-questioned rigor of Euclidean diagrammatic argumentation.

The goal of this thesis is to take the system of *E* one step further, and demonstrate how diagrammatic reasoning, as defined in *E*, can be codified and executed by a computer. A software program named the E Proof Checker (EPC), built using the Java programming language, was created to this end. This program will be discussed, detailing salient elements of the implementation, interesting results from its application, and the potential for future extension.

1. Euclidean Diagrammatic Reasoning

To best understand the role of diagrammatic reasoning in Euclidean proofs, it is worthwhile to first consider an example from book I of the Elements, as provided by the Heath translation [3]:

> **Proposition I.39**: Equal triangles which are on the same base and on the same side are also in the same parallels.

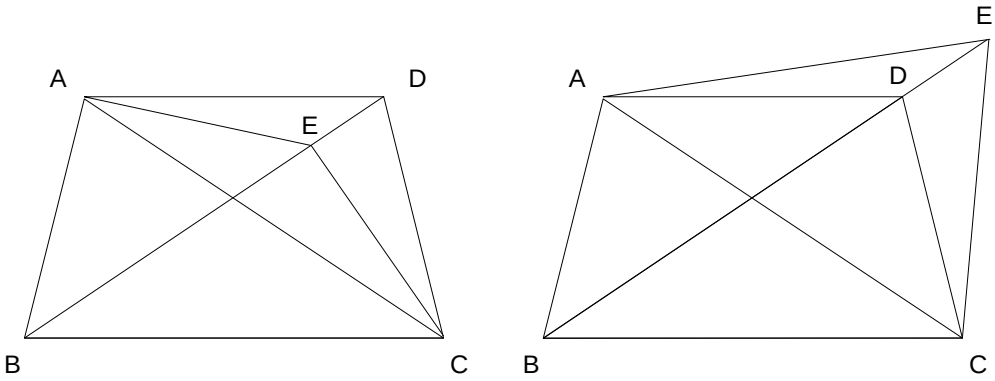


> Let ABC, DBC be equal triangles which are on the same base BC and on the same side of it; I say that they are also in the same parallels.
>
> And let AD be joined; I say that AD is parallel to BC.
>
> For, if not, let AE be drawn through the point A parallel to the straight line BC [I. 31], and let EC be joined.
>
> Therefore the triangle ABC is equal to the triangle EBC; for it is on the same base BC with it and in the same parallels. [I. 37]
>
> But ABC is equal to DBC; therefore DBC is also equal to EBC [C.N. I], the greater to the less: which is impossible.
>
> Therefore AE is not parallel to BC.
>
> Similarly we can prove that neither is any other straight line except AD; therefore AD is parallel to BC
>
> Q.E.D.

It is useful first to note the structure of the proof, as it is consistent with all others in the *Elements*. After stating the proposition to be proved, the proof begins with a *construction* phase, where new objects (e.g. points, lines, etc.), beyond those already introduced by the proposition itself, are created and configured on the diagram. For

example, in proof I.39 above, "let AE be drawn through the point A parallel to the straight line BC" is a construction step, one which augments the existing diagrammatic objects established in the proposition (i.e. triangles ABC, DBC). Next, the proof infers new conclusions (including the proposition itself) in a *demonstration* phase, through the application of postulates, common notions, previously proved propositions, and lastly "intuitive" diagrammatic reasoning. It is the latter mode of inference which roots the concern over the *Element's* robustness.

To see the diagram's role in Euclidean proofs, and to understand why it is problematic, consider again Proof I.39. After creating the initial "context" of the proof, that there are two equal triangles ABC and BDC on the same base and same side of BC, the proof instructs that a new line AE be constructed parallel to BC. At this point, however, there is a choice. Since it is not yet known that AD is parallel to BC, it is not clear whether E should be between BD or whether D should be between BE. In other words, either of these two diagrams could accurately represent the construction as it is stated in the proof.

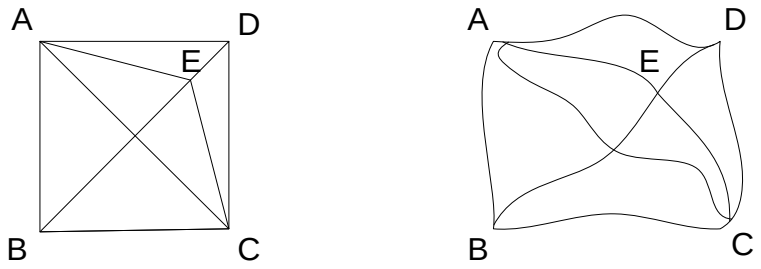


Later in the demonstration phase, it is clear that the former diagram is assumed, when the proof shows, through a reductio, the impossibility of ABC, DBC, and EBC all being equal. The tacit assumption here is that because E is between B and D, EBC is contained within DBC, and therefore must be less than and not equal to DBC. The question, therefore, is whether this general conclusion can be made, given that it depends on the particular rendering of the diagram. Had D instead been between B and E, would the reasoning still hold? Obviously, for this simple case, it is possible to consider and affirm the inferential leap, that EBC could not equal DBC, but in more difficult proofs, where there are a greater number of particular diagrams that could be drawn for a given construction, can general properties like this be inferred just from looking and reading off properties of the diagram? What is to say that the property that is read from the diagram is not some unique feature of that particular drawing, and would not necessarily hold true if the diagram was drawn slightly differently?

This is the problem of generality. Particular drawings are constructed, and then general conclusions are made based on them. The diagram therefore is not a mere pedagogical accoutrement, but an actual basis for inference. Philosophers since Hilbert have thus attacked the role of the diagram in Euclidean proofs, and sought to extricate the former from the latter. Most notably Hilbert, Peano, and Pasch all created informal axiomatizations of the *Elements*, leading later to Tarski's formal axiomatization. The conclusions of the *Elements* were preserved, but the argumentation based on diagrams was not. The question stood, however, if the diagrammatic reasoning used in the *Elements* was so flawed, how did it stand with such stability for so long?

It is here that Manders [5] makes a significant discovery, one which eventually led to the restoration of Euclidean diagrammatic reasoning. Manders distinguished between the two types of properties attributed to objects in a Euclidean proof. The first type of property, Manders labels "exact", and refers to specific magnitudes of objects (e.g. angles, line segments) used for the purpose of comparison or determining equality. For example, in the proof above, "let ABC, DBC be equal triangles" would be an example of such an exact attribution. The other type of property, which Manders calls "co-exact", refers to those topological features of the diagram which dictate spatial relationships between objects (e.g. containment, etc.). Again, inspecting the proof above, the statement that "E is between B and D" would be a co-exact property of the diagram.

What is consequential about this distinction is that Manders shows that only co-exact properties are used in the diagrammatic inferences in Euclidean arguments. For example, a proof would never read from the diagram that two line segments are congruent, but rather would depend on the argument of the text for such an attribution. Conversely, however, a proof *may* use the diagram to convey some topological property (e.g. that a point resides within a circle), and trust in the intuitive diagrammatic reasoning of the reader to know that the property is indeed correct and general to all diagrams. Therefore, so long as the objects rendered in the diagram relate to each other in space in a way that is consistent with the construction, regardless of whether the ruler or compass was used precisely, the conclusions should still hold generally. For example, the diagrams below would each suffice for the proof of proposition I.39, despite their differences or in-exactness, since all that is necessary is for the diagram to show the topological property EBC is contained within DBC. That the lines AD and BC are not exactly parallel is unimportant, from the perspective of what the diagram needs to convey. In other words, the proof works for either rendering, since they are "co-exact equivalent.



Knowing that Euclidean proofs refer only to the diagram for co-exact properties, it then suffices to show that there exists some sound procedure governing the topological inferences. Both Nathanial Miller and John Mumma do exactly this.

Miller, in a Formal Geometry (FG), [8] ensures generality of diagrammatic inferences by considering, for each construction step, every different topological possibility. For example, in the proof of I.39, the diagram would split into two separate cases, where E is between B and D, and where D is between B and E (as depicted in the figure shown prior). Given that subsequent demonstration steps in the proof work in *all* cases, then the conclusion could be assured to be valid. Essentially, Miller's case splits combine possible diagrams in disjunctive normal form, where each case must be considered. And while this approach does in fact provide a mechanical procedure for ensuring generality in Euclidean proofs, Mumma notes two flaws. First, there still exists no general method for deriving the different topological possibilities for a construction – the reader is left to his intuition to guarantee that all cases have been considered. Second, for non-trivial proofs, the number of cases to consider can be over-whelming, as all cases are explicated, even ones with features not necessarily germane to the proof. In the end, although FG can justify generality, it seems less faithful to the Euclidean diagrammatic argument.

Mumma, in his PhD thesis [9], also tackles this same problem of generality, though takes a path different than Miller. Rather than considering case splits at constructions, Mumma, through a diagram-based proof system *Eu*, justifies the generality of diagrammatic inferences by first determining whether diagrams are equivalent with respect to their topological, or co-exact, properties. It is important to note that equivalence depends not just on the end diagram, but also the construction steps that produced it, since a particular diagram could be created via different constructions. In *Eu*, therefore, proofs are reduced to conditionals in the form:

$\sum 1, D1, \blacktriangleright 1 \rightarrow \sum 2, D2, \blacktriangleright 2$

where $\sum$ is the diagram, D the metric information, and $\blacktriangleright$ a partial ordering (for handling constructions). In other words, given some diagram with exact and co-exact properties, a new diagram follows, with additional diagrammatic objects and

properties. Through *Eu*, generality of diagrammatic inferences can be secured, all while preserving the role of the diagram.

Building on *Eu*, the team of Avigad, Mumma, and Dean (referred to hereafter as AMD), created a new formal system *E*, which also allows for valid diagrammatic inference, but at a higher level of abstraction. In *E*, a diagram is not a visual object as in *Eu*, but rather is merely an assemblage of diagrammatic "relations" (e.g. intersects, between, etc.) ranging over a set of elements (e.g. points, lines, etc.) that together are generally valid. For example, a simple diagram could be merely a single fact: "point *a* is on line *L*". Diagrams in *E* are built using a set of construction rules that are quite similar to those used in original Euclidean proofs. From a constructed diagram, inference rules are then used to demonstrate the "direct diagrammatic consequences, essentially identifying those properties that generally hold for a specific diagram. As in Eu, properties are either exact ("metric" in *E*) or co-exact ("diagrammatic" in *E*) facts. In this way, through the use of a simple set of diagram-based construction and inference rules, *E* delivers Euclidean proofs with the robustness of the formal axiomitizations of the *Elements*, but in a style that is faithful to their original form. In the next section, this system of *E* is explored in detail.

2. Analysis of Diagrammatic Reasoning

This section will describe the language, construction rules, and inference rules of *E*, and how these combine into proofs powerful and expressive enough to reach the geometric propositions of the *Elements*.

2.1 Language

There are six "elements" in the language of E, the first three diagrammatic and the latter three metric:

- *point*: with variables a, b, c ranging over
- *line*: with variables L, M, N ranging over
- *circle*: with variables α, δ, ε ranging over
- *segment(a, b)*: the length of the segment from a to b
- *angle(a, b, c)*: the magnitude of the angle abc, written <abc
- *area(a, b, c)*: the area of the triangle abc, written Δabc

Operating on these elements are nine basic relations. Note that the last three can all be overloaded, such that they may be used on different types of elements.

