Autoformalizing Euclidean Geometry

Logan Murphy [1*] **Kaiyu Yang** [2*] **Jialiang Sun** [1] **Zhaoyu Li** [1] **Anima Anandkumar** [2] **Xujie Si** [1]

Abstract

Autoformalization involves automatically translating informal math into formal theorems and proofs that are machine-verifiable. Euclidean geometry provides an interesting and controllable domain for studying autoformalization. In this paper, we introduce a neuro-symbolic framework for autoformalizing Euclidean geometry, which combines domain knowledge, SMT solvers, and large language models (LLMs). One challenge in Euclidean geometry is that informal proofs rely on diagrams, leaving gaps in texts that are hard to formalize. To address this issue, we use theorem provers to fill in such diagrammatic information automatically, so that the LLM only needs to autoformalize the explicit textual steps, making it easier for the model. We also provide automatic semantic evaluation for autoformalized theorem statements. We construct LeanEuclid, an autoformalization benchmark consisting of problems from Euclid's *Elements* and the UniGeo dataset formalized in the Lean proof assistant. Experiments with GPT-4 and GPT-4V show the capability and limitations of state-of-the-art LLMs on autoformalizing geometry problems. The data and code are available at `https://github.com/loganrjmurphy/LeanEuclid`.

1. Introduction

Euclidean geometry is one of the oldest branches of mathematics. It has served as a test of human intelligence for more than two millennia and has recently been used to test AI. Substantial work has focused on solving geometry problems automatically (Wu, 2008), e.g., AlphaGeometry (Trinh et al., 2023) can solve some of the IMO geometry problems. These methods consume problems and produce solutions in structured formats. In this work, we address a complementary task, *autoformalization*: Can AI understand human-written problems/solutions and translate them automatically into formal theorems/proofs? Specifically, we focus on the setting where formal theorems/proofs can be verified by the Lean proof assistant (de Moura & Ullrich, 2021). Lean provides a language for writing formal proofs. It is popular among mathematicians and has a growing ecosystem of integration with large language models (LLMs), e.g., LeanDojo (Yang et al., 2023) and Lean Copilot (Song et al., 2024).

We demonstrate that Euclidean geometry provides an interesting and controllable domain for autoformalization. First, an automatic evaluation of autoformalized theorems is difficult in general but feasible in Euclidean geometry. Second, the logical gaps in informal proofs are well understood in Euclidean geometry, making it easier to faithfully formalize the proofs. Third, combining text-based and diagrammatic reasoning makes Euclidean geometry a natural domain to study multimodal reasoning models. Therefore, autoformalizing Euclidean geometry is an attractive target for AI.

Evaluating Autoformalized Theorem Statements. Despite the promise of machine learning and LLMs in autoformalizing theorems (Wu et al., 2022), a major roadblock is the lack of reliable and automatic evaluation. Comparing the model output verbatim with ground truth would be too rigid since there are many valid ways to formalize a theorem. Checking the logical equivalence between two theorems is generally intractable. Researchers have resorted to proxy metrics such as the BLEU score (Papineni et al., 2002). However, LLMs can score high on such metrics without generating correct formalization (Jiang et al., 2023b). Alternatively, human evaluation is widely used as a last resort, but it is costly, especially if we want to use the results to improve the method iteratively.

Our Approach to Evaluating Autoformalization. To overcome the evaluation bottleneck, we introduce a new automatic approach for evaluating the semantics of autoformalized theorems. The key insight is that equivalence checking can be made feasible in specific domains (such as Euclidean geometry) by combining domain knowledge with automated reasoning tools, such as satisfiability modulo theories (SMT) solvers (Barrett & Tinelli, 2018).

To evaluate the autoformalized theorems, we develop a sym-

[*]Equal contribution [1]University of Toronto [2]Caltech. Correspondence to: Logan Murphy <lmurphy@cs.toronto.edu>, Kaiyu Yang <kaiyuy@caltech.edu>, Xujie Si <six@cs.toronto.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

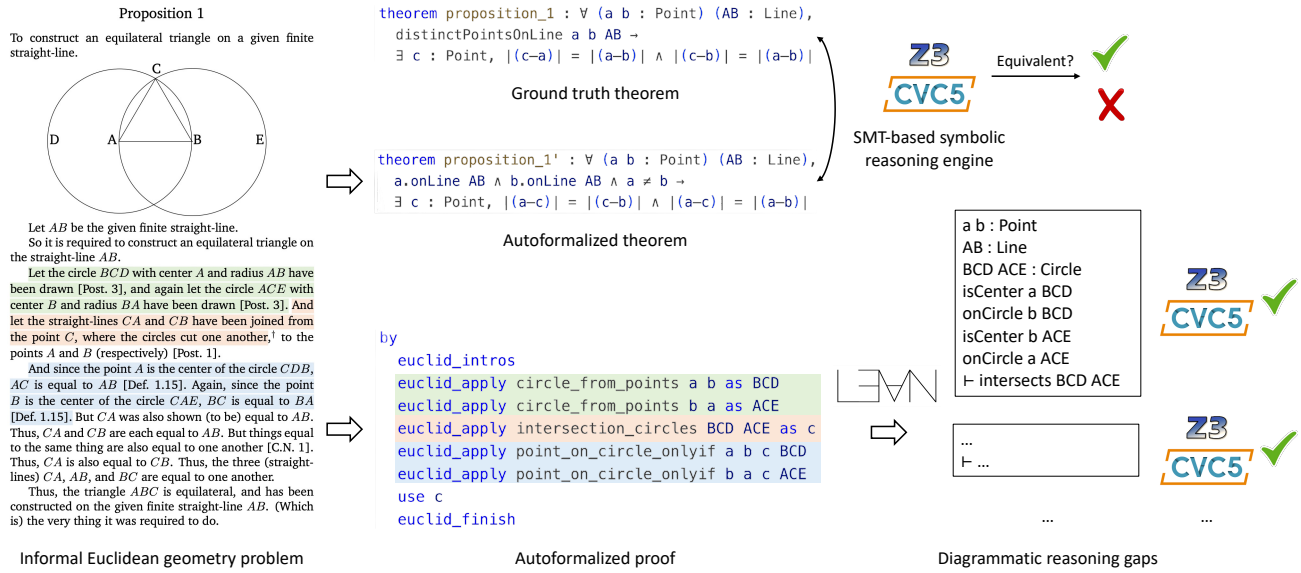


Figure 1. **Left**: Proposition 1 in Euclid's *Elements* (Book I). The orange text involves diagrammatic reasoning: Euclid did not explicitly prove the two circles actually intersect, but the reader can use the diagram to implicitly fill in the logical gap. **Top right**: The model autoformalizes the problem into a formal theorem (`proposition_1'`), which is evaluated by checking its logical equivalence with the ground truth (`proposition_1`), leveraging domain knowledge and a symbolic automated reasoning engine based on SMT (satisfiability modulo theories) solvers. **Bottom right**: A proof autoformalized by the model. Like Euclid's proofs, it does not need to handle diagrammatic reasoning explicitly. Lean can check the proof to identify a list of diagrammatic reasoning gaps, e.g., "`intersects BCD ACE`". Then, it attempts to fill in all gaps automatically using the symbolic reasoning engine based on SMT solvers.

bolic reasoning engine based on SMT solvers. As Fig. 1 (*Top right*) shows, given a ground-truth formal theorem T_{gt} and the autoformalized theorem T_{pred} produced by a language model, we use the symbolic engine to try to prove their equivalence ($T_{gt} \Leftrightarrow T_{pred}$). If successful, their logical gap is small enough to conclude that T_{pred} is correct. Even if the symbolic engine cannot prove $T_{gt} \Leftrightarrow T_{pred}$, it can provide partial results useful for a more fine-grained analysis. We validate this evaluation protocol by showing it correlates well with human evaluation.

LeanEuclid: Formalizing Proofs and Diagrams. We construct *LeanEuclid*, a benchmark for testing machine learning on autoformalizing Euclidean geometry. As in Fig 1 (*Left*), each example in LeanEuclid has an informal theorem, proof, and diagram in LaTeX, as well as a formal theorem and proof in Lean. Data examples in LeanEuclid are manually formalized into Lean from Euclid's *Elements* (Heiberg, 2007) and the UniGeo dataset (Chen et al., 2022).