- *on(a, L)*: point a is on L
- *same-side(a, b, L)*: points a and b are on the same side of L
- *between(a, b, c)*: points a, b, and c are collinear and b is between a and c
- *on(a, ε):* point a is on circle ε
- *inside(a, ε):* point a is inside circle ε
- *center(a, ε):* point a is the center of circle ε
- *intersects(x, y):* where x and y are of the same "type" and are diagrammatic elements (i.e. point, line, or circle), x intersects y (e.g. line L intersects line M)

- *equals(x, y)*: where x and y are of the same type, if diagrammatic x and y are the same (e.g. point a is the same as point b), or if metric x and y are congruent or of the same magnitude (e.g. segment ab is congruent to segment ba)

- *less-than(x, y)*: where x and y are metric, x is less than y (e.g. segment ab is less than segment bc)

In addition to these relations, two constants are defined:

- *zero*: represented as 0, for metric relations

- *right-angle*: for angles, used as <abc = right-angle

The language of *E* includes one function, +, operating on metric elements (segment, angle, area). And finally the logical negation ! is defined as is normal, such that a *literal* is defined as an atomic formula or the negation of an atomic formula.

On top of the basic relations of *E*, a number of other more powerful relations may be built, however it is important to note that because of their derivative nature, they are considered "syntactic sugar" rather than fundamental to the language of *E*. Examples of more complex relations include:

- *diff-side(a, b)*: can be written as !on(a, L), !on(b, L), and !same-side(a, b, L)

- *outside(a, ε):* can be written as !inside(a, ε) and !on(a, ε)

2.2 Construction Rules

As described in Section 1.1, Euclid uses a construction stage to introduce new objects having specific properties and add them to the diagram. In proofs in *E*, these are accomplished via a set of construction rules, whereby each rule carries both a set of preconditions (i.e. diagrammatic facts that must be true in order to invoke the rule) and also a set of conclusions (i.e. diagrammatic facts that can be asserted). For example, to place a point on a diagram that sits on the same side of a specific line as another point, an *E* proof would write:

> Let *a* be a point on the same side of *L* as *b*.

The prerequisite here is that *b* is not on *L*, and if this precondition held, then the proof would introduce the new point *a*, and also conclude the same-side relation, namely that *a* is on the same side of *L* as *b*.

It is useful to note that one rule may yield many atomic diagrammatic assertions. For example, a rule which extends a line segment *bc* to point *a* would conclude not only that *a* is on this line, but also that *c* is between *a* and *b*. In this way, construction rules serve as a layer of abstraction on top of the more base diagrammatic relations. AMD note that although technically it would be possible for E to jettison the rules and instead allow proofs to flexibly introduce any object which satisfies relational constraints consistent with the diagram, this approach would add an unsatisfying degree of complexity to the proof, inconsistent with the Euclidean style, and unintuitive from the reader's perspective.

The complete set of construction rules defined in *E* is documented in Appendix A.

2.3 Inference Rules

Recall that in a Euclidean proof's demonstration stage, new facts are deduced from the elements and properties established in the construction stage. In *E*, these "direct diagrammatic consequences" are justified via a set of inference rules. Rules in *E* are of three types:

- *Diagrammatic*: generalities, between, same side, Pasch, triple-incidence, circle, intersection, equality
- *Metric*: equality, ordering, addition
- *Transfer*: segment, angle, area

The first two sets of rules, diagrammatic and metric, take facts of that type and make assertions of that same type (i.e. diagrammatic to diagrammatic, or metric to metric). For example, the second Betweenness Rule infers the diagrammatic relation "on", assuming three other diagrammatic relations are satisfied:

> If *b* is between *a* and *c*, *a* is on *L*, and *b* is on *L*, then *c* is on *L*

Transfer rules, on the other hand, take facts of one type and assert a fact of the opposite type (i.e. diagrammatic to metric or vice-versa). For example, the first Diagram Segment transfer rule takes a diagrammatic relation and concludes a metric one:

> If *b* is between *a* and *c*, then $ab + bc = ac$

It is important to note that all inference rules come in a simple "if-then" form, with a conjunction of relations in the antecedent, and a single relation in the consequent. In other words:

> if $\alpha 1, \alpha 2, \ldots, \alpha n$ then δ

In some cases, this convention seems to be ignored, as there are conjunctions in the consequent. For example the second Circle rule:

> If a is inside α or on α, *c* is not inside α, and *c* is between *a* and *b*, then *b* is neither inside α nor on α.

However, in this case, the rule can simply be split into two:

> If a is inside α or on α, *c* is not inside α, and *c* is between *a* and *b*, then *b* is not inside.

…and…

> If a is inside α or on α, *c* is not inside α, and *c* is between *a* and *b*, then *b* is not on α.

The importance of this is detailed in the next section. Further, AMD provide a richer discussion of the nature, origin, and intention of the rules, but these explanations lie outside of the scope of this thesis. For the purposes here, the inference rules of E are merely presented in Appendix B.

2.4 Direct Diagrammatic Consequence

From the elements, relations, construction rules and inference rules of *E*, a definition of proof and diagrammatic inference can be framed. In simplest terms, a proof in E establishes that some new assertions for a set of points, lines, and circles can be produced given some existing set of diagrammatic facts. In logical form, AMD model a proof as:

> $\Gamma \Rightarrow \exists b,M,\beta\ \Delta$

…where Γ and Δ are literals of *E* (e.g. on, same-side) satisfied for a set of points, lines, and circles, and b,M,β a set of points, lines, and circles, not existing in Γ, and which do exist in the assertions satisfied in Δ.

What warrants the new diagrammatic assertions are the inference rules, such that each new assertion is a "direct consequence" of the application of a specific inference rule on the existing elements and relations. AMD note that the notion of direct consequence in E was designed to satisfy a number of key considerations. First,

it should remain faithful to the Euclidean style, in the sense that it neither infers too much (i.e. takes large inferential leaps, beyond what would be explicitly explained in the *Elements*) nor too little (i.e. needs many steps to demonstrate that which would be a single step a Euclidean proof). Second, it should be both sound and complete with respect to the set of first order consequences of the inference rules. And finally, it should be "computationally tractable", such that a computer could automate the inference for simple proofs in a reasonable amount of time.

To best explain the notion of direct consequence in *E*, first recall that all inference rules are all presented in rule form:

if $\alpha 1$, $\alpha 2$, ..., αn then δ

AMD note, however, that contrapositive variants of these rules should also be considered as direct consequences. For example, take the simple Generalities rule:

If *a* is the center of α then *a* is inside α

Obviously if the relation center(a, α) holds, then inside(a, α) would be a direct consequence. However, inverting the rule should also be valid:

If *a* is not inside α then *a* is not the center of α

Generalizing then, each individual rule is simplified into a simple disjunction:

not $\alpha 1$, not $\alpha 2$, ..., not αn, δ

From this disjunction, each possible variant, in rule-based form (i.e. "if-then"), can be applied to deliver a diagrammatic assertion. Direct consequence, therefore, is obtained by closing under the set of these inferences.

2.4 Proofs

Finally, with the language and rules defined, and a notion of direct consequence in place, AMD demonstrate how proofs in the *Elements* can be replicated in *E*. As an example, they consider the problem from Proposition 1 in the first book of the *Elements*, where Euclid demonstrates how to construct an equilateral triangle.

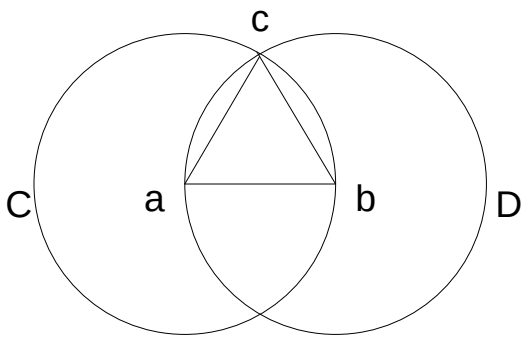


Figure 4.1.a: *Proposition I.1*

ADM translate this proof in the language of *E* to be:

> Let α be a circle with center *a* passing through *b*.
> Let *β* be a circle with center *b* passing through *a*.
> Let *c* be a point on the intersection of α and *β*.
> Have *ab* = *bc* [since they are radii of α].
> Have *ba* = *bc* [since they are radii of *β*].
> Hence *ab* = *bc* and *bc* = *ca*.

As is typical in the *Elements*, the proof begins with a construction stage, in this case using the valid construction rules of *E*, where diagrammatic properties are asserted. After this, a demonstration follows, where metric or diagrammatic assertions (in this case only metric) are made. Note that in *E*, assertions are made using the words "hence" (to introduce assertions that follow from previous metric assertions) or "have" (to introduce assertions that follow from the diagram). All assertions are grounded in some specific direct consequence (i.e. inference rule). As is seen here, the end result, that *ac*, *ab*, and *bc* are all equal is obtained through the application of two Circle transfer rules and a simple metric one. In the next chapter, it will be shown how diagrammatic inferences such as these can be automated by a computer.

3. Implementation

This chapter will introduce the E Proof Checker (EPC), a software program which automates diagrammatic reasoning, as codified in the language of *E*. Section 3.1 will first give a high-level account of the program, describing what it does, how the user interacts with it, and the extent to which it captures the language of *E*. From here, Section 3.2 and 3.3 will delve deeper into the implementation, highlighting the salient data structures and algorithms used. Lastly, Section 3.4 will discuss some of the limitations of the EPC, and identify future points of extension.

3.1 Program Overview

The EPC is a Java-based software program which allows users to build and validate proofs in the language of E. Recall that proofs in E consist of two distinct stages. In the construction stage, geometric elements (e.g. points, lines, etc.) are introduced and composed into a structure which adheres to certain metric or diagrammatic properties, using a pre-defined set of construction rules. Next, in the demonstration stage, inference rules are invoked to generate new information, to the eventual conclusion of the proposition itself. Correspondingly, the EPC consists of two core modules, a Parser which accepts user-input in the form of construction rules of E, and an Inference Engine which validates diagrammatic demonstrations, via a simple saturation algorithm (that is, the repeated application of modus ponens on the inference rules of E).

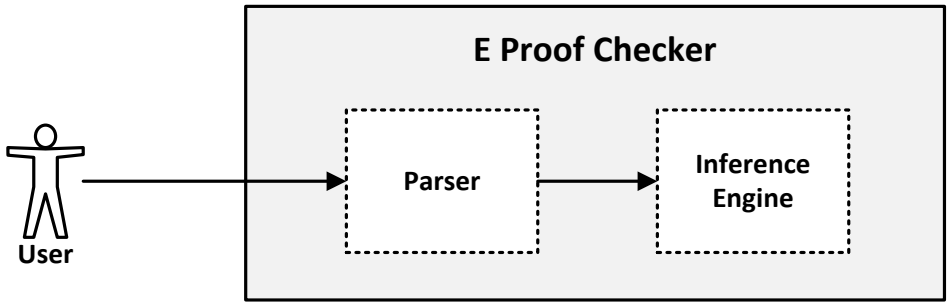


Figure 3.1.a: *EPC Program Structure*

To start, users of the EPC create a text file with the E proof in it. For example, consider a simple proof:

```
Let a be a point on L.
Let c be a point on L distinct from a.
```

```
Let b be a point on L between a and c.
Hence a and b are not on the same side of L.
```

This EPC is then run, passing this text file as input.

```
%> java EPC > simple-proof.txt
```

The EPC then reads in each construction step, one-by-one. For each, it checks the various preconditions as specified by E, and then passes the new diagrammatic assertions to the inference engine. For example:

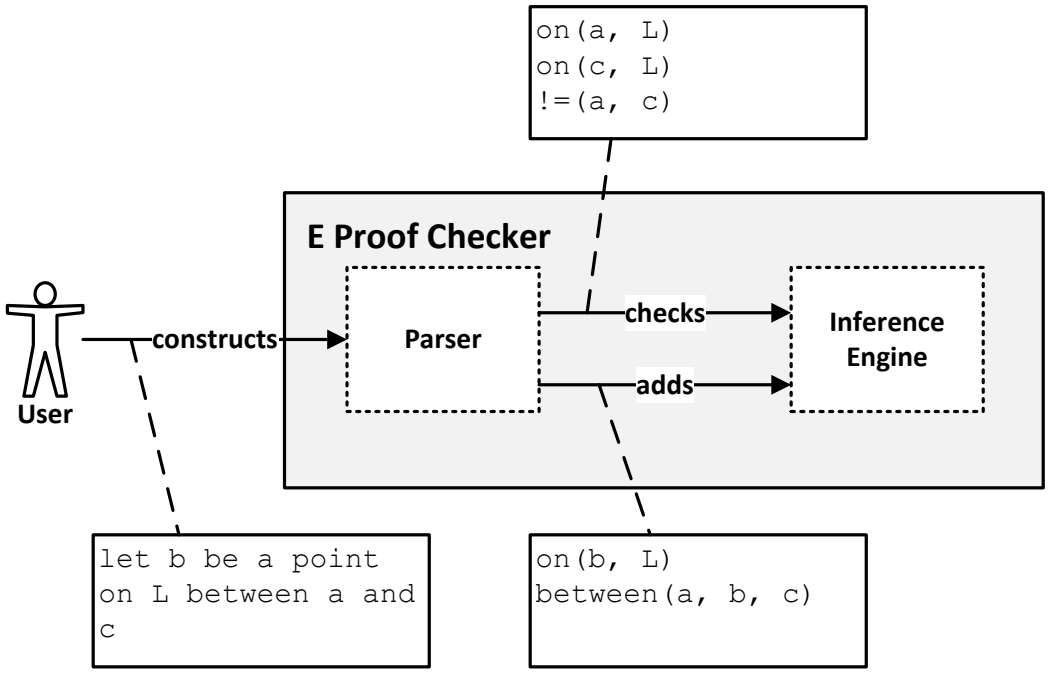


Figure 3.1.b: *Parsing a Construction Rule*

As shown in the figure above, the Parser is responsible for three main tasks. First, it checks whether input supplied by the user is a well-formed construction rule in the language of *E*. Next, it strips out the variables in the construction, and checks whether the pre-conditions specified for that construction hold in the diagram thus far generated. And finally, assuming the first two steps complete successfully, it adds one or many new lower-level diagrammatic relations to the Inference Engine.

At this point, as each new diagrammatic assertion is made, the EPC uses a forward-chaining algorithm, which is to say it uses the repeated application of modus ponens given the inference rules defined in E, to generate a set of entailed diagrammatic facts. It is important to note that the algorithm is not goal-directed, but rather generates *all* possible diagrammatic assertions that can be reached given the inference rules and

available facts after each construction step, whether or not they will be helpful in the demonstration stage later. For example, consider again the proof given above.

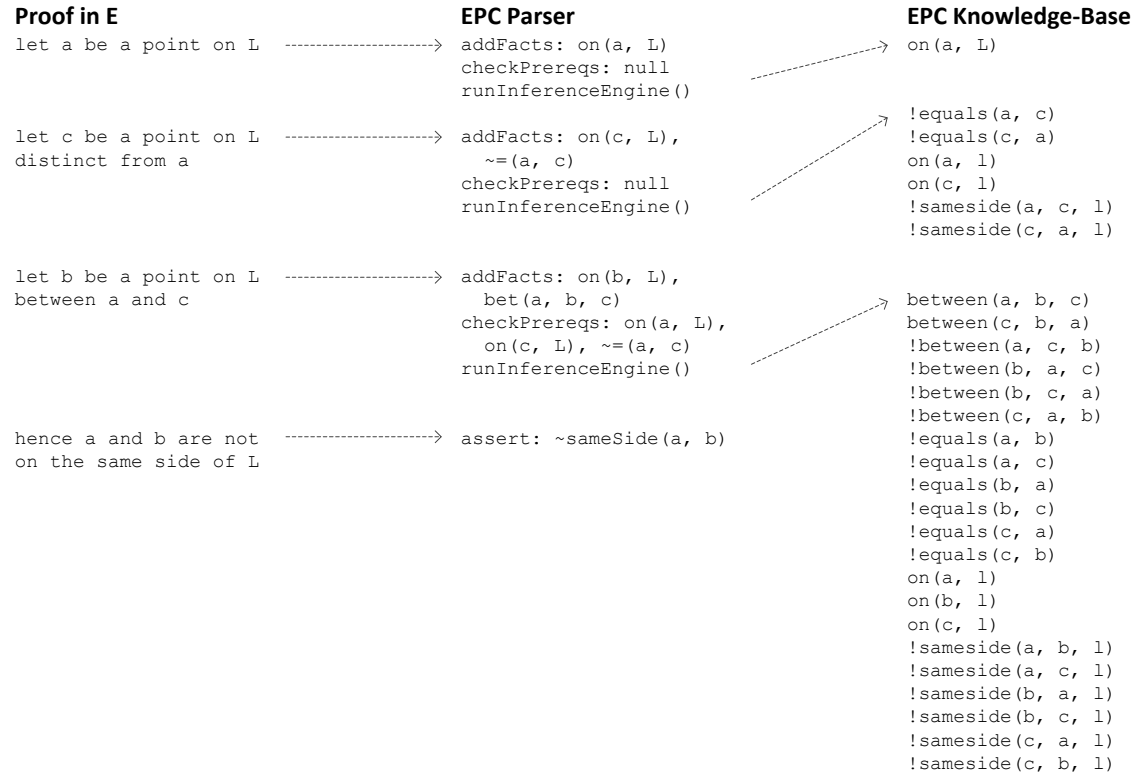


Figure 3.1.c: *Forward-chaining algorithm*

Finally, after each construction has been read and all entailments have been generated, the parser interprets and checks the demonstrations. In the language of E, "hence" is used to preface diagrammatic assertions and "have" metric assertions. In the EPC, because entailments are generated at each step, a demonstration is merely a check against this pre-generated set of assertions. The algorithm outlined here is discussed in greater detail in section 3.3, and is built on a set of fundamental data structures, which is the focus of the next section.

3.2. Data Structures

The EPC was developed in an Object-Oriented style, which is to say it uses data structures which resemble closely the "building blocks" of *E*: elements, relations, and rules. Note that in this section, the word "class" will refer to the Object-Oriented term, which represents a data structure and its associated behavior. Also note that when an element is capitalized (e.g. "Point" vs. "point"), it will refer to this "class" (i.e. data

structure) in the EPC, and not the term in language of *E* (e.g. Point is the class in the EPC whereas point is an element in *E*).

At the foundation of the EPC are classes for the three basic diagrammatic elements in E: Point, Line, and Circle. Each of these classes has two attributes: its identifier and a flag for whether it is to be considered free or bound. For example, a point referred to in the construction "`let a be a point on L`" would map to a class of type Point which is bound and has id="a", and a Point referred to in the rule "inside(a, C) → !on(a, C)" would map to a class of type Point which is free and has id="a". The definition of the Point class, simplified, is given below:

```
class Point extends Element {
      String id;
      Boolean isFree;
}
```

The classes for Line and Circle are identical. It is useful to note here that metric "elements" (i.e. segment, angle, and area), are not represented as Element classes in the EPC. The justification and explanation for this is given in Section 3.4.

Building on top of Elements are Relations. Each relation in *E* is a separate class in the EPC, one which contains some number Elements. For example, the "between" relation in *E* is represented in the EPC as the following class:

```
class Between extends Relation {
      Point pLeft;
      Point pBetween;
      Point pRight;
      Boolean negation;
}
```

All Relations contain a flag indicating whether it is an affirmative or a negation. In other words, `!on(a, L)` and `on(a, L)` are separate instances of Relations in the terms of the EPC. This detail may seem slightly counter-intuitive, since in logical terms, the negation is not typically considered a part of the atomic relation, but rather outside of it. However, because the user interacts only via the construction rules, this implementation artifact is not exposed, and so has no real deleterious effect.[1]

Finally, Elements and Relations assemble to form Rules in the EPC. Recall that all inference rules in *E* can be simplified into disjunctive normal form. So, for example, the Same Side 1 rule:

[1] Note that storing a relation and its negation as separate instances is a common approach in theorem provers.

```
!on(a, L) → same-side(a, a, L)
```

…would simplify as follows…

```
on(a, L) or same-side(a, a, L)
```

Given that rules are just disjunction of literals, a Rule in the EPC is simply a generic container of Relations:

```
class Rule {
      Relation relations[];
}
```

Instances of specific Rules are created by simply initializing it with the appropriate Relations of free Elements. For example, the Same Side 1 rule above would be instantiated as:

```
boolean FREE = true;
boolean NEGATED = true;
Point a = new Point("a", FREE);
Line l = new Line("l", FREE);
On on = new On(a, l, !NEGATED);
SameSide ss = new SameSide(a, a, L, !NEGATED);
Rule ss1 = new Rule(on, ss);
```

Each Rule resolves into many Formulas, where a Formula is simply a single or conjunction of Relations as an antecedent and a single Relation as a consequent. In the EPC, therefore, a Formula is a simple class with two members, a container of Relations and a single Relation, for the antecedent and consequent, respectively:

```
class Formula {
      Relation antecedent[];
      Relation consequent;
}
```

So, for example, taking again the Same Side 1 rule:

```
on(a, L) or same-side(a, a, L)
```

…would be translated into two formulas:

```
!on(a, L) → same-side(a, a, L)

!same-side(a, a, L) → on(a, L)
```

As Formulas are used by the EPC to generate new Relations (given the existing Relations supplied by the construction stage and the Rules), and these new bound

Relations are then stored in a central Knowledge Base, which is implemented simply as an array of Relations (whose elements would all be bound).

```
Relation[] knowledgeBase;
```

Finally, putting it all together, the following diagram depicts these fundamental building blocks, Element, Relation, Rule, and Formula, and shows how on top of these, the EPC can capture diagrammatic facts and verify proofs:

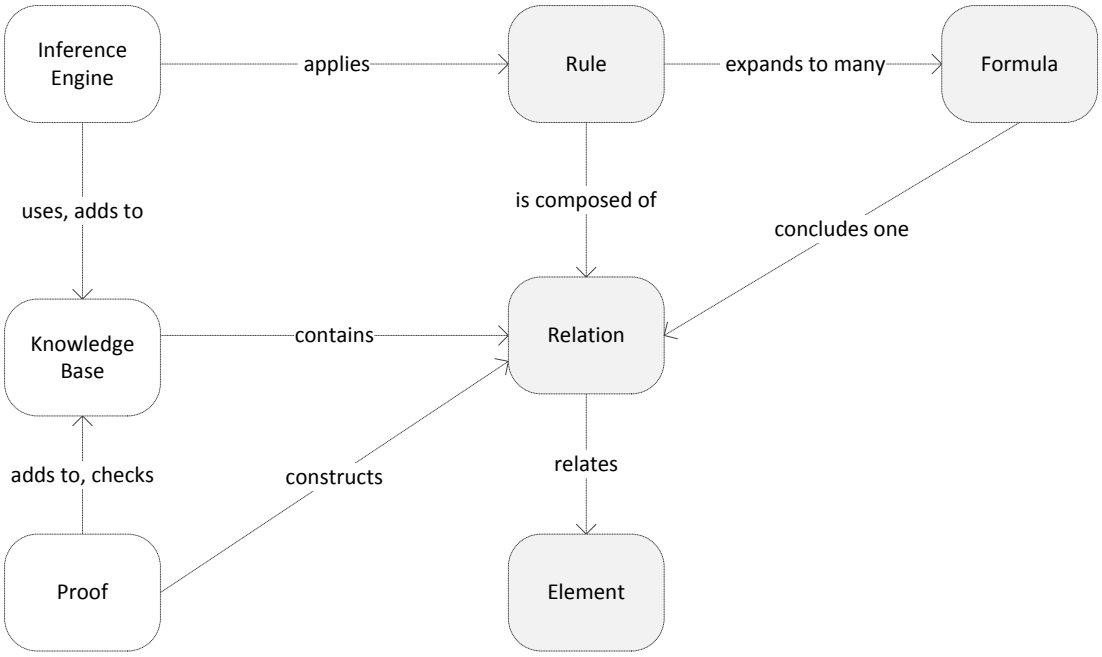


Figure 3.2.a: *EPC Semantic Map*

Summarizing the diagram above, Proofs, in their construction stage, build a set of Relations, and add them to the Knowledge Base. After each addition of a new Relation, the Inference Engine iterates through a set of rules, expands them to Formulas, and then applies these to the Relations already stored in the Knowledge Base. If a Formula concludes a new Relation, this new Relation is then added to the Knowledge Base. The set of all Relations stored in the Knowledge Base is then used to verify the demonstrations in proofs, determining whether it is in fact valid. Exactly how this is implemented in the EPC is discussed in the next section.