LeanEuclid serves as a benchmark for autoformalizing not only theorems but also proofs. Geometric proofs are challenging to formalize faithfully. Humans (ancient or modern, including Euclid himself) use diagrams to license proof steps without making every detail explicit. Fig. 1 shows an example of diagrammatic reasoning from Euclid's *Elements*. Euclid uses the intersection of two circles (C) without proving its existence. Most readers would not find the proof problematic, as the two circles intersect in the diagram. Such implicit diagrammatic reasoning is ubiquitous in informal geometric proofs but needs to be handled explicitly in formal proofs (Beeson et al., 2019). Therefore, a naive attempt to autoformalize the proofs would be difficult, as it requires the model to fill in many diagrammatic reasoning gaps, with nothing to reference in the informal texts.

To mitigate diagrammatic gaps, LeanEuclid adopts a formal system named E (Avigad et al., 2009), introduced by philosophers for modeling diagrammatic reasoning in Euclid's *Elements*. It teases out a set of diagrammatic rules so that diagrammatic reasoning can be modeled as logical deductions. We implement E in Lean and provide proof automation to fill in diagrammatic reasoning gaps, using the same symbolic reasoning engine developed for equivalence checking. Our system enables formalizing all 48 theorems and proofs from *Elements* (Book I), following Euclid's original proofs as closely as possible, with diagrammatic reasoning carried out implicitly and automatically (see Fig. 1). The data is included in LeanEuclid, making autoformalizing Euclid's proofs feasible. The language model now only needs to autoformalize the explicit textual proof steps, leaving the "obvious" implicit reasoning to the symbolic engine.

LLMs for Autoformalization. Using LeanEuclid, we experimentally investigate the capability of GPT-4 (OpenAI, 2023) and GPT-4V (with vision) on autoformalizing theorems and proofs in Euclidean geometry. With 5-shot prompting, GPT-4 can autoformalize only 18.9% of the theorems correctly, demonstrating the difficulty of LeanEuclid. GPT-4V performs slightly better (21.0%), showing the potential benefit of multimodal models. Furthermore, GPT-4V can autoformalize 23.1% of LeanEuclid proofs correctly, generating the entire proof without searching the space of individual steps. Finally, our study shows that imperfect autoformalized proofs can often be manually repaired into correct proofs through a few simple modifications.

Contributions. We make three main contributions: First, we introduce an SMT-based symbolic engine for automatically evaluating the *semantics* of theorem statements in Euclidean geometry. Second, we demonstrate the feasibility of autoformalizing only explicit geometric proof steps, relying on the symbolic engine for implicit diagrammatic reasoning. Third, our experiments reveal the capability and limitations of state-of-the-art LLMs on autoformalizing Euclidean geometry. Overall, our approach demonstrates Euclidean geometry as a promising target for autoformalization. It may facilitate autoformalization research to move from human inspection to automatic evaluation. Such a transition is essential for accelerating the development of machine learning methods on this task and ultimately easing the human effort required by formalization.

2. Related Work

Neural Theorem Proving and Autoformalization. Machine learning and LLMs can prove theorems using proof assistants (Polu & Sutskever, 2020; Yang & Deng, 2019; Yang et al., 2023; First et al., 2023; Lample et al., 2022; Han et al., 2022). In theorem proving, everything is formal: The model generates a formal proof given a formal theorem. In contrast, we focus on autoformalization, which aims to translate informal math into formal theorems and proofs. Li et al. (2024) provides a comprehensive survey on deep learning for theorem proving and autoformalization.

Machine learning for autoformalization is strained by the lack of informal-formal pairs for training. Manually annotating such a dataset would be prohibitively expensive. Researchers have explored two approaches to overcome data scarcity. First, we can generate synthetic training data either procedurally (Wang et al., 2018; Cunningham et al., 2023) or using LLMs (Chen et al., 2023; Jiang et al., 2023b). Second, we can leverage the few-shot capability of LLMs such as GPT-4 (OpenAI, 2023). We only need to annotate a few examples for in-context demonstration and a relatively small testing set (Wu et al., 2022; Agrawal et al., 2022; Jiang et al., 2023a; Azerbayev et al., 2023). LeanEuclid follows the latter approach and is intended as a benchmark for testing only.

Autoformalized theorems are difficult to evaluate, so all existing works have resorted to human evaluation. In addition, they have used proxy metrics such as compilation rates (Jiang et al., 2023b; Azerbayev et al., 2023) and the BLEU score (Wu et al., 2022; Azerbayev et al., 2023), none of which is sufficiently accurate. In contrast, we are the first to evaluate the semantics of autoformalized statements reliably and automatically.

Formalizing Euclidean Geometry. To our knowledge, no machine learning method has attempted to autoformalize Euclidean geometry. Tarski's (Schwabhäuser et al., 2013) and Hilbert's axioms (Hilbert, 2013) are two classical axiomatic systems that put Euclidean geometry on a rigorous foundation. Beeson et al. (2019) uses a variant of Tarski's system to manually formalize Euclid's *Elements* (Book I) in the Coq proof assistant (Barras et al., 1997). Hernandez-Espiet (2023) is an ongoing work formalizing the book in Lean. However, they require formal proofs to explicitly handle all gaps left by diagrammatic reasoning. As a result, formal proofs in their systems are much more verbose than Euclid's proofs (examples in Appendix C), making their systems unsuitable for studying autoformalization.

Proofs in Euclidean geometry have gaps due to diagrammatic reasoning. However, for more than two millennia, they have seldom troubled *Elements*' readers around the world. Philosophers have posited many theories aiming at a systematic understanding of diagrammatic reasoning (Miller, 2001; Mumma, 2010). In particular, we build upon the formal system E (Avigad et al., 2009) and are the first to turn this framework into a practical system that includes an automated procedure for diagrammatic reasoning.

Geometric Problem Solving. Automatic geometry problem solvers have been studied extensively (Nevins, 1975; Zhang et al., 1995). Wu's method (Wu, 2008) and Gröbner bases method (Buchberger, 2005) transform geometric problems into algebraic equations. AlphaGeometry (Trinh et al., 2023) is a concurrent work that successfully solves many IMO geometry problems. Similar to us, it is a hybrid system using neural networks for difficult proof steps (e.g., constructing new geometric objects) and a symbolic engine for deducing simple facts. However, our method focuses on autoformalizing human-written theorems/proofs instead of finding new proofs. AlphaGeometry consumes problems and produces solutions in structured representations, without handling informal mathematical language. Furthermore, they represent theorems and proofs in a domain-specific language tailored for Euclidean geometry, whereas we use Lean: a general-purpose proof assistant. Lean provides a

highly trusted foundation for the soundness of our system,[1] and it opens up opportunities to interoperate with other branches of mathematics formalized in Lean's mathematical library (The mathlib Community, 2020).

Euclidean geometry has also served as benchmarks in machine learning and NLP for multimodal understanding of geometric problems and diagrams (Lu et al., 2021; Chen et al., 2021a; Gao et al., 2023; Peng et al., 2023; Kazemi et al., 2023; Liang et al., 2023). These methods produce informal solutions given informal problem statements, whereas we formalize both problems and solutions. UniGeo (Chen et al., 2022) is a dataset consisting of 9,543 informal theorems and proofs. We formalize 125 random examples from UniGeo and include them in LeanEuclid, which shows the generality of our formalization beyond Euclid's *Elements*.

3. Formalizing Euclidean Geometry in Lean

To construct the LeanEuclid benchmark, we implement a variant of system E (Avigad et al., 2009) in Lean and use it to formalize Euclidean geometry problems. LeanEuclid has two key features: First, it supports implicit diagrammatic reasoning, which is critical to faithfully formalizing geometric proofs. Second, it can check the logical equivalence between theorems, which is essential for automatically evaluating the autoformalized theorems.

3.1. The Lean Proof Assistant

Proof assistants, a.k.a. interactive theorem provers, are languages and IDEs for humans to write formal proofs that can be checked by computers (Leroy et al., 2016; Hales et al., 2017). Widely used proof assistants include Coq, Isabelle, and Lean. They are similar from a machine learning perspective. Fig. 1 (*Middle*) shows examples of theorems and proofs in Lean. Each theorem has a name and a theorem statement. A proof consists of multiple steps, and each step is called a "tactic". For example, "`use c`" and "`euclid_apply circle_from_points b a as ACE`" are tactics.

Given a theorem and its proof, Lean can check the proof and output an error message if it is incorrect. Lean can also check if the theorem is well-formed. The percentage of well-formed theorems autoformalized by the model is widely used as a proxy metric in prior works (Jiang et al., 2023b; Azerbayev et al., 2023). However, well-formedness does not guarantee correctness, since a predicted theorem statement can be well-formed but have a different meaning from the ground truth.

[1]Our symbolic reasoning engine (Sec. 3.3) relies on SMT solvers, whose results currently cannot be fully certified by Lean.

3.2. The Formal System E

E is a formal system for modeling theorems and proofs in Euclidean geometry inspired by Euclid's *Elements*. It defines basic geometric objects (e.g., points and lines) and relations between them (e.g., a point being on a line). It also prescribes how to formally state and prove theorems. E is a conceptual framework that can be implemented within any proof assistant. We explain our Lean implementation using `proposition_1` in Fig. 1 as a running example. For a complete specification of E, please refer to Appendix A and E's paper (Avigad et al., 2009).

Geometric Objects, Relations, and Theorems. E has six types of objects: points, lines, segments, circles, angles, and triangles. They can form various relations, e.g., "`onLine a AB`" means the point `a` is on the line `AB`. Some relations are defined as syntactic sugar of more basic relations, e.g., "`distinctPointsOnLine a b AB`" is the conjunction of "`onLine a AB`", "`onLine b AB`", and "`a` $\neq$ `b`". A segment `(a--b)` is specified by its two endpoints `a` and `b`, and it has a length `|(a--b)|` $\in \mathbb{R}$. Using these concepts, it is straightforward to understand the theorem statement of `proposition_1`:

```
theorem proposition_1 : ∀ (a b : Point) (AB
    : Line), distinctPointsOnLine a b AB
   → ∃ c : Point, |(c--a)| = |(a--b)| ∧
   |(c--b)| = |(a--b)|
```

Like above, most theorems in Euclidean geometry take the form of *given existing objects satisfying certain properties, constructing new objects satisfying certain properties*. Below is the general form of theorems in E:

$$\forall\, (\overrightarrow{P} : Point)\ (\overrightarrow{L} : Line)\ (\overrightarrow{C} : Circle),\ \bigwedge_{i=1}^{m} a_i \rightarrow$$
$$\exists\, (\overrightarrow{P'} : Point)\ (\overrightarrow{L'} : Line)\ (\overrightarrow{C'} : Circle),\ \bigwedge_{j=1}^{n} a'_j$$

That is, given points $\overrightarrow{P}$, lines $\overrightarrow{L}$, and circles $\overrightarrow{C}$ satisfying $\bigwedge_{i=1}^{m} a_i$, one can construct points $\overrightarrow{P'}$, lines $\overrightarrow{L'}$, and circles $\overrightarrow{C'}$ satisfying $\bigwedge_{j=1}^{n} a'_j$. Each clause a_i is called a "precondition". Logically, it is a disjunction of literals with variables from $\overrightarrow{P}$, $\overrightarrow{L}$, and $\overrightarrow{C}$. Each a'_j is called a "postcondition", which is a disjunction of literals with variables from $\overrightarrow{P}$, $\overrightarrow{L}$, $\overrightarrow{C}$, $\overrightarrow{P'}$, $\overrightarrow{L'}$, and $\overrightarrow{C'}$. Variable sets can be empty. For example, in `proposition_1`, we have $\overrightarrow{C} = \overrightarrow{L'} = \overrightarrow{C'} = \varnothing$.

Axioms. Axioms take the same form as theorems, but they are built into the system and do not need to be proved. Appendix A contains the complete list of axioms in our instantiation of the formal system E. They fall into two

categories: construction rules and non-construction rules.[2] Construction rules construct new objects, i.e., at least one of $\overrightarrow{P'}$, $\overrightarrow{L'}$, and $\overrightarrow{C'}$ must be non-empty. For example, `circle_from_points` constructs a circle α given a point `a` as its center and another point `b` on it:

```
axiom circle_from_points : ∀ (a b : Point),
     a ≠ b → ∃ α : Circle, (isCenter a α)
    ∧ (onCircle b α)
```

Given two intersecting circles α and β, `intersection_circles` constructs their intersection:

```
axiom intersection_circles : ∀ (α β :
    Circle), intersectsCircle α β → ∃ c :
    Point, (onCircle c α) ∧ (onCircle c β)
```

Non-construction rules deduce properties of existing objects without creating new objects. For examples:

```
axiom center_inside_circle : ∀ (a : Point)
    (α : Circle), isCenter a α →
    insideCircle a α

axiom intersection_circle_circle_2: ∀ (a b
    : Point) (α β : Circle), (onCircle a α)
     ∧ (insideCircle b α) ∧ (insideCircle a
     β) ∧ (onCircle b β) →
    intersectsCircle α β
```

The distinction between construction and non-construction rules is important for automated reasoning in Sec. 3.3.

Proofs. We implement the formal system E in Lean. Therefore, a proof can be made of arbitrary tactics as long as the entire proof can be checked by Lean. Nevertheless, we provide several customized tactics that can cover most steps used by Euclid (complete list in Appendix A). The most important tactic we provide is `euclid_apply`, which applies a rule (either an axiom or an existing theorem) in the forward direction. Taking `intersection_circles` as an example, when using this rule, one should explicitly specify two circles, say, `BCD` and `ACE`:

```
intersection_circles BCD ACE :
    intersectsCircle BCD ACE → ∃ c : Point,
     (onCircle c BCD) ∧ (onCircle c ACE)
```

"`euclid_apply intersection_circles BCD ACE as c`" instructs Lean to check if the precondition "`intersectsCircle BCD ACE`" is a known fact. If so, it introduces a new point `c` and two new facts, "`onCircle c BCD`" and "`onCircle c ACE`". If not, it triggers diagrammatic reasoning, trying to prove "`intersectsCircle BCD ACE`" using a symbolic automated reasoning engine (Sec. 3.3). If successful, it proceeds as normal. Otherwise, the tactic fails. Fig. 2 shows three proof steps using `euclid_apply` with different rules. The last step triggers diagrammatic reasoning.

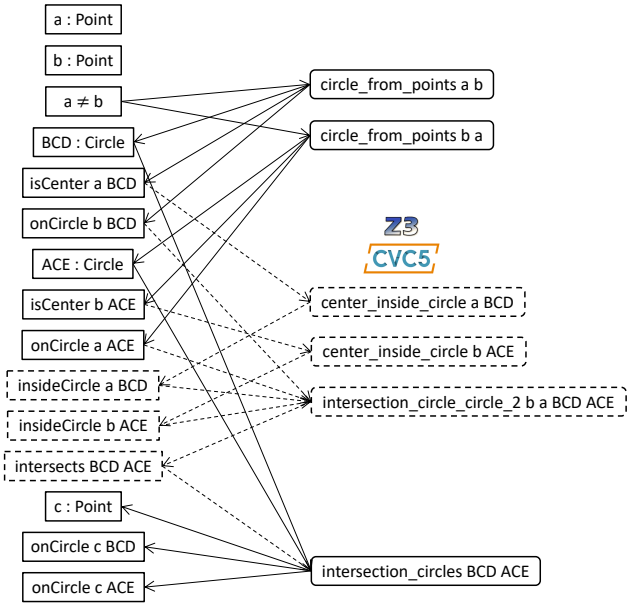


Figure 2. Three steps in the proof in Fig. 1. **Left**: Geometric objects and facts. **Right**: Rules applied to construct new objects and deduce new facts. Each rule has inbound arrows from its preconditions and outbound arrows to its postconditions. **Dashed lines**: When applying a rule with missing preconditions, we try to fill the gap using a symbolic reasoning engine based on SMT solvers. These implicit steps performed by the symbolic engine can only use non-construction rules, whereas explicit steps performed by humans (or machine learning models) can use any rules.

3.3. Symbolic Reasoning Engine Based on SMT

We introduce a symbolic reasoning engine for deducing "obvious" facts from existing geometric objects and relations. It has two important uses in LeanEuclid: (1) diagrammatic reasoning, as shown in the previous example, and (2) checking the equivalence between theorems.

In Euclidean geometry, a natural way to measure obviousness is whether construction rules are used. Applying construction rules (e.g., drawing auxiliary lines) requires creativity, which is non-obvious and hard to automate. In contrast, non-construction rules are mechanic and more amenable to automated reasoning. Intuitively, they do not introduce new objects and will not explode the search space. Therefore, we design the symbolic engine to be an automated theorem prover equipped with non-construction rules in E.

Instead of generic first-order provers (Schulz, 2013), our symbolic engine builds on top of SMT solvers. They have

[2] "Non-construction rules" include diagrammatic, metric, and transfer rules in E's original terminology (Avigad et al., 2009).

efficient decision procedures for handling real numbers, which is important for reasoning about geometric quantities such as lengths, angles, and areas. Given a target conclusion (e.g., "`intersectsCircle BCD ACE`" in Fig. 2), our symbolic engine proves it by refutation: It generates a list of SMT formulas, including all non-construction rules in E, all existing objects/relations, and the *negation* of the conclusion. Then, it runs off-the-shelf SMT solvers to check if the list of formulas is satisfiable. The target conclusion is successfully proved if the solver returns UNSAT (not satisfiable).