3.3 Inference Algorithm

As described in the sections above, the EPC inference engine uses a forward-chaining algorithm to recursively apply the rules defined in E to the relations stored in the knowledge base. In pseudo-code, the algorithm is simply:

```
1: PROOF applies CONSTRUCTION

2: New RELATION added to KNOWLEDGE BASE

3: INFERENCE ENGINE loops through RULES

4:    RULE is expanded to many FORMULAS

5:    If new RELATION fits in antecedent of FORUMLA

5:          RELATIONS from KNOWLEDGE BASE are substituted into FORMULAS

5:          If FORMULA fully bound and produces new RELATION then

6:                RELATION added to KNOWLEDGE BASE

7:                Go to step 3
```

To examine the algorithm more closely, it is best to step through an example. Assume the following facts already exist in the Knowledge Base:

```
on(a, L)
on(c, L)
```

In the language of E, the following construction is made to the diagram:

```
let b be a point not on L
```

This would be translated into the EPC-based command:

```
buildNotOn(b, L)
```

…and the relation would be added to the knowledge base:

```
!on(b, L)
```

The proof engine would then loop through all inference rules. Rules are stored by the EPC in disjunctive normal form, so, for example, the rule “Betweenness 3”:

```
bet(x, y, z) and on(x, M) and on(z, M) -> on(y, M)
```

…would be codified in the EPC as disjunction of atomic relations, where elements in relations are initially free (using the convention ‘*’ to indicate a free element):

```
!bet(*p1, *p2, *p3) or !on(*p1, *l1) or !on(*p3, *l1) or on(*p2, *l1)
```

Now, looping through rules, the set of possible formulas would be generated for each such that the antecedent is a conjunction of atomic relations and the consequent a single atomic relation. For example, Betweenness 3 would be expanded to the following four formulas:

```
bet(*p1, *p2, *p3) and on(*p1, *l1) and on(*p3, *l1) → on(*p2, *l1)

!on(*p2, *l1) and bet(*p1, *p2, *p3) and on(*p1, *l1) → !on(*p3, *l1)

on(*p3, *l1) and !on(*p2, *l1) and bet(*p1, *p2, *p3) → !on(*p1, *l1)

on(*p1, *l1) and on(*p3, *l1) and !on(*p2, *l1) → !bet(*p1, *p2, *p3)
```

For the purposes of programmatic speed, each formula would then be examined by the proof engine to determine whether a new fact could possibly be deduced by its application – in other words, does the type and sign of the relation (e.g. `on(-, -)` and "`!`") exist somewhere in the antecedent of a formula. Those formulas for which the new relation type and sign does *not* match would be pruned from the execution path. Continuing the example, the EPC would therefore keep the first three formulas as potentially fruitful, as they all have a "`!on(-,-)`" in their antecedent:

```
!on(*p2, *l1) and bet(*p1, *p2, *p3) and on(*p1, *l1) -> !on(*p3, *l1)

on(*p3, *l1) and !on(*p2, *l1) and bet(*p1, *p2, *p3) -> !on(*p1, *l1)

on(*p1, *l1) and on(*p3, *l1) and !on(*p2, *l1) -> !bet(*p1, *p2, *p3)
```

The EPC would then loop through each resulting formula, substituting the bound relation ~on(b, L) for the free relation ~on(*p2, *l1), and then substituting the elements "b" and "L" across the rest of the formula to form a new semi-bound formula:

```
on(*p1, L) and on(*p3, L) and !on(b, L) -> !bet(*p1, b, *p3)
```

Note that although an element could be bound into the consequent, it still contains two free elements, *p1 and *p3, and so could not yet deduce any new relations. It would therefore loop through the known relations in the knowledge base, and determine if by substituting any other known relation into the antecedent of the semi-bound formula it could completely bind it, and therefore conclude some new relation. Taking the first relation in the knowledge base:

```
on(a, L)
```

This relation could be bound successfully into the semi-bound formula in one of two ways, yielding both:

```
on(a, L) and on(*p3, L) and !on(b, L) -> !bet(a, b, *p3)

on(*p1, L) and on(a, L) and !on(b, L) -> !bet(*p1, b, a)
```

Still, however, the consequent would only be partially bound, and so a new relation could not be generated. The engine then would take the first of the resulting semi-bound formulas:

```
on(a, L) and on(*p3, L) and !on(b, L) -> !bet(a, b, *p3)
```

…and look to the next relation in the knowledge base:

```
on(c, L)
```

Again, this relation could be substituted into the antecedent of the semi-bound formula, and its element "c" into the consequent:

```
on(a, L) and on(c, L) and ~on(b, L) -> !bet(a, b, c)
```

At this point, the formula would be completely bound, and so, using modus ponens, the new relation:

```
!bet(a, b, c)
```

…would be added to the knowledge base. The EPC would then continue to recursively apply all other inference rules to the known relations, including the new "`!bet(a, b, c)`", to generate new diagrammatic facts, until nothing more could be gleaned.

3.4 Limitations, Constraints, and Divergences

In general, the EPC remains largely faithful to the language of *E*, as described by AMD [1]. There are, however, some deviations from the exact expression of *E*, and also some portions of *E* that were not realized in full by the EPC. This section will discuss these limitations and divergences, and describe the extent to which the EPC could, in the future, be enhanced or extended to overcome these short-comings.

Metric Relations and Rules

The first, and most significant difference, is the EPC's expression, codification, and implementation of the metric elements and rules of *E*. As previously noted in Section 3.2, the metric elements, or magnitudes (i.e. segments, angles, and areas), are not encoded as separate, first-class objects, like their diagrammatic counterparts (i.e. points, lines, and circles). In fact, within the EPC, metric elements are not actually Elements at all (from a data structure perspective). Instead, metric elements are embedded within the relations that use them. For example, the simple, non-metric equality relation for two points *a* and *b* is implemented as:

```
equals(a, b)
```

…but the equality relation for two segments *ab* and *cd* is not, as would be expected:

```
equals(ab, cd)
```

Instead, equality between segments is implemented as a separate 4-parameter relation:

```
seg-equals(a, b, c, d)
```

That *a* and *b* represent a separate element, namely the segment *ab*, is not actually "known" by the EPC. Instead, the EPC relationship between these elements is held in tact only by their arrangement in the parameter list. If, for example, the relation was instead `seg-equals(a, c, b, d)`, it would refer to the semantically different assertion *ac* = *bd*.

Similarly, the magnitudes of zero and right-angle, defined explicitly in E, are also not implemented in the EPC in a pure sense, but rather codified again in the relations that they embody. For example, comparing an angle to zero, is not implemented as:

```
equals(abc, 0)
```

…but rather via the fabricated, ternary relation:

```
angle-equals-zero(a, b, c)
```

Lastly, as evinced in the above examples, the comparison and addition symbols also come in the form of custom relations, and not implemented as primitives of the EPC. Comparing the sum of two segments to another segment, ab + bc = de, would ideally be implemented as a nesting of relations:

```
equals(plus(ab, bc), de)
```

…but is instead encoded as:

```
seg-plus-seg-equals(a, b, b, c, d, e)
```

Other additions and comparisons of magnitudes (e.g. angle equals, etc.) are similarly implemented.

The consequence of this limitation is significant, but not damning. On the positive side, this work-around does allow the EPC to leverage all of Transfer rules, albeit in this unauthentic way. For example, the first Diagram-segment transfer axiom is encoded in the EPC as:

```
bet(a, b, c) → seg-plus-seg-equals(a, b, b, c, d, e)
```

From the perspective of the end-user, there is no perceptible effect, since he interacts with the engine using the higher-level construction and assertion rules, and not via the low-level relations. The user, for example, could type the following proof:

```
Let L be a line.
Let a be a point on L.
Let c be a point on L.
Let b be a point on L between a and c.
Have a b + b c = d e.
```

…and the EPC would validate the assertion. The user would be completely ignorant of the lower-level scaffolding used to obtain the result.

The more significant, and perhaps more pernicious impact, however, is that the Metric axioms are essentially out of bounds. Because the EPC has no native ability to either add or compare magnitudes, these metric elements are often left in their more primitive, relational form. Given the example above, the user would not be able to assert a simple re-arrangement of the final demonstration like:

Have bc + ab = de.

…much less a more complex (albeit rather inconsequential) result:

Have bc + ab = de + 0.

Given that most Euclidean proofs rely on the basic metric rules (e.g. commutative, associative, or transitive) to obtain their final conclusion, the EPC is therefore a bit hamstrung - able to run most of the race, but not able to cross the finish line. This limitation will be made clear in the next chapter, as some of the propositions from the *Elements* are run through the EPC.

In general, despite the exclusion of metric rules, the EPC does still fulfill its central mission: to demonstrate how diagrammatic inferences, as defined in the language of E, can be automated by a computer. As will be shown, diagrammatic proofs are gracefully constructed in the language of E, and effortlessly executed via EPC engine.

Parser

A less consequential, but not unimportant, deviation from the language of *E* is in the phraseology of the construction rules. There are two main differences between the EPC and *E*. First, for the purpose of simplifying the implementation, minor modifications to the Intersection rules were necessary. For example, in some of the

rules where only one single intersection between two elements is possible (e.g. a line and another line), points are introduced as:

> Let *a* be the point.

…however in other rules where multiple intersections are possible, points are introduced as:

> Let *a* be a point.

Normalizing this simple difference such that all rules use the word "a" instead of "the" in their opening phrase, subtracts little from the essence of *E*, but makes the implementation significantly less complex. There are other similar modifications to the language in the rules, and they can all be found in Appendix C. Note that the other point, line, and circle construction rules remain the same.

A second deviation stems from the EPC's circumvention of metric sorts (as described above). Because magnitudes are not first class elements in the EPC, they cannot be stated as such in the construction rules. This is noticeable only in the demonstration stage, and again, is mainly cosmetic. For example, the assertion in *E*:

> Have *ab* + *cd* = *ef*.

…would instead be written for the EPC as (note the spaces):

```
Have a b + c d = e f.
```

Essentially, the magnitudes are not truly elements themselves, and so the EPC must read each constituent element individually.

Importing Previous Proof Results

In the *Elements*, propositions build on the results of preceding propositions, using their conclusions either as a foundation for construction of new diagrams or as justifications for inference. In *E*, referencing prior results like this is done outside of the construction rules, via free, descriptive text. For example, the formalized proof of Proposition I.2 begins with:

> By Proposition I.1 applied to *a* and *b*, let *d* be a point such that *d* is distinct from *a* and *b* and *ab* = *bd* and *bd* = *da*.

In the EPC, such references are not currently supported - the only way to apply another diagram and its conclusions to an existing diagram would be to manually specify the construction again in its entirety. This is not a constraint, but rather an unimplemented feature that could be added in the future.

4. Findings

In this section, it will be described how sample proofs, both from the *Elements* and contrived, can be interpreted via the EPC parser and then validated by the inference engine. In addition, the EPC has also produced other interesting findings with respect to proofs in *E*, and these will be highlighted in Sections 4.3 and 4.4.

4.1 Examples from the *Elements*

Consider again Proposition 1 in the first book of the *Elements*, where Euclid creates an equilateral triangle:

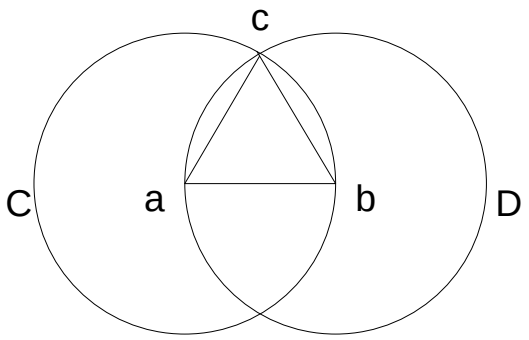


Figure 4.1.a: *Proposition I.1*

Second 2.4 gives the proof in *E*, and the proof that the EPC reads is:

```
Let a be a point.
Let b be a point.
Let C be a circle with center a passing through b.
Let D be a circle with center b passing through a.
Let c be a point of intersection of C and D.
Have a b = a c.
Have b a = b c.
```

The construction stage is nearly identical, with the exception of the variables ranging over elements, and a subtle change in the article "the" to "a" (as described in Section 3.4). The demonstration stage, however, has a few marked differences. Again, as described in Section 3.4, note that each segment is expressed as two separate "tokens" (e.g. on line 6, "`a b = a c`" rather than "`ab = ac`"). Second, and more consequential, the final step, that *ab* must equal *bc* (and vice-versa) is entirely absent from the EPC-parsed proof. This assertion would be obtained by applying two simple metric

inferences in E (namely symmetry of line segments and transitivity of equality), however, since metric relations are out of the scope of the EPC proof engine, this final result could not be obtained.

From an execution perspective, the EPC engine, as described in Chapter 3, is not goal directed, and so would not stop at the conclusion of segments *ab*, *bc*, and *ac* all being equal. Instead, it would saturate the knowledge base with all facts that could be inferred from the constructed diagram. Specifically, from the 6 relations directly built in the construction phase…

```
center(a, C)      on(a, D)          on(c, C)
center(b, D)      on(b, C)          on(c, D)
```

…22 new relations would be generated, via the application of the inference rules in E:

```
!center(a, D)     !in(c, D)         !segment-lessthan(a, b, a, c)
!center(b, C)     !in(a, D)         !segment-lessthan(a, c, a, b)
!center(c, C)     !in(b, C)         !segment-lessthan(b, a, b, c)
!center(c, D)     !in(c, C)         !segment-lessthan(b, c, b, a)
in(a, C)          intersects(C, D)  segments-equal(a, b, a, c)
in(b, D )         intersects(D, C)  segments-equal(a, c, a, b)
                  !on(a, C)         segments-equal(b, a, b, c)
                  !on(b, D)         segments-equal(b, c, b, a)
```

The EPC obtains these results nearly instantaneously, parsing the proof and generating conclusions in just over 300 milliseconds. [2]

Taking another example from the *Elements*, Proposition I.2 constructs a new segment equal to another segment.

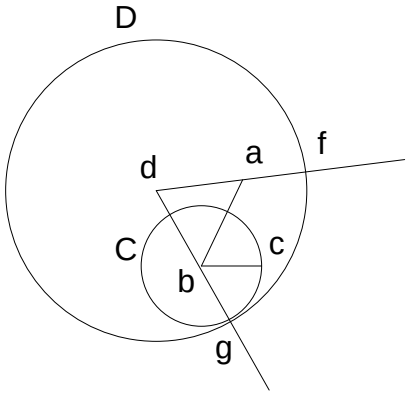


Figure 4.1.b: *Proposition I.2*

[2] All proofs were run on a Dell Laptop running Windows XP, Intel Core Duo CPU, 2.2GHz and 3GB of RAM

In [2008], AMD translate this proof in *E* to:

> By Proposition I.1 applied to *a* and *b*, let *d* be a point such that *d* is distinct from *a* and *b* and *ab* = *bd* and *bd* = *da*.
> Let *M* be the line through *d* and a.
> Let *N* be the line through *d* and *b*.
> Let α be the circle with center *b* passing through *c*.
> Let *g* be the point of intersection of *N* and α extending the segment from *d* to *b*.
> Have *dg* = *db* + *bg*
> Hence *dg* = *da* + *bg* [since *da* = *db*]
> Hence *da* < *dg*.
> Let *β* be the circle with center *d* passing through *g*.
> Hence *a* is inside *β* [since d is the center and *da* < *dg*].
> Let *f* be the intersection of *β* and M extending the segment from *d* to *a*.
> Have *df* = *da* + *af*.
> Have *df* = *dg* [since they are both radii of *β*].
> Hence *da* + *af* = *da* + *bg*.
> Hence *af* = *bg*.
> Have *bg* = *bc* [since they are both radii of α].
> Hence *af* = *bc*.

Again, the proof in *E* must be slightly modified for the EPC parser to interpret. Further, as mentioned in Section 3.4, the EPC is not capable of "importing" diagrams from other proofs, and so the construction in Proposition I.1, necessary to prove I.2, must be reiterated. For this proof in particular, the impact is relatively minor with respect to either the engine's performance or the proof's brevity.

```
// Rebuild Prop I.1
Let a be a point.
Let b be a point.
Let E be a circle with center a passing through b.
Let F be a circle with center b passing through a.
Let d be a point of intersection of E and F.

// Start Prop I.2
Let M be a line through d and a.
Let N be a line through d and b.
Let C be a circle with center b passing through c.
Let g be a point of intersection of N and C extending the segment
from d to b.
Let D be a circle with center d passing through g.
Let f be a point of intersection of D and M extending the segment
from d to a.
Have d b + b g = d g.
Have d a < d g.
Have d a + d f = d f.
```

In all, the EPC generates 198 facts for this construction, and takes just over 25 seconds to do so (for all inferences, see Appendix E). Further, using only the Diagrammatic and Transfer inferences, the EPC does fulfill a portion of the proof's demonstrations, and so takes steps toward the end goal of Proposition I.2, to show that *af* = *bc*. For example, via the Circle rules, it concludes that *a* is indeed in *D*, and then applying the Segment-transfer rules, that *da* < *dg*. Again, however, without the Metric inferences, the final conclusion of Proposition I.2 cannot be demonstrated.

4.2 Contrived Examples

Consider now a more complicated example (not from the *Elements*), in which 5 lines intersect at different points. Working from the lower-relational level (rather than via the construction rules of *E*), the diagram is constructed using the following simple "on", "between", and "equals" relations:

```
on(a, L)          on(b, L)          between(a, d, f)
on(a, M)          on(b, O)          between(b, d, c)
on(a, N)          on(g, L)          between(d, e, f)
on(c, N)          on(d, O)          notOn(c, L)
on(c, O)          on(b, P)          notEquals(a, b)
on(d, M)          on(f, P)
on(f, M)          on(g, L)
```

Visually, the diagram would look as follows:

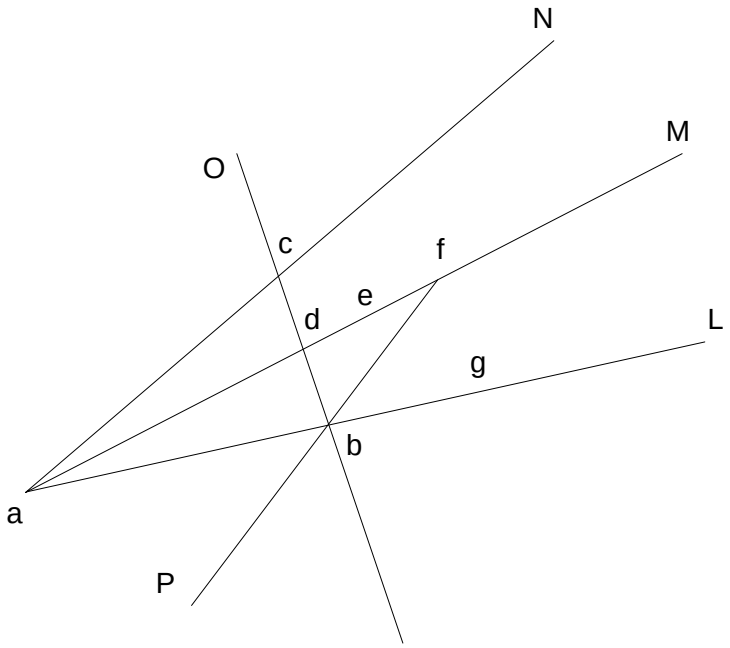


Figure 4.2.a: *Five-line intersection example*

From these constructed relations, the EPC inference engine returns 804 diagrammatic consequences in a slow 40 minutes.[3] The most interesting, are the same-side entailments, generated via a combination of Pasch, Between, and Same-Side rules.

```
sameside(a, a, O)  sameside(c, e, L)   sameside(e, d, P)
sameside(a, a, P)  sameside(c, e, P)   sameside(e, e, L)
sameside(a, c, P)  sameside(c, f, L)   sameside(e, e, N)
sameside(a, d, P)  sameside(d, a, P)   sameside(e, e, O)
sameside(a, e, P)  sameside(d, b, N)   sameside(e, e, P)
sameside(b, b, M)  sameside(d, c, L)   sameside(e, f, L)
sameside(b, b, N)  sameside(d, c, P)   sameside(e, f, N)
sameside(b, d, N)  sameside(d, d, L)   sameside(e, f, O)
sameside(b, e, N)  sameside(d, d, N)   sameside(f, b, N)
sameside(b, f, N)  sameside(d, d, P)   sameside(f, c, L)
sameside(c, a, P)  sameside(d, e, L)   sameside(f, d, L)
sameside(c, c, L)  sameside(d, e, N)   sameside(f, d, N)
sameside(c, c, M)  sameside(d, e, P)   sameside(f, e, L)
sameside(c, c, P)  sameside(d, f, L)   sameside(f, e, N)
sameside(c, d, L)  sameside(d, f, N)   sameside(f, e, O)
sameside(e, c, L)  sameside(e, a, P)   sameside(f, f, L)
sameside(e, c, P)  sameside(e, b, N)   sameside(f, f, N)
sameside(e, d, L)  sameside(e, d, N)   sameside(f, f, O)
sameside(c, d, P)
```

[3] Note that a small set of more complex rules are responsible for much of the processing. If, for example, the intersection and transfer rules are pruned, the exact same-side and between relations are produced in only 20 minutes.

4.3 Proving Pasch 3 from the other Rules

An interesting finding produced by the EPC engine was a redundancy in the Pasch rules. As originally defined in E, Pasch 3 stated:

> If *b* is between *a* and *c* and *b* is on *L* then *a* and *c* are not on the same side of *L*

Codified into the relational structure of the EPC program, the rule looks like this:

```
between(a, b, c) & on(b, L) → !sameside(a, c, L)
```

Now, excluding Pasch 3 from the set of all diagrammatic rules, the EPC was still able to reach the conclusion, that *a* and *c* are not on the same side of *L*, from the same two facts, that *b* is between and *c* and that *b* is on *L*, via a different, albeit more circuitous route. The proof is as follows:

1)	between(a, b, c)	Construction
2)	on(b, L)	Construction
3)	on(b, L) → !sameside(b, c, L)	Same Side 3
4)	!sameside(b, c, L) → !sameside(c, b, L)	Same Side 2
5)	between(a, b, c) → between(c, b, a)	Betweenness 1
6)	between(c, b, a) & !sameside(c, b, L) → !sameside(c, a, L)	Pasch 1
7)	!sameside(c, a, L) → !sameside(a, c, L)	Same Side 2

What the EPC finds, is that because of E's semantics of the "same side" relation, specifically that if a point is on a line it cannot be on the same side of that line with any other point, it is not necessary to state Pasch 3 explicitly.

Appendix A: Construction Rules of E

The following are the construction rules in the language of E, as defined in [2008]:

Points

1. Let a be a point [distinct from . . .].
   Prerequisites: none
   Conclusion: [a is distinct from. . .]

2. Let a be a point on L [distinct from . . .].
   Prerequisites: [L is distinct from lines. . .]
   Conclusion: a is on L, [a is distinct from. . .]