3.4. LeanEuclid: Overview and Dataset Statistics

In constructing LeanEuclid, we manually formalized 173 theorems/proofs in Lean. 48 examples are from Euclid's *Elements* (Book I),[3] and 125 are adapted from UniGeo (Chen et al., 2022). Our formal proofs of *Elements* vindicate Euclid's use of diagrams, as they follow Euclid's texts closely and leave diagrammatic reasoning implicit. We identified several small gaps in *Elements*. Euclid occasionally used a stronger version of a theorem he had proved. Furthermore, when there were multiple cases, Euclid would discuss only one and omit others. Most omissions are acceptable in modern mathematics, e.g., when multiple cases are symmetric. The only omission we find questionable is in the proof of Proposition 24, where Euclid discussed only a simple case but omitted a more challenging case (details in Appendix B).

UniGeo contains problems in five categories: triangle, congruent, similarity, quadrilateral, and parallel. LeanEuclid includes 25 random problems from each category. Problems from UniGeo are generally easier than those from *Elements*. Each problem comes with a diagram and a textual question. Unlike *Elements*, UniGeo's text does not include complete information about the problem, so we manually add missing diagrammatic details to the text.

4. Autoformalizing Euclidean Geometry with Large Language Models

Next, we describe our approach to autoformalizing (and evaluating) theorems and proofs targeting LeanEuclid.

4.1. Autoformalizing and Evaluating Theorems

Autoformalizing a theorem statement requires taking natural language input and generating the corresponding formal theorem in Lean. We use LLMs' in-context learning capability for autoformalization. Our prompt template contains an overview of LeanEuclid syntax, some examples of well-formed formulas, and various guidelines for the task. For few-shot learning, we also include k examples of informal-formal pairs as in-context demonstrations. The complete prompt template is in Appendix D.

When using our symbolic engine for checking the equivalence between theorems, we call it E3 (**E**uclidean **E**quivalence **E**ngine). We also provide a Python wrapper for E3, supporting various configuration options and easy integration with different LLMs. We provide a high-level overview of E3's two primary evaluation procedures: logical equivalence and approximate equivalence.

Logical Equivalence. The logical equivalence procedure takes a predicted formula T_{pred} and ground truth formula T_{gt} and attempts to prove $T_{pred} \iff T_{gt}$ using SMT solvers. E3 performs a separate satisfiability check for each direction, each of which can succeed or fail.[4] Of course, T_{pred} may actually be unsatisfiable (meaning it contains a contradiction), in which case we will always be able to prove $T_{pred} \implies T_{gt}$. As such, E3 also performs a separate check to see whether a contradiction can be found in T_{pred}.

Approximate Equivalence. If logical equivalence cannot be proven, we may still want to gauge whether T_{pred} is a semantically "close" formalization of T_{gt}. For instance, T_{pred} may differ from T_{gt} by only a single extra precondition or postcondition. To this end, E3 also supports *approximate equivalence checking*, which is a low-level semantic comparison between the two formulas. Specifically, E3 will attempt to separately prove each of the preconditions and the postconditions of each formula in a clause-by-clause fashion. However, this requires choosing a unification of the bound variables of the two formulas. Since trying each possible unification is infeasible, we instead choose the best n unifications using a string similarity heuristic. For each candidate unification, we check how many pre/postconditions can be proved from one proposition to the other. [5]

4.2. Autoformalizing Proofs

To autoformalize proofs, we provide the LLM with a natural language description of the proposition, its natural language proof, and a formalization of the proposition in LeanEuclid. Its task is to produce a formal proof consisting of a sequence of tactics. We provide a prompt template similar to the one used to autoformalize theorem statements. In addition to a brief overview of system E, we describe the custom LeanEuclid tactics (e.g., `euclid_apply`), and provide a list of axioms and theorems the model can use in its proof. Beyond the core axioms of E, the list of helper theorems available to the model depends on the proposition being proven. For

[3] We use an open-source version of Euclid's *Elements* at `https://github.com/rfitzp/Elements`.

[4] E3 uses not only non-construction rules but also a handful of construction rules (details in Appendix A).

[5] E3 currently only performs approximate equivalence checking when the predicted formula has the same number of bound variables as the ground truth formula.

example, if we are formalizing the proof of Proposition 10 in *Elements*, then the set of available helper theorems will include all versions of Propositions 1–9. Once the autoformalized proof is generated, we check its correctness using Lean and attempt to fill in diagrammatic reasoning gaps using the symbolic reasoning engine. The complete prompt template is given in Appendix D.

5. Experiments

5.1. Autoformalizing Theorem Statements

Experimental Setup. We evaluated the efficacy of LLMs on autoformalizing theorem statements using GPT-4 and GPT-4V. The input modality for GPT-4 was restricted to textual questions, while GPT-4V extended this by including diagrams as well. To implement few-shot learning, we randomly selected five propositions from Euclid's *Elements* and five problems from each category in the UniGeo dataset, serving as in-context learning examples. We then use E3 to automatically evaluate the results of each round. To see how well E3 correlates with human evaluation, we manually evaluate a sample of formalized theorems from *Elements* to identify any false negatives/positives.

	GPT-4			GPT-4V		
Dataset	0-shot	1-shot	5-shot	0-shot	1-shot	5-shot
Elements	2.3%	4.7%	16.3%	2.3%	4.7%	20.9%
UniGeo	3.0%	9.0%	20.0%	5.0%	10.0%	21.0%
Overall	2.8%	7.7%	18.9%	4.2%	8.4%	21.0%

Table 1. Percentage of proved semantic equivalences from autoformalized theorem statements from *Elements* and UniGeo. Experiments were conducted in January 2024 using `gpt-4-1106-preview` and `gpt-4-1106-vision-preview`.

Results. Table 1 shows the accuracies of autoformalizing theorem statements, where correctness is measured by using E3 to check logical equivalence. Results on the *Elements* and UniGeo parts of LeanEuclid are also shown separately. Across both parts, few-shot learning with 5 shots produces correct formalizations at a rate of 21%. We also see a small improvement when visual inputs are included. We emphasize that Table 1 was generated automatically by E3. While some autoformalized theorem statements are syntactic matches of their ground truth and, therefore, easily identified as correct, others are not. Manual evaluation of these instances is expensive and tedious, whereas E3 is able to identify these cases easily.

For cases where equivalence fails, E3 reports whether either of $T_{gt} \implies T_{pred}$ or $T_{pred} \implies T_{gt}$ can be proved, allowing us to partition the failed cases into different classes. For instance, two rounds of autoformalization with GPT-4 (1-shot) yielded 67 rejected predictions, for 31 of these we could prove only $T_{gt} \implies T_{pred}$, for 15 we could prove only $T_{pred} \implies T_{gt}$, and for 21 we could prove neither.

Given that only a minority of predictions are provably correct, we can send the remainder to E3's approximate equivalence checker to glean "close" formalizations. For brevity, we only showcase this evaluation on the results of GPT-4 (5-shot) on Euclid's *Elements*. Of the 36 rejected predictions, 10 possess the correct quantity and type of bound variables, and so are amenable to approximate analysis. The results of this analysis are shown in Fig. 3. For each proposition, we show the total number of clauses (preconditions and postconditions of both formulas) and the number of clauses that could be proved. We can see that, in addition to the 7 propositions that were provably equivalent, this round produced 5 propositions for which more than 90% of all proof obligations can be resolved. Some interesting examples produced during our experiments are in Appendix E.

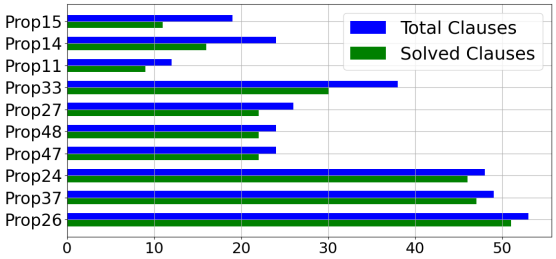


Figure 3. Approximate equivalence checking results for theorems from *Elements* formalized by GPT-4 (5-shots).

Comparison with Manual Evaluation. To compare the accuracy of E3 to manual evaluation, we took a sample of 86 formalizations and investigated them for false negatives. The examples were taken from the results of GPT-4 and GPT-4V (both 5-shot). Among the 86 autoformalized theorem statements, 16 were proved equivalent to the ground truth, and all 16 were judged to be correct by humans. The remaining 70 could not be proved equivalent, but human inspection revealed that 3 should be equivalent to the ground truth. This gives us an estimated false negative rate of 15.8%. These instances are shown in Appendix E.