3. Let a be a point on L between b and c [distinct from . . .].
   Prerequisites: b is on L, c is on L, b != c, [L is distinct from lines . . .]
   Conclusion: a is on L, a is between b and c, [a is distinct from. . .]

4. Let a be a point on L extending the segment from b to c [with a distinct from. . .].
   Prerequisites: b is on L, c is on L, b != c, [L is distinct from lines . . .]
   Conclusion: a is on L, c is between b and a, [a is distinct from. . .]

5. Let a be a point on the same side of L as b [distinct from. . .]
   Prerequisite: b is not on L
   Conclusion: a is on the same side of L as b, [a is distinct from. . .]

6. Let a be a point on the side of L opposite b [distinct from. . .]
   Prerequisite: b is not on L.
   Conclusion: a is not on L, a is on the same side of L as b, [a is distinct from. . .]

7. Let a be a point on α [distinct from . . .].
   Prerequisite: [α is distinct from other circles]
   Conclusion: a is on α, [a is distinct from. . .]

8. Let a be a point inside α [distinct from . . .]
   Prerequisites: none
   Conclusion: a is inside α, [a is distinct from. . .]

9. Let a be a point outside α [distinct from . . .].
   Prerequisites: none
   Conclusion: a is outside α, [a is distinct from. . .]

Lines and Circles

1. Let L be the line through a and b.
   Prerequisite: a != b
   Conclusion: a is on L, b is on L

2. Let α be the circle with center a passing through b.
Prerequisite: a != b
Conclusion: a is the center of α, b is on α

Intersections

1. Let a be the intersection of L and M.
Prerequisite: L and M intersect
Conclusion: a is on L, a is on M

2. Let a be a point of intersection of α and L.
Prerequisite: α and L intersect
Conclusion: a is on α, a is on L

3. Let a and b be the two points of intersection of α and L.
Prerequisite: α and L intersect
Conclusion: a is on α, a is on L, b is on α, b is on L, a != b

4. Let a be the point of intersection of L and α between b and c.
Prerequisites: b is inside α, b is on L, c is not inside α, c is not on α, c is on L
Conclusion: a is on α, a is on L, a is between b and c

5. Let a be the point of intersection of L and α extending the segment from c to b.
Prerequisites: b is inside α, b is on L, c != b, c is on L.
Conclusion: a is on α, a is on L, b is between a and c

6. Let a be a point on the intersection of α and β.
Prerequisite: α and β intersect
Conclusion: a is on α, a is on β

7. Let a and b be the two points of intersection of α and β.
Prerequisite: α and β intersect
Conclusion: a is on α, a is on β, b is on α, b is on β, a != b

8. Let a be the point of intersection of α and β, on the same side of L as b, where L is the line through their centers, c and d, respectively.
Prerequisites: α and β intersect, c is the center of α, d is the center of β, c is on L, d is on L, b is not on L
Conclusion: a is on α, a is on β, a and b are on the same side of L

9. Let a be the point of intersection of α and β, on the side of L opposite b, where L is the line through their centers, c and d, respectively.
Prerequisite: α and β intersect, c is the center of α, d is the center of β, c is on L, d is on L, b is not on L
Conclusion: a is on α, a is on β, a and b are not on the same side of L, a is not on L.

Appendix B: Inference Rules of E

The following are the inference rules (metric, diagrammatic, and transfer) defined in [2008] by AMD:

Generalities

1. If a != b, a is on L, and b is on L, a is on M and b is on M, then L = M.

2. If a and b are both centers of α then a = b.

3. If a is the center of α then a is inside α.

4. If a is inside α, then a is not on α.

Between

1. If b is between a and c then b is between c and a, a != b, a != c, and a is not between b and c.

2. If b is between a and c, a is on L, and b is on L, then c is on L.

3. If b is between a and c, a is on L, and c is on L, then b is on L.

4. If b is between a and c and d is between a and b then d is between a and c.

5. If b is between a and c and c is between b and d then b is between a and d.

6. If a, b, and c are distinct points on a line L, then then either b is between a and c, or a is between b and c, or c is between a and b.

7. If b is between a and c and b is between a and d then b is not between c and d.

Same Side

1. If a is not on L, then a and a are on the same side of L.

2. If a and b are on the same side of L, then b and a are on the same side of L.

3. If a and b are on the same side of L, then a is not on L.

4. If a and b are on the same side of L, and a and c are on the same side of L, then b and c are on the same side of L.

5. If a, b, and c are not on L, and a and b are not on the same side of L,then either

Pasch

1. If b is between a and c and a and c are on the same side of L, then a and b are on the same side of L.

2. If b is between a and c and a is on L and b is not on L, then b and c are on the same side of L.

3. If b is between a and c and b is on L then a and c are not on the same side of L.

4. If b is the intersection of distinct lines L and M, a and c are distinct points on M, a != b, c != b, and a and c are not on the same side of L, then b is between a and c.

Triple incidence axioms

1. If L, M, and N are lines meeting at a point a, and b, c, and d are points on L, M, and N respectively, and if c and d are on the same side of L, and b and c are on the same side of N, then b and d are not on the same side of M.

2. If L, M, and N are lines meeting at a point a, and b, c, and d are points on L, M, and N respectively, and if c and d are on the same side of L, and b and d are not on the same side of M, and d is not on M and b != a, then b and c are on the same side of N.

3. If L, M, and N are lines meeting at a point a, and b, c, and d are points on L, M, and N respectively, and if c and d are on the same side of L, and b and c are on the same side of N, and d and e are on the same side of M, and c and e are on the same side of N, then c and e are on the same side of L.

Circle

1. If a, b, and c are on L, a is inside α, b and c are on α, and b != c, then a is between b and c.

2. If a and b are each inside α or on α, and c is between a and b, then c is inside α.

3. If a is inside α or on α, c is not inside α, and c is between a and b, then b is neither inside α nor on α.

4. Let α and β be distinct circles that intersect in distinct points c and d. Let a be a the center of α, let b be the center of β, and let L be the line through a and b. Then c and d are not on the same side of L.

Intersection

1. If a and b are on different sides of L, and M is the line through a and b, then L and M intersect.

2. If a is on or inside α, b is on or inside α, and a and b are on different sides of L, then L and α intersect.

3. If a is inside α and on L, then L and α intersect.

4. If a is on or inside α, b is on or inside α, a is inside β, and b is outside β, then α and β intersect.

5. If a is on α, b is in α, a is in β, and b is on β, then α and β intersect

Equality

1. x = x *(where x ranges over any element – e.g. point, line, etc.)*

2. If x = y and ϕ(x), then ϕ(y) *(where ϕ can be any atomic formula)*

Metric

- + is associative and commutative, with identity 0.28

- < is a linear ordering with least element 0.

- For any x, y, and z, if x < y then x + z < y + z

Additional Metric

1. ab = 0 if and only if a = b.
2. ab ≥ 0
3. ab = ba.
4. a != b and a != c imply ∠abc = ∠cba.
5. 0 ≤ ∠abc and ∠abc ≤ right-angle + right-angle.
6. ∆aab = 0.
7. ∆abc ≥ 0.
8. ∆abc = ∆cab and ∆abc = ∆acb.
9. If ab = a′b′, bc = b′c′, ca = c′a′, ∠abc = ∠a′b′c′, ∠bca = ∠b′c′a′, and∠cab = ∠c′a′b′, then abc = ∆a′b′c′

Diagram-segment transfer

1. If b is between a and c, then ab + bc = ac.
2. If a is the center of α and β, b is on α, c is on β, and ab = ac, then α = β.
3. If a is the center of α and b is on α, then ac = ab if and only if c is on α.
4. If a is the center of α and b is on α, and ac < ab if and only if c is in α.

Diagram-angle transfer

1. Suppose a != b, a != c, a is on L, and b is on L. Then c is on L and a is not between b and c if and only if ∠bac = 0.

2. Suppose a is on L and M, b is on L, c is on M, a != q, a 6= c, d is not on L or M, and L != M. Then ∠bac = ∠bad + ∠dac if and only if b and d are on the same side of M and c and d are on the same side of L.

3. Suppose a and b are points on L, c is between a and b, and d is not on L. Then ∠acd = ∠dcb if and only if ∠acd is equal to right-angle.

4. Suppose a, b, and b′are on L, a, c, and c′are on M, b != a, b′ != a, c != a,c′ 6 , a is not between b and b′, and a is not between c and c′Then .∠bac = ∠b′ac′.

5. Suppose a and b are on L, b and c are onM, and c and d are on N. Suppose also that b != c, a and d are on the same side of N, and ∠abc + ∠bcd < right- ngle + right-angle. Then L and N intersect, and if e is on L and N, then e and a are on the same side of M.

Diagram-area transfer

1. If a and b are on L and a =6 b, then ∆abc = 0 if and only if c is on L.

2. If a, b, c are on L and distinct from one another, d is not on L, then c is between a and b if and only if ∆acd +∆dcb = ∆adb.

Appendix C: Construction Rules in the EPC

As explained in Section 3.4, the intersection rules defined in E were modified slightly to simplify the implementation. The intersection rules that the EPC "understands" are:

1. Let a be a point of intersection of L and M.

2. Let a be a point of intersection of C and L.

3. Let a and b be the two points of intersection of C and L.

4. Let a be a point of intersection of L and C between b and c.

5. Let a be a point of intersection of L and C extending the segment from c to b.

6. Let a be a point of intersection of C and D.

7. Let a and b be the two points of intersection of C and D.

8. Let a be a point of intersection of C and D on the same side of L as b where L is the line through their centers c and d respectively.

9. Let a be a point of intersection of C and D on the side of L opposite b where L is the line through their centers c and d respectively.

Appendix D: Inference Rules in the EPC

The following are the inference rules encoded in the EPC. Note that to simplify rules into disjunctive normal form, in some cases one rule in *E* resulted in multiple rules in the EPC (noted as "1a", "1b", etc.). For example, a rule in E could be defined as:

if a and b then c and d

…and so would then be translated into the EPC as two rules:

```
!a v !b v c

!a v !b v d
```

The 72 rules in the EPC are as follows:

```
Generalities1: equals(*p7*, *p8*) v !on(*p7*, *l1*) v !on(*p8*, *l1*) v
on(*p7*, *l2*) v !on(*p8*, *l2*) v equals(*l1*, *l2*)

Generalities2: !center(*p7*, *c1*) v !center(*p8*, *c1*) v equals(*p7*, *p8*)

Generalities3: !center(*p1*, *c1*) v in(*p1*, *c1*)

Generalities4: !in(*p7*, *c1*) v !on(*p7*, *c1*)

EqualsOn1: !equals(*p1*, *p2*) v !on(*p1*, *l1*) v on(*p2*, *l1*)

EqualsOn2: !equals(*l1*, *l2*) v !on(*p1*, *l1*) v on(*p1*, *l2*)

EqualsBetween1: !equals(*p1*, *p2*) v !between(*p1*, *p3*, *p4*) v
between(*p2*, *p3*, *p4*)

EqualsBetween1: !equals(*p3*, *p2*) v !between(*p1*, *p3*, *p4*) v
between(*p1*, *p2*, *p4*)

EqualsBetween1: !equals(*p4*, *p2*) v !between(*p1*, *p3*, *p4*) v
between(*p1*, *p3*, *p2*)

EqualsAssociativity: !equals(*p1*, *p2*) v equals(*p2*, *p1*)

Betweenness1a: !between(*p7*, *p8*, *p9*) v between(*p9*, *p8*, *p7*)

Betweenness1b: !between(*p7*, *p8*, *p9*) v !equals(*p7*, *p8*)

Betweenness1c: !between(*p7*, *p8*, *p9*) v !equals(*p7*, *p9*)

Betweenness1d: !between(*p7*, *p8*, *p9*) v !between(*p8*, *p7*, *p9*)

Betweenness2: !between(*p7*, *p8*, *p9*) v !on(*p7*, *l1*) v !on(*p8*, *l1*)
v on(*p9*, *l1*)

Betweenness3: !between(*p7*, *p8*, *p9*) v !on(*p7*, *l1*) v !on(*p9*, *l1*)
v on(*p8*, *l1*)
```

```
Betweenness 4: !between(*p7*, *p8*, *p9*) v !between(*p7*, *p10*, *p8*) v
between(*p7*, *p10*, *p9*)

Betweenness5: !between(*p7*, *p8*, *p9*) v !between(*p8*, *p9*, *p10*) v
between(*p7*, *p8*, *p10*)

Betweenness6: equals(*p7*, *p8*) v equals(*p8*, *p9*) v equals(*p7*, *p9*) v
!on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v between(*p7*, *p8*,
*p9*) v between(*p8*, *p7*, *p9*) v between(*p7*, *p9*, *p8*)

Betweenness7: !between(*p7*, *p8*, *p9*) v !between(*p7*, *p8*, *p10*) v
!between(*p9*, *p8*, *p10*)

SameSide1: on(*p7*, *l1*) v sameside(*p7*, *p7*, *l1*)

SameSide2: !sameside(*p7*, *p8*, *l1*) v sameside(*p8*, *p7*, *l1*)

SameSide3: !sameside(*p7*, *p8*, *l1*) v !on(*p7*, *l1*)

SameSide4: !sameside(*p7*, *p8*, *l1*) v !sameside(*p7*, *p9*, *l1*) v
sameside(*p8*, *p9*, *l1*)

SameSide5a: on(*p1*, *l1*) v on(*p2*, *l1*) v on(*p3*, *l1*) v sameside(*p1*,
*p2*, *l1*) v sameside(*p2*, *p3*, *l1*) v sameside(*p1*, *p3*, *l1*)

Pasch1: !between(*p7*, *p8*, *p9*) v !sameside(*p7*, *p9*, *l1*) v
sameside(*p7*, *p8*, *l1*)

Pasch2: !between(*p7*, *p8*, *p9*) v !on(*p7*, *l1*) v on(*p8*, *l1*) v
sameside(*p8*, *p9*, *l1*)

Pasch3: !between(*p7*, *p8*, *p9*) v !on(*p8*, *l1*) v !sameside(*p7*, *p9*,
*l1*)

Pasch4: !on(*p8*, *l1*) v !on(*p8*, *l2*) v equals(*l1*, *l2*) v !on(*p7*,
*l2*) v !on(*p9*, *l2*) v equals(*p7*, *p8*) v equals(*p8*, *p9*) v
equals(*p7*, *p9*) v sameside(*p7*, *p9*, *l1*) v between(*p7*, *p8*, *p9*)

TripleIncidence1: !on(*p7*, *l2*) v !on(*p7*, *l3*) v !on(*p7*, *l1*) v
!on(*p8*, *l1*) v !on(*p9*, *l2*) v !sameside(*p9*, *p10*, *l1*) v
!sameside(*p8*, *p9*, *l3*) v !on(*p10*, *l3*) v !sameside(*p8*, *p10*, *l2*)

TripleIncidence2: !on(*p7*, *l1*) v !on(*p7*, *l2*) v !on(*p7*, *l3*) v
!on(*p8*, *l1*) v !on(*p9*, *l2*) v !on(*p10*, *l3*) v !sameside(*p9*, *p10*,
*l1*) v sameside(*p8*, *p10*, *l2*) v on(*p10*, *l2*) v equals(*p8*, *p7*) v
sameside(*p8*, *p9*, *l3*)

TripleIncidence3: !on(*p7*, *l1*) v !on(*p7*, *l2*) v !on(*p7*, *l3*) v
!on(*p8*, *l1*) v !on(*p9*, *l2*) v !on(*p10*, *l3*) v !sameside(*p9*, *p10*,
*l1*) v !sameside(*p8*, *p9*, *l3*) v !sameside(*p10*, *p11*, *l2*) v
!sameside(*p9*, *p11*, *l3*) v sameside(*p9*, *p11*, *l1*)
```

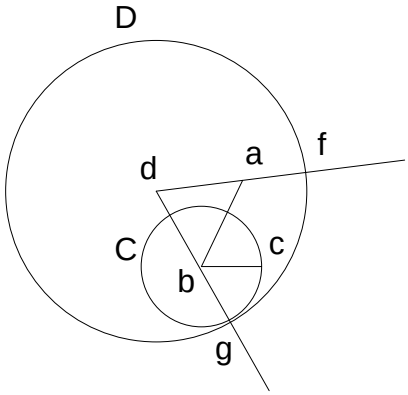
```
Circle1: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !in(*p7*,
*c1*) v !on(*p8*, *c1*) v !on(*p9*, *c1*) v equals(*p8*, *p9*) v
between(*p8*, *p7*, *p9*)

Circle2a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !on(*p8*,
*c1*) v !on(*p9*, *c1*) v !between(*p8*, *p7*, *p9*) v in(*p7*, *c1*)

Circle2b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !in(*p8*,
*c1*) v !on(*p9*, *c1*) v !between(*p8*, *p7*, *p9*) v in(*p7*, *c1*)

Circle2c: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !in(*p8*,
*c1*) v !in(*p9*, *c1*) v !between(*p8*, *p7*, *p9*) v in(*p7*, *c1*)

Circle2d: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !on(*p8*,
*c1*) v !in(*p9*, *c1*) v !between(*p8*, *p7*, *p9*) v in(*p7*, *c1*)

Circle3a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !in(*p7*,
*c1*) v in(*p9*, *c1*) v !between(*p7*, *p9*, *p8*) v !in(*p8*, *c1*)

Circle3b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !on(*p7*,
*c1*) v in(*p9*, *c1*) v !between(*p7*, *p9*, *p8*) v !in(*p8*, *c1*)

Circle3c: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !in(*p7*,
*c1*) v in(*p9*, *c1*) v !between(*p7*, *p9*, *p8*) v !on(*p8*, *c1*)

Circle3d: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v !on(*p7*,
*c1*) v in(*p9*, *c1*) v !between(*p7*, *p9*, *p8*) v !on(*p8*, *c1*)

Circle4: !on(*p9*, *c1*) v !on(*p9*, *c2*) v !on(*p10*, *c1*) v !on(*p10*,
*c2*) v equals(*p9*, *p10*) v !center(*p7*, *c1*) v !center(*p8*, *c2*) v
!on(*p7*, *l1*) v !on(*p8*, *l1*) v !sameside(*p9*, *p10*, *l1*)

Intersects0a: !intersects(*l1*, *l2*) v intersects(*l2*, *l1*)

Intersects0b: !intersects(*c1*, *c2*) v intersects(*c2*, *c1*)

Intersects1: on(*p7*, *l1*) v on(*p8*, *l1*) v sameside(*p7*, *p8*, *l1*) v
!on(*p7*, *l2*) v !on(*p8*, *l2*) v intersects(*l1*, *l2*)

Intersects2a: !on(*p7*, *c1*) v !on(*p8*, *c1*) v on(*p7*, *l1*) v on(*p8*,
*l1*) v sameside(*p7*, *p8*, *l1*) v intersects(*l1*, *c1*)

Intersects2a: !in(*p7*, *c1*) v !on(*p8*, *c1*) v on(*p7*, *l1*) v on(*p8*,
*l1*) v sameside(*p7*, *p8*, *l1*) v intersects(*l1*, *c1*)

Intersects2a: !on(*p7*, *c1*) v !in(*p8*, *c1*) v on(*p7*, *l1*) v on(*p8*,
*l1*) v sameside(*p7*, *p8*, *l1*) v intersects(*l1*, *c1*)

Intersects2a: !in(*p7*, *c1*) v !in(*p8*, *c1*) v on(*p7*, *l1*) v on(*p8*,
*l1*) v sameside(*p7*, *p8*, *l1*) v intersects(*l1*, *c1*)

Intersects3: !in(*p7*, *c1*) v !on(*p7*, *l1*) v intersects(*l1*, *c1*)

Intersects5: !on(*p7*, *c1*) v !in(*p8*, *c1*) v !in(*p7*, *c2*) v !on(*p8*,
*c2*) v intersects(*c1*, *c2*)
```

```
DiagramSegmentTransfer1: !between(*p7*, *p8*, *p9*) v segment-plus-segment-
equals-segment(*p7*, *p8*, *p8*, *p9*, *p7*, *p9*)

DiagramSegmentTransfer2: !center(*p7*, *c1*) v !center(*p7*, *c2*) v
!on(*p8*, *c1*) v !on(*p9*, *c2*) v !segments-equal(*p7*, *p8*, *p7*, *p9*) v
equals(*c1*, *c2*)

DiagramSegmentTransfer3a: !center(*p7*, *c1*) v !on(*p8*, *c1*) v !on(*p9*,
*c1*) v segments-equal(*p7*, *p9*, *p7*, *p8*)

DiagramSegmentTransfer3b: !center(*p7*, *c1*) v !on(*p8*, *c1*) v !segments-
equal(*p7*, *p9*, *p7*, *p8*) v on(*p9*, *c1*)

DiagramSegmentTransfer4a: !center(*p7*, *c1*) v !on(*p8*, *c1*) v !in(*p9*,
*c1*) v segment-lessthan(*p7*, *p9*, *p7*, *p8*)

DiagramSegmentTransfer4b: !center(*p7*, *c1*) v !on(*p8*, *c1*) v !segment-
lessthan(*p7*, *p9*, *p7*, *p8*) v in(*p9*, *c1*)

DiagramAngleTransfer1a: equals(*p7*, *p8*) v equals(*p7*, *p9*) v !on(*p7*,
*l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v between(*p8*, *p7*, *p9*) v
angle-equals-0(*p8*, *p7*, *p9*)

DiagramAngleTransfer1b: equals(*p7*, *p8*) v equals(*p7*, *p9*) v !on(*p7*,
*l1*) v !on(*p8*, *l1*) v !angle-equals-0(*p8*, *p7*, *p9*) v on(*p9*, *l1*)

DiagramAngleTransfer1c: equals(*p7*, *p8*) v equals(*p7*, *p9*) v !on(*p7*,
*l1*) v !on(*p8*, *l1*) v !angle-equals-0(*p8*, *p7*, *p9*) v !between(*p8*,
*p7*, *p9*)

DiagramAngleTransfer2: !on(*p7*, *l1*) v !on(*p7*, *l2*) v !on(*p8*, *l1*) v
!on(*p9*, *l2*) v equals(*p7*, *p8*) v equals(*p7*, *p9*) v on(*p10*, *l2*) v
on(*p10*, *l1*) v equals(*l1*, *l2*) v !sameside(*p8*, *p10*, *l2*) v
!sameside(*p9*, *p10*, *l1*) v angle-plus-angle-equals-angle(*p8*, *p7*,
*p10*, *p10*, *p7*, *p9*, *p8*, *p7*, *p9*)

DiagramAngleTransfer2b: !on(*p7*, *l1*) v !on(*p7*, *l2*) v !on(*p8*, *l1*) v
!on(*p9*, *l2*) v equals(*p7*, *p8*) v equals(*p7*, *p9*) v on(*p10*, *l2*) v
on(*p10*, *l1*) v equals(*l1*, *l2*) v !angle-plus-angle-equals-angle(*p8*,
*p7*, *p10*, *p10*, *p7*, *p9*, *p8*, *p7*, *p9*) v sameside(*p8*, *p10*,
*l2*)

DiagramAngleTransfer2c: !on(*p7*, *l1*) v !on(*p7*, *l2*) v !on(*p8*, *l1*) v
!on(*p9*, *l2*) v equals(*p7*, *p8*) v equals(*p7*, *p9*) v on(*p10*, *l2*) v
on(*p10*, *l1*) v equals(*l1*, *l2*) v !angle-plus-angle-equals-angle(*p8*,
*p7*, *p10*, *p10*, *p7*, *p9*, *p8*, *p7*, *p9*) v sameside(*p9*, *p10*,
*l1*)

DiagramAngleTransfer3a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !between(*p7*,
*p9*, *p8*) v on(*p10*, *l1*) v !angles-equal(*p7*, *p9*, *p10*, *p10*, *p9*,
*p8*) v angle-equals-90(*p7*, *p9*, *p10*)
```

```
DiagramAngleTransfer3b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !between(*p7*,
*p9*, *p8*) v on(*p10*, *l1*) v !angle-equals-90(*p7*, *p9*, *p10*) v angles-
equal(*p7*, *p9*, *p10*, *p10*, *p9*, *p8*)

DiagramAngleTransfer4: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p10*, *l1*) v
!on(*p7*, *l2*) v !on(*p9*, *l2*) v !on(*p11*, *l2*) v equals(*p7*, *p8*) v
equals(*p7*, *p9*) v equals(*p7*, *p10*) v equals(*p7*, *p11*) v
between(*p8*, *p7*, *p10*) v between(*p9*, *p7*, *p11*) v angles-equal(*p8*,
*p7*, *p9*, *p10*, *p7*, *p11*)

DiagramAngleTransfer5a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p8*, *l2*) v
!on(*p9*, *l2*) v !on(*p9*, *l3*) v !on(*p10*, *l3*) v equals(*p8*, *p9*) v
!sameside(*p7*, *p10*, *l2*) v !angle-plus-angle-less-than-180(*p7*, *p8*,
*p9*, *p8*, *p9*, *p10*) v intersects(*l1*, *l3*)

DiagramAngleTransfer5b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p8*, *l2*) v
!on(*p9*, *l2*) v !on(*p9*, *l3*) v !on(*p10*, *l3*) v equals(*p8*, *p9*) v
!sameside(*p7*, *p10*, *l2*) v !angle-plus-angle-less-than-180(*p7*, *p8*,
*p9*, *p8*, *p9*, *p10*) v !on(*p11*, *l1*) v !on(*p11*, *l3*) v
sameside(*p11*, *p7*, *l2*)

DiagramAreaTransfer1a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v equals(*p7*, *p8*)
v !area-equals-0(*p7*, *p8*, *p9*) v on(*p9*, *l1*)

DiagramAreaTransfer1b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v equals(*p7*, *p8*)
v !on(*p9*, *l1*) v area-equals-0(*p7*, *p8*, *p9*)

DiagramAreaTransfer2a: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v
equals(*p7*, *p8*) v equals(*p7*, *p9*) v equals(*p8*, *p9*) v on(*p10*,
*l1*) v !between(*p7*, *p9*, *p8*) v area-plus-area-equals-area(*p7*, *p9*,
*p10*, *p10*, *p9*, *p8*, *p7*, *p10*, *p8*)

DiagramAreaTransfer2b: !on(*p7*, *l1*) v !on(*p8*, *l1*) v !on(*p9*, *l1*) v
equals(*p7*, *p8*) v equals(*p7*, *p9*) v equals(*p8*, *p9*) v on(*p10*,
*l1*) v !area-plus-area-equals-area(*p7*, *p9*, *p10*, *p10*, *p9*, *p8*,
*p7*, *p10*, *p8*) v between(*p7*, *p9*, *p8*)
```