With respect to false positives, we did not find indications of soundness bugs in E3. However, a false positive can occur when an autoformalized theorem *happens* to be provably equivalent to the ground truth, but is unlikely to be identified by a human as a "faithful" formalization of the given proposition. We identified only one such case from our experiments, which is also shown in Appendix E.

Why is Formalizing Theorem Statements Hard? Compared to previous autoformalization results (Wu et al., 2022), our experiments show that the models struggle to correctly formalize most of the theorems in our dataset. We suspect this is primarily a result of using the formal system E as a specification language. E is designed primarily as a proof system, and not as a specification language; as noted by Avigad, many basic relation constructs in E (e.g., `between`, `sameSide`) are almost never mentioned explicitly in Euclid's actual writing (Avigad et al., 2009). Furthermore, the language can only refer to composite structures (angles, triangles, etc.) in terms of their atomic components (points, lines, etc.). This makes theorem statements in E relatively verbose, and this verbosity introduces more room for the model to make small mistakes.

In summary, while state-of-the-art models struggle to successfully autoformalize many of the theorems in our dataset, we see that `E3` can successfully identify and quantify the correctness of autoformalized theorem statements; in particular, despite being incomplete by design, `E3` only produces a small number of false negatives. We believe that `E3` can significantly facilitate the training and validation of autoformalization models targeting Euclidean geometry. Moreover, we believe that similar tools can be developed for other domains if an appropriate formal theory can be defined.

5.2. Autoformalizing Proofs

To check whether LeanEuclid is a suitable target for autoformalizing proofs, we attempted to autoformalize proofs from *Elements* and UniGeo using GPT-4 and GPT-4V.

Experimental Setup. We tested each model against 43 proofs from *Elements* and 100 proofs from UniGeo. To demonstrate concretely the capabilities and limitations of the model in writing formal LeanEuclid proofs, we attempted to formalize entire proofs from single queries, rather than using an iterative or search-based autoformalization procedure. We evaluated the formalized UniGeo proofs based on whether it is verified by Lean as-is, and experimented with 0-shot, 1-shot, and 5-shot prompts. The proofs from *Elements* are more complex, so we did not anticipate many proofs to be completely correct. Instead, we measured how much effort is required to *repair* the autoformalized proofs into proofs that are accepted by Lean.

We manually repaired each incorrectly autoformalized proof from *Elements*, attempting to make as few alterations as required. Using our ground truth proof as a reference point, we modified invalid tactics that could be repaired (e.g., by rearranging the order of its arguments), added missing tactics, and removed tactics that could not easily be repaired. Unnecessary but valid tactics were left unchanged.

	GPT-4		GPT-4V	
Category	1-shot	5-shot	1-shot	5-shot
Triangle	35%	45%	45%	70%
Similarity	5%	15%	10%	15%
Congruent	5%	25%	15%	25%
Quadrilateral	35%	25%	20%	30%
Parallel	5%	15%	5%	15%
Overall	17%	25%	19%	31%

Table 2. Percentage of successfully autoformalized proofs from UniGeo. Experiments were conducted in January 2024 using `gpt-4-1106-preview` and `gpt-4-1106-vision-preview`.

Results. Table 2 shows the results of autoformalizing proofs from UniGeo proofs. Models with 0-shot prompts are not included since they failed to autoformalize any proofs. In general, the performance of the models significantly depends on the type of geometry problems and the number of few-shot demonstrations. Compared to theorem statements, we see a more significant improvement in the success rate when visual inputs are provided for autoformalizing proofs.

When autoformalizing proofs from *Elements* with 5-shot prompts, we found that GPT-4 and GPT-4V were both only able to completely formalize the same two proofs (Propositions 1 and Proposition 17). That is to say, when combined with the UniGeo results, GPT-4 formalized correct proofs at a rate of 18.8% on LeanEuclid, while GPT-4V achieved a rate of 23.1%. The remaining 41 proofs from *Elements* required some degree of repair to be accepted by Lean.

While it is difficult to precisely measure the quality of imperfect proofs, we can gain a rough approximation by computing the Levenshtein ratio between the original and repaired proofs. Doing so reveals that, for GPT-4, the autoformalized proofs had a median similarity ratio of 61.7% compared to their repaired versions, with proofs in the 75th percentile scoring at least 75.2%. For GPT-4V, the median similarity ratio was 64.0%, and the proofs in the 75h percentile proofs scored at least 72.9%. Moreover, we found that many of the modifications required to repair the proofs are very simple, such as strengthening a theorem slightly or rearranging tactics arguments; in general, the models are good at choosing relevant theorems, even if they do not invoke them correctly. Additional data and examples are in Appendix F.

We believe that these results reflect well on LeanEuclid as a target language for autoformalizing Euclidean proofs. In particular, our tactic language and proof automation allow the model to focus only on explicit reasoning steps in the input text. This means the resulting proofs are much shorter than they would be if all reasoning steps were made explicit (and, as a result, they are easier to repair). Given that these results were obtained from standalone queries, we expect

that neural theorem provers or proof repair tools targeting LeanEuclid could be used to successfully formalize a significant portion of the proofs in our dataset.

6. Conclusion

We have demonstrated Euclidean geometry as an attractive target for autoformalization. With our SMT-based symbolic engine, it is feasible to (1) automatically evaluate autoformalized theorems statements and (2) have the model autoformalize only explicit proof steps, leaving diagrammatic reasoning implicit. We have constructed the LeanEuclid benchmark to facilitate future research on autoformalizing Euclidean geometry.

7. Limitations and Open Problems

LeanEuclid's size (173 examples) is reasonable as a benchmark for testing, e.g., considering HumanEval (Chen et al., 2021b), the most widely used benchmark for code generation, which has only 164 examples. Nevertheless, more data examples are necessary for finetuning-based approaches. A promising way of collecting examples inexpensively is to automatically translate existing geometry datasets, e.g., the synthetic dataset in AlphaGeometry, into our formalism.

Even though we use GPT-4 in experiments, data contamination is unlikely since our test data, LeanEuclid, was not publicly available during the experiments. GPT-4 may have seen other formalizations of *Elements* on the Internet, but they are fundamentally different from LeanEuclid (Sec. 2).

We focus on Euclidean geometry, but findings related to LeanEuclid may be helpful to autoformalization in general. First, LeanEuclid provides a benchmark for developing domain-agnostic ideas for autoformalization, e.g., visual modality, retrieval, and learning from feedback. Second, we choose Euclidean geometry because its domain knowledge enables automated reasoning that can (1) check the equivalence between theorems and (2) fill in implicit reasoning gaps. Other domains with similar characteristics may also benefit from our methodology for constructing LeanEuclid.

Our work focuses on constructing a benchmark for autoformalization rather than introducing a novel model. Nevertheless, the LeanEuclid benchmark paves the way for future modeling efforts, such as having LLMs incorporate feedback from Lean to improve autoformalization.

Acknowledgements

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grants (individual) program. Kaiyu Yang is partially supported by Caltech's Computing, Data, and Society Postdoctoral Fellowship. Xujie Si is partially supported by the Canada CIFAR AI Chairs program. We thank Jia Deng for his contributions to the conceptualization of this project when Kaiyu Yang was a Ph.D. student under his mentorship. We are also thankful to Jeremy Avigad for his invaluable guidance on the formal system E. Special thanks go to Marsha Chechik and Kuldeep Meel for their insightful feedback on the initial version of this paper.

Impact Statement

This work aims to advance autoformalization by studying it in the controlled domain of Euclidean geometry. Autoformalization can be applied to formal mathematics, verification, and autonomous systems (Chen et al., 2023; Cosler et al., 2023; Pan et al., 2023). There are many potential societal consequences of these downstream applications, none of which we feel must be specifically highlighted here.

References

Agrawal, A., Gadgil, S., Goyal, N., Narayanan, A., and Tadipatri, A. Towards a mathematics formalisation assistant using large language models. *arXiv preprint arXiv:2211.07524*, 2022.

Avigad, J. Notes on "A formal system for Euclid's Elements". https://www.andrew.cmu.edu/user/avigad/Papers/euclid_notes.htm.

Avigad, J., Dean, E., and Mumma, J. A formal system for Euclid's Elements. *The Review of Symbolic Logic*, 2009.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

Barrett, C. and Tinelli, C. *Satisfiability modulo theories*. 2018.

Beeson, M., Narboux, J., and Wiedijk, F. Proof-checking Euclid. *Annals of Mathematics and Artificial Intelligence*, 2019.

Buchberger, B. Applications of gröbner bases in non-linear computational geometry. In *Trends in Computer Algebra*, 2005.

Chen, J., Tang, J., Qin, J., Liang, X., Liu, L., Xing, E., and Lin, L. GeoQA: A geometric question answering benchmark towards multimodal numerical reasoning. In

Findings of the Association for Computational Linguistics: ACL, 2021a.

Chen, J., Li, T., Qin, J., Lu, P., Lin, L., Chen, C., and Liang, X. UniGeo: Unifying geometry logical reasoning via reformulating mathematical expression. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Chen, Y., Gandhi, R., Zhang, Y., and Fan, C. NL2TL: Transforming natural languages to temporal logics using large language models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., and Trippel, C. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification (CAV)*, 2023.

Cunningham, G., Bunescu, R. C., and Juedes, D. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs. *arXiv preprint arXiv:2301.02195*, 2023.

de Moura, L. d. and Ullrich, S. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction (CADE)*, 2021.

First, E., Rabe, M., Ringer, T., and Brun, Y. Baldur: Whole-proof generation and repair with large language models. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

Gao, J., Pi, R., Zhang, J., Ye, J., Zhong, W., Wang, Y., Hong, L., Han, J., Xu, H., Li, Z., et al. G-LLaVA: Solving geometric problem with multi-modal large language model. *arXiv preprint arXiv:2312.11370*, 2023.

Hales, T., Adams, M., Bauer, G., Dang, T. D., Harrison, J., Le Truong, H., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., et al. A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*, volume 5, 2017.

Han, J. M., Rute, J., Wu, Y., Ayers, E., and Polu, S. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Heiberg, J. L. *Euclid's Elements*. 2007. URL https://github.com/rfitzp/Elements.

Hernandez-Espiet, A. feat: synthetic geometry. https://github.com/leanprover-community/mathlib4/pull/7300, 2023.

Hilbert, D. *Grundlagen der geometrie*. Springer-Verlag, 2013.

Jiang, A., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, Sketch, and Prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023a.

Jiang, A. Q., Li, W., and Jamnik, M. Multilingual mathematical autoformalization. *arXiv preprint arXiv:2311.03755*, 2023b.

Kazemi, M., Alvari, H., Anand, A., Wu, J., Chen, X., and Soricut, R. GeomVerse: A systematic evaluation of large models for geometric reasoning. *arXiv preprint arXiv:2312.12241*, 2023.

Lample, G., Lacroix, T., Lachaux, M.-A., Rodriguez, A., Hayat, A., Lavril, T., Ebner, G., and Martinet, X. HyperTree proof search for neural theorem proving. In *Neural Information Processing Systems (NeurIPS)*, 2022.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. CompCert—a formally verified optimizing compiler. In *Embedded Real Time Software and Systems*, 2016.

Li, Z., Sun, J., Murphy, L., Su, Q., Li, Z., Zhang, X., Yang, K., and Si, X. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*, 2024.

Liang, Z., Yang, T., Zhang, J., and Zhang, X. UniMath: A foundational and multimodal mathematical reasoner. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Lu, P., Gong, R., Jiang, S., Qiu, L., Huang, S., Liang, X., and Zhu, S.-c. Inter-GPS: Interpretable geometry problem solving with formal language and symbolic reasoning. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2021.

Miller, N. G. *A diagrammatic formal system for Euclidean geometry*. 2001.

Mumma, J. Proofs, pictures, and Euclid. *Synthese*, 2010.

Nevins, A. J. Plane geometry theorem proving using forward chaining. *Artificial Intelligence*, 1975.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Pan, J., Chou, G., and Berenson, D. Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *International Conference on Robotics and Automation (ICRA)*, 2023.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. BLEU: a method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2002.

Peng, S., Fu, D., Liang, Y., Gao, L., and Tang, Z. GeoDRL: A self-learning framework for geometry problem solving using reinforcement learning in deductive reasoning. In *Findings of the Association for Computational Linguistics: ACL*, 2023.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Schulz, S. System description: E 1.8. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2013.

Schwabhäuser, W., Szmielew, W., and Tarski, A. *Metamathematische methoden in der geometrie*. Springer-Verlag, 2013.

Song, P., Yang, K., and Anandkumar, A. Towards large language models as copilots for theorem proving in Lean. *arXiv preprint arXiv: Arxiv-2404.12534*, 2024.

The mathlib Community. The Lean mathematical library. In *Certified Programs and Proofs (CPP)*, 2020.

Trinh, T., Wu, Y., Le, Q., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 2023.

Wang, Q., Kaliszyk, C., and Urban, J. First experiments with neural translation of informal to formal mathematics. In *Conferences on Intelligent Computer Mathematics (CICM)*, 2018.

Wu, W.-t. On the decision problem and the mechanization of theorem-proving in elementary geometry. In *Selected Works Of Wen-Tsun Wu*. 2008.

Wu, Y., Jiang, A., Li, W., Rabe, M., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. In *Neural Information Processing Systems (NeurIPS)*, 2022.

Yang, K. and Deng, J. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*, 2019.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

Zhang, J.-Z., Chou, S.-C., and Gao, X.-S. Automated production of traditional proofs for theorems in Euclidean geometry I. the Hilbert intersection point theorems. *Annals of Mathematics and Artificial Intelligence*, 1995.

A. The Formal System E

We describe our variant of the formal system E in Lean-like pseudocode, including the complete list of axioms. Our design mostly follows E's original specification (Avigad et al., 2009) with a few minor deviations.

A.1. Geometry Objects

E has six types of geometric objects: points, lines, circles, segments, angles, and triangles.[6] Points, lines, and circles are basic types, whereas segments, angles, and triangles are parameterized by points. For example, points `a`, `b`, `c` can form segment `(a--b)`, angle ∠ `a:b:c`, and triangle △ `a:b:c`.

```
Point : Type
Line : Type
Circle : Type
Segment : Point → Point → Type
Angle : Point → Point → Point → Type
Triangle : Point → Point → Point → Type
```

A.2. Functions, Constants, and Relations

Segments, angles, and triangles are associated with metric properties such as length, degree, and size.

```
length : Segment → ℝ
degree : Angle → ℝ
size : Triangle → ℝ
```

Right angle (∟) is a special constant, denoting 90 degrees.

```
∟ : ℝ
```

We have nine basic predicates denoting the geometric relations between points, lines, and circles. For example, "`sameSide a b L`" means points `a` and `b` are on the same side of the line `L`. Also, "`between a b c`" means points `a`, `b`, and `c` are collinear, and `b` lies between `a` and `c`.

```
onLine : Point → Line → Prop
sameSide : Point → Point → Line → Prop
between : Point → Point → Point → Prop
onCircle : Point → Circle → Prop
insideCircle : Point → Circle → Prop
isCentre : Point → Circle → Prop
intersectsLine : Line → Line → Prop
Line.intersectsCircle : Line → Circle → Prop
Circle.intersectsCircle : Circle → Circle → Prop
```

In addition to the basic predicates, we have four predicates defined as syntactic sugars. For example, "`opposingSides a b L`" is the conjunction of "`¬(onLine a L)`", "`¬(onLine b L)`", and "`¬(sameSide a b L)`".

```
opposingSides (a : Point) (b : Point) (l : Line) : Prop := ¬(onLine a l) ∧ ¬(onLine b l)
    ∧ ¬(sameSide a b l)

outsideCircle (p : Point) (c : Circle) : Prop := ¬(insideCircle p c) ∧ ¬(onCircle p c)

formTriangle (a b c : Point) (AB BC CA : Line) : Prop := (distinctPointsOnLine a b AB) ∧ (
    onLine b BC) ∧ (onLine c BC) ∧ (onLine c CA) ∧ (onLine a CA) ∧ AB ≠ BC ∧ BC ≠ CA ∧ CA
     ≠ AB

formParallelogram (a b c d : Point) (AB CD AC BD : Line) : Prop := (onLine a AB) ∧ (onLine
     b AB) ∧ (onLine c CD) ∧ (onLine d CD) ∧ (onLine a AC) ∧ (onLine c AC) ∧ (
    distinctPointsOnLine b d BD) ∧ (sameSide a c BD) ∧ ¬(intersectsLine AB CD) ∧ ¬(
    intersectsLine AC BD)
```

[6] Triangles are called "areas" in Avigad et al. (2009).

A.3. Axioms

Axioms in E fall into five categories: construction rules, diagrammatic rules, metric rules, transfer rules, and superposition rules (the last four categories are collectively referred to as "non-construction rules" in this paper). Avigad et al. (2009) has proved E's axioms to be sound and complete through a bi-directional translation between E and Tarski's axioms (Schwabhäuser et al., 2013). E's axioms may have redundancy, as Avigad et al. (2009) did not attempt to minimize the number of axioms.