Appendix E: Diagrammatic Inferences

Given the proof of Proposition I.2 of the Elements, described in Section 4.1:



…the following inferences were generated by the EPC:

```
angle-equals-0(a, d, f)
angle-equals-0(a, f, d)
angle-equals-0(b, d, g)
angle-equals-0(b, g, d)
angle-equals-0(d, f, a)
angle-equals-0(d, g, b)
angle-equals-0(f, d, a)
angle-equals-0(g, d, b)
!angle-equals-0(d, a, f)
!angle-equals-0(d, b, g)
!angle-equals-0(f, a, d)
!angle-equals-0(g, b, d)
angles-equal(a, d, b, f, d, g)
angles-equal(a, d, g, f, d, b)
angles-equal(b, d, a, g, d, f)
angles-equal(b, d, f, g, d, a)
angles-equal(f, d, b, a, d, g)
angles-equal(f, d, g, a, d, b)
angles-equal(g, d, a, b, d, f)
angles-equal(g, d, f, b, d, a)
area-equals-0(a, d, f)
area-equals-0(a, f, d)
area-equals-0(b, d, g)
area-equals-0(b, g, d)
area-equals-0(d, a, f)
area-equals-0(d, b, g)
area-equals-0(d, f, a)
area-equals-0(d, g, b)
area-equals-0(f, a, d)
area-equals-0(f, d, a)
area-equals-0(g, b, d)
area-equals-0(g, d, b)
between(d, a, f)
between(d, b, g)
between(f, a, d)
on(a, F)
on(a, M)
on(b, E)
on(b, N)
on(c, C)
on(d, E)
on(d, F)
on(d, M)
on(d, N)
on(f, D)
on(f, M)
on(g, C)
on(g, D)
on(g, N)
!on(a, D)
!on(a, E)
!on(b, C)
!on(b, D)
!on(b, F)
!on(d, D)
!on(f, F)
!on(g, E)
!sameside(a, b, M)
!sameside(a, b, N)
!sameside(a, c, M)
!sameside(a, d, M)
!sameside(a, d, N)
!sameside(a, f, M)
!sameside(a, g, N)
!sameside(b, a, M)
!sameside(b, a, N)
!sameside(b, c, N)
!sameside(b, d, M)
!sameside(b, d, N)
!sameside(b, f, M)
```

```
between(g, b, d)
!between(a, d, f)
!between(a, f, d)
!between(b, d, g)
!between(b, g, d)
!between(d, f, a)
!between(d, g, b)
!between(f, d, a)
!between(g, d, b)
center(a, E)
center(b, C)
center(b, F)
center(d, D)
!center(a, D)
!center(a, F)
!center(b, D)
!center(b, E)
!center(c, C)
!center(d, C)
!center(d, E)
!center(d, F)
!center(f, D)
!center(f, E)
!center(f, F)
!center(g, C)
!center(g, D)
!center(g, E)
!center(g, F)
!equals(a, d)
!equals(a, f)
!equals(b, d)
!equals(b, g)
!equals(d, a)
!equals(d, b)
!equals(d, f)
!equals(d, g)
!equals(f, a)
!equals(f, d)
!equals(g, b)
!equals(g, d)
in(a, D)
in(a, E)
in(b, C)
in(b, D)
in(b, F)
in(d, D)
!in(a, F)
!in(b, E)
!in(c, C)
!in(d, E)
!in(d, F)
!in(f, D)
!in(f, F)
!in(g, C)
!in(g, D)
!in(g, E)
intersects(E, F)
intersects(F, E)
intersects(M, D)
intersects(M, E)
intersects(N, C)
!sameside(b, g, N)
!sameside(c, a, M)
!sameside(c, b, N)
!sameside(c, d, M)
!sameside(c, d, N)
!sameside(c, f, M)
!sameside(c, g, N)
!sameside(d, a, M)
!sameside(d, a, N)
!sameside(d, b, M)
!sameside(d, b, N)
!sameside(d, c, M)
!sameside(d, c, N)
!sameside(d, f, M)
!sameside(d, f, N)
!sameside(d, g, M)
!sameside(d, g, N)
!sameside(f, a, M)
!sameside(f, b, M)
!sameside(f, c, M)
!sameside(f, d, M)
!sameside(f, d, N)
!sameside(f, g, M)
!sameside(g, a, N)
!sameside(g, b, N)
!sameside(g, c, N)
!sameside(g, d, M)
!sameside(g, d, N)
!sameside(g, f, M)
segment-lessthan(d, a, d, f)
segment-lessthan(d, a, d, g)
segment-lessthan(d, b, d, f)
segment-lessthan(d, b, d, g)
!segment-lessthan(a, b, a, d)
!segment-lessthan(a, d, a, b)
!segment-lessthan(a, g, a, b)
!segment-lessthan(a, g, a, d)
!segment-lessthan(b, a, b, d)
!segment-lessthan(b, c, b, g)
!segment-lessthan(b, d, b, a)
!segment-lessthan(b, f, b, a)
!segment-lessthan(b, f, b, d)
!segment-lessthan(b, g, b, c)
!segment-lessthan(d, f, d, g)
!segment-lessthan(d, g, d, f)
segment-plus-segment-equals-
segment(d, a, a, f, d, f)
segment-plus-segment-equals-
segment(d, b, b, g, d, g)
segment-plus-segment-equals-
segment(f, a, a, d, f, d)
segment-plus-segment-equals-
segment(g, b, b, d, g, d)
segments-equal(a, b, a, d)
segments-equal(a, d, a, b)
segments-equal(b, a, b, d)
segments-equal(b, c, b, g)
segments-equal(b, d, b, a)
segments-equal(b, g, b, c)
segments-equal(d, f, d, g)
segments-equal(d, g, d, f)
```

```
intersects(N, D)
intersects(N, F)
```

```
!segments-equal(a, g, a, b)
!segments-equal(a, g, a, d)
!segments-equal(b, f, b, a)
!segments-equal(b, f, b, d)
!segments-equal(d, a, d, f)
!segments-equal(d, a, d, g)
!segments-equal(d, b, d, f)
!segments-equal(d, b, d, g)
```

References

1. Jeremy Avigad, Edward Dean, John Mumma. *A formal system for Euclid's elements*. The Review of Symbolic Logic, 2009.

2. David Hilbert. Grundlagen der Geometrie. *In Festschrift zur Feier der Enthüllung des Gauss-Weber Denkmals in G¨ottingen*. Teubner, Leipzig, 1899. Translated by Leo Unger as Foundations of Geometry, Open Court, La Salle, 1971. Ninth printing, 1997.

3. Euclid. *The Thirteen Books of the Elements, volumes I–III*. Dover Publications, New York, second edition, 1956. Translated with introduction and commentary by Sir Thomas L. Heath, from the text of Heiberg. The Heath translation has also been issued as Euclid's Elements: All Thirteen Books Complete in One Volume, Green Lion Press, Santa Fe, 2002.

4. Robin Hartshorne. *Geometry: Euclid and beyond*. Springer, New York, 2005.

5. Philip C. Jackson Jr. *Introduction to Artificial Intelligence: Second, Enlarged Edition*. Dover Publications, 2 Enl Sub Edition, 1985.

6. Kenneth Manders. *Diagram-based geometric practice*. In Paolo Mancosu, editor, The Philosophy of Mathematical Practice, pages 65–79. Oxford University Press, Oxford, 2008.

7. Kenneth Manders. *The Euclidean diagram*. In Paolo Mancosu, editor, The Philosophy of Mathematical Practice, pages 80–133. Oxford University Press, Oxford, 2008. MS first circulated in 1995.

8. Nathaniel Miller. *Euclid and his Twentieth Century Rivals: Diagrams in the Logic of Euclidean Geometry*. CSLI, Stanford, 2008. Based on Miller's 2001 PhD thesis, "A diagrammatic formal system for Euclidean geometry," Cornell University.

9. John Mumma. *Proofs, pictures, and Euclid*. Synthese, 2009.

10. John Mumma. *Intuition Formalized: Ancient and Modern Methods of Proof in Elementary Geometry*. PhD thesis, Carnegie Mellon University, 2006.

11. John Mumma. *Review of Euclid and his Twentieth Century Rivals, by Nathaniel Miller*. Philosophia Mathematica, 16:256–264, 2008.

12. Moritz Pasch. *Vorlesungen über neuere Geometrie*. Teubner, Leipzig, 1882.

13. Alfred Tarski. *What is elementary geometry?* In Leon Henkin, Patrick Suppes, and Alfred Tarski, editors, The Axiomatic Method: with Special Reference to Geometry and Physics, pages 16–29. North-Holland, 1st edition, 1959.

14. Alfred Tarski and Steven Givant. *Tarski's system of geometry*. Bulletin of Symbolic Logic, 5:175–214, 1999.