Construction Rules. We have 32 construction rules. Among them, two rules are not from E's paper but convenient for formalizing some proofs: `point_between_points_shorter_than`, `extend_point_longer`. One rule is stronger than its original version: `point_on_line_same_side`.

```
arbitrary_point : ∃ a : Point, true

distinct_points : ∀ a : Point, ∃ b : Point, a ≠ b

line_nonempty : ∀ l : Line, ∃ p : Point, onLine p l

exists_distincts_points_on_line : ∀ l : Line, ∀ a : Point, ∃ b : Point, a ≠ b ∧ onLine b l

exists_point_between_points_on_line : ∀ (L : Line) (b c : Point), distinctPointsOnLine b c
     L → ∃ a : Point, (onLine a L) ∧ (between b a c)

exists_point_between_points_not_on_line : ∀ (L M : Line) (b c : Point), (
    distinctPointsOnLine b c L) ∧ L ≠ M → ∃ a : Point, (onLine a L) ∧ (between b a c) ∧
    ¬(onLine a M)

point_between_points_shorter_than : ∀ (L : Line) (b c : Point) (s : Segment),
    distinctPointsOnLine b c L ∧ (|s| > 0) → ∃ a : Point, (onLine a L) ∧ (between b a c)
    ∧ (|b--a| < |s|)

extend_point : ∀ (L : Line) (b c : Point), distinctPointsOnLine b c L → ∃ a : Point, (
    onLine a L) ∧ (between b c a)

extend_point_not_on_line : ∀ (L M : Line) (b c : Point), (distinctPointsOnLine b c L) ∧ L
    ≠ M → ∃ a : Point, (onLine a L) ∧ (between b c a) ∧ ¬(onLine a M)

extend_point_longer : ∀ (L : Line) (b c : Point) (s : Segment), distinctPointsOnLine b c L
     → ∃ a : Point, (onLine a L) ∧ (between b c a) ∧ (|c--a| > |s|)

point_same_side : ∀ (L : Line) (b : Point), ¬(onLine b L) → ∃ a : Point, sameSide a b L

distinct_point_same_side: ∀ (L : Line) (b c : Point), ¬(onLine b L) → ∃ a : Point, a ≠ c
    ∧ (sameSide a b L)

point_on_line_same_side : ∀ (L M : Line) (b : Point), ¬(onLine b L) ∧ (intersectsLine L M)
     → ∃ a : Point, (onLine a M) ∧ (sameSide a b L)

exists_point_opposite : ∀ (L : Line) (b : Point), ¬(onLine b L) → ∃ a : Point,
    opposingSides a b L

exists_distinct_point_opposite_side : ∀ (L : Line) (b c : Point), ¬(onLine b L) → ∃ a :
    Point, a ≠ c ∧ (opposingSides a b L)

exists_point_on_circle : ∀ (α : Circle), ∃ a : Point, onCircle a α

exists_distinct_point_on_circle : ∀ (α : Circle) (b : Point), ∃ a : Point, a ≠ b ∧ (
    onCircle a α)

exists_point_inside_circle : ∀ (α : Circle), ∃ a : Point, insideCircle a α

exists_distinct_point_inside_circle : ∀ (α : Circle) (b : Point), ∃ a : Point, a ≠ b ∧ (
    insideCircle a α)
```

```
exists_point_outside_circle : ∀ (α : Circle), ∃ a : Point, outsideCircle a α

exists_distinct_point_outside_circle : ∀ (α : Circle) (b : Point),  ∃ a : Point, a ≠ b ∧
    (outsideCircle a α)

line_from_points : ∀ (a b : Point), a ≠ b → ∃ L : Line, (onLine a L) ∧ (onLine b L)

circle_from_points : ∀ (a b : Point), a ≠ b → ∃ α : Circle, (isCentre c α) ∧ (onCircle b
    α)

intersection_lines : ∀ (L M : Line), intersectsLine L M → ∃ a : Point, (onLine a L) ∧ (
    onLine a M)

intersection_circle_line : ∀ (α : Circle) (L : Line), intersectsCircle L α → ∃ a : Point,
     (onCircle a α) ∧ (onLine a L)

intersections_circle_line : ∀ (α : Circle) (L : Line), intersectsCircle L α → ∃ (a b :
    Point), (onCircle a α) ∧ (onLine a L) ∧ (onCircle b α) ∧ (onLine b L) ∧ a ≠ b

intersection_circle_line_between_points : ∀ (α : Circle) (L : Line) (b c :Point), (
    insideCircle b α) ∧ (onLine b L) ∧ (outsideCircle c α) ∧ (onLine c L) → ∃ a : Point,
    (onCircle a α) ∧ (onLine a L) ∧ (between b a c)

intersection_circle_line_extending_points : ∀ (α : Circle) (L : Line) (b c :Point), (
    insideCircle b α) ∧ distinctPointsOnLine b c L → ∃ a : Point, (onCircle a α) ∧ (
    onLine a L) ∧ (between a b c)

intersection_circles : ∀ (α β : Circle), intersectsCircle α β → ∃ a : Point, (onCircle a
    α) ∧ (onCircle a β)

intersections_circles : ∀ (α β : Circle), intersectsCircle α β → ∃ (a b : Point), (
    onCircle a α) ∧ (onCircle a β) ∧ (onCircle b α) ∧ (onCircle b β) ∧ a ≠ b

intersection_same_side : ∀ (α β : Circle) (b c d : Point) (L : Line), (intersectsCircle α
    β) ∧ (isCentre c α) ∧ (isCentre d β) ∧ (onLine c L) ∧ (onLine d L) ∧ ¬(onLine b L)
    → ∃ a : Point, (onCircle a α) ∧ (onCircle a β) ∧ (sameSide a b L)

intersection_opposite_side : ∀ (α β : Circle) (b c d : Point) (L : Line), (
    intersectsCircle α β) ∧ (isCentre c α) ∧ (isCentre d β) ∧ (onLine c L) ∧ (onLine d L)
     ∧ ¬(onLine b L) → ∃ a : Point, (onCircle a α) ∧ (onCircle a β) ∧ (opposingSides a b
    L)
```

Diagrammatic Rules. We have 36 diagrammatic rules for deducing diagrammatic facts, i.e., relations among objects. They include four rules not in E's paper: `intersection_lines_common_point`, `parallel_line_unique`, `intersection_symm`, and `parallelogram_same_side`.

```
two_points_determine_line : ∀ (a b : Point) (L M : Line), distinctPointsOnLine a b L ∧ (
    onLine a M) ∧ (onLine b M) → L = M

centre_unique : ∀ (a b : Point) (α : Circle), (isCentre c α) ∧ (isCentre b α) → a = b

center_inside_circle : ∀ (a : Point) (α : Circle), isCentre c α → insideCircle a α

inside_not_on_circle : ∀ (a : Point) (α : Circle), insideCircle a α → ¬(onCircle a α)

between_symm : ∀ (a b c : Point), between a b c → (between c b a) ∧ (a ≠ b) ∧ (a ≠ c) ∧
    ¬(between b a c)

between_same_line_out : ∀ (a b c : Point) (L : Line), (between a b c) ∧ (onLine a L) ∧ (
    onLine b L) → onLine c L

between_same_line_in : ∀ (a b c : Point) (L : Line), (between a b c) ∧ (onLine a L) ∧ (
    onLine c L) → onLine b L
```

```
between_trans_in : ∀ (a b c d : Point), (between a b c) ∧ (between a d b) → between a d c

between_trans_out : ∀ (a b c d : Point), (between a b c) ∧ (between b c d) → between a b
    d

between_points : ∀ (a b c : Point) (L : Line), (a ≠ b) ∧ (b ≠ c) ∧ (c ≠ a) ∧ (onLine a L
    ) ∧ (onLine b L) ∧ (onLine c L) → (between a b c) ∨ (between b a c) ∨ (between a c b)

between_not_trans : ∀ (a b c d : Point), (between a b c) ∧ (between a b d) → ¬(between c
    b d)

same_side_rfl : ∀ (a : Point) (L : Line), ¬(onLine a L) → sameSide a a L

same_side_symm : ∀ (a b : Point) (L : Line), sameSide a b L → sameSide b a L

same_side_not_on_line : ∀ (a b : Point) (L : Line), sameSide a b L → ¬(onLine a L)

same_side_trans : ∀ (a b c : Point) (L : Line), (sameSide a b L) ∧ (sameSide a c L) →
    sameSide b c L

same_side_pigeon_hole : ∀ (a b c : Point) (L : Line), ¬(onLine a L) ∧ ¬(onLine b L) ∧ ¬(
    onLine c L) → (sameSide a b L) ∨ (sameSide a c L) ∨ (sameSide b c L)

pasch_1: ∀ (a b c : Point) (L : Line), (between a b c) ∧ (sameSide a c L) → sameSide a b
    L

pasch_2: ∀ (a b c : Point) (L : Line), (between a b c) ∧ (onLine a L) ∧ ¬(onLine b L) →
    sameSide b c L

pasch_3: ∀ (a b c : Point) (L : Line), (between a b c) ∧ (onLine b L) → ¬(sameSide a c L)

pasch_4: ∀ (a b c : Point) (L M : Line), (L ≠ M) ∧ (onLine b L) ∧ (onLine b M) ∧
    distinctPointsOnLine a c M ∧ (a ≠ b) ∧ (c ≠ b) ∧ ¬(sameSide a c L) → between a b c

triple_incidence_1 : ∀ (L M N : Line) (a b c d : Point), (onLine a L) ∧ (onLine a M) ∧ (
    onLine a N) ∧ (onLine b L) ∧ (onLine c M) ∧ (onLine d N) ∧ (sameSide c d L) ∧ (
    sameSide b c N) → ¬(sameSide b d M)

triple_incidence_2 : ∀ (L M N : Line) (a b c d : Point), (onLine a L) ∧ (onLine a M) ∧ (
    onLine a N) ∧ (onLine b L) ∧ (onLine c M) ∧ (onLine d N) ∧ (sameSide c d L) ∧ ¬(
    sameSide b d M) ∧ ¬(onLine d M) ∧ (b ≠ a) → sameSide b c N

triple_incidence_3 : ∀ (L M N : Line) (a b c d e : Point), (onLine a L) ∧ (onLine a M) ∧ (
    onLine a N) ∧ (onLine b L) ∧ (onLine c M) ∧ (onLine d N) ∧ (sameSide c d L) ∧ (
    sameSide b c N) ∧ (sameSide d e M) ∧ (sameSide c e N) → sameSide c e L

circle_line_intersections : ∀ (a b c : Point) (L : Line) (α : Circle), (onLine a L) ∧ (
    onLine b L) ∧ (onLine c L) ∧ (insideCircle a α) ∧ (onCircle b α) ∧ (onCircle c α) ∧ (
    b ≠ c) → between b a c

circle_points_between : ∀ (a b c : Point) (α : Circle), ¬(outsideCircle a α) ∧ ¬(
    outsideCircle b α) ∧ (between a c b) → insideCircle c α

circle_points_extend : ∀ (a b c : Point) (α : Circle), ¬(outsideCircle a α) ∧ ¬(
    insideCircle c α) ∧ (between a c b) → (outsideCircle b α)

circles_intersections_diff_side : ∀ (a b c d : Point) (α β : Circle) (L : Line), (α ≠ β)
    ∧ (onCircle c α) ∧ (onCircle c β) ∧ (onCircle d α) ∧ (onCircle d β) ∧ (c ≠ d) ∧ (
    isCentre c α) ∧ (isCentre b β) ∧ (onLine a L) ∧ (onLine b L) → ¬(sameSide c d L)

intersection_lines_opposing: ∀ (a b : Point) (L M : Line), (opposingSides a b L) ∧ (onLine
     a M) ∧ (onLine b M) → intersectsLine L M

intersection_lines_common_point : ∀ (a : Point) (L M : Line), onLine a L ∧ (onLine a M) ∧
```

```
    (L ≠ M) → intersectsLine L M

parallel_line_unique : ∀ (a : Point) (L M N : Line), ¬(onLine a L) ∧ (onLine a M) ∧
    onLine a N ∧ ¬(intersectsLine L M) ∧ ¬(intersectsLine L N) → M = N

intersection_symm : ∀ (L M : Line), intersectsLine L M → intersectsLine M L

intersection_circle_line_1: ∀ (a b : Point) (α : Circle) (L: Line), ¬(outsideCircle a α)
    ∧ ¬(outsideCircle b α) ∧ (opposingSides a b L) → intersectsCircle L α

intersection_circle_line_2: ∀ (a : Point) (α : Circle) (L: Line), (insideCircle a α) ∧ (
    onLine a L) → intersectsCircle L α

intersection_circle_circle_1: ∀ (a b : Point) (α β : Circle), ¬(outsideCircle a α) ∧ ¬(
    outsideCircle b α) ∧ (insideCircle a β) ∧ (outsideCircle b β) → intersectsCircle α β

intersection_circle_circle_2: ∀ (a b : Point) (α β : Circle), (onCircle a α) → (
    insideCircle b α) → (insideCircle a β) → (onCircle b β) → intersectsCircle α β

parallelogram_same_side : ∀ (a b c d : Point) (AB CD AC BD : Line), formParallelogram a b
    c d AB CD AC BD → sameSide b d AC ∧ sameSide c d AB ∧ sameSide a b CD
```

Metric Rules. We have 11 metric rules. Unlike diagrammatic rules, metric rules deduce metric facts, e.g., about lengths, angles, and areas. In addition, since we use SMT solvers as the symbolic reasoning engine, their built-in theories on real numbers are also included implicitly as metric rules.

```
zero_segment_if : ∀ (a b : Point), |(a--b)| = 0 → a = b

zero_segment_onlyif : ∀ (a b : Point), a = b → |(a--b)| = 0

segment_gte_zero : ∀ (s : Segment), 0 ≤ s.length

segment_symmetric : ∀ (a b : Point), |(a--b)| = |(b--a)|

angle_symm : ∀ (a b c : Point), (a ≠ b) ∧ (b ≠ c) → (∠ a:b:c = ∠ c:b:a)

angle_range : ∀ (ang : Angle), (0 : ℝ) ≤ ang ∧ ang ≤ ∟ + ∟

degenerated_area : ∀ (a b : Point), Triangle.area △ a:a:b = 0

area_gte_zero : ∀ (ar : Triangle), 0 ≤ Triangle.area ar

area_symm_1 : ∀ (a b c : Point), Triangle.area (△a:b:c) = Triangle.area(△c:a:b)

area_symm_2 : ∀ (a b c : Point), Triangle.area (△ a:b:c) = Triangle.area (△a:c:b)

area_congruence : ∀ (a b c a′ b′ c′ : Point), (a--b) = (a′--b′) ∧ (b--c) = (b′--c′) ∧ (c
    --a) = (c′--a′) ∧ (∠ a:b:c) = (∠ a′:b′:c′) ∧ (∠ b:c:a) = (∠ b′:c′:a′) ∧ (∠ c:a:b) = (
    ∠ c′:a′:b′) → Triangle.area (△ a:b:c) = Triangle.area (△ a′:b′:c′)
```

Transfer Rules. We have 23 transfer rules acting as the bridge between diagrammatic facts and metric facts. They include 5 rules not in E's paper: `flat_angle_if`, `flat_angle_onlyif`, `parallelogram_area`, `sum_parallelograms_area`, and `rectangle_area`.

```
between_if : ∀ (a b c : Point), between a b c → |(a--b)| + |(b--c)| = |(a--c)|

equal_circles : ∀ (a b c : Point) (α β : Circle), (isCentre c α) ∧ (isCentre c β) ∧ (
    onCircle b α) ∧ (onCircle c β) ∧ |(a--b)| = |(a--c)| → α = β

point_on_circle_if : ∀ (a b c : Point) (α : Circle), (isCentre c α) ∧ (onCircle b α) ∧ |(
    a--c)| = |(a--b)| → onCircle c α
```

```
point_on_circle_onlyif : ∀ (a b c : Point) (α : Circle), (isCentre c α) ∧ (onCircle b α)
    ∧ (onCircle c α) → |(a--c)| = |(a--b)|

point_in_circle_if : ∀ (a b c : Point) (α : Circle), (isCentre c α) ∧ (onCircle b α) ∧
    (|(a--c)| < |(a--b)|) → insideCircle c α

point_in_circle_onlyif : ∀ (a b c : Point) (α : Circle), (isCentre c α) ∧ (onCircle b α)
    ∧ (insideCircle c α) → |(a--c)| < |(a--b)|

degenerated_angle_if : ∀ (a b c : Point) (L : Line), (a ≠ b) ∧ (a ≠ c) ∧ (onLine a L) ∧ (
    onLine b L) ∧ (onLine c L) ∧ ¬(between b a c) → ∠ b:a:c = 0

degenerated_angle_onlyif : ∀ (a b c : Point) (L : Line), (a ≠ b) ∧ (a ≠ c) ∧ (onLine a L)
     ∧ (onLine b L) ∧ (∠ b:a:c = 0) → (onLine c L) ∧ ¬(between b a c)

sum_angles_if : ∀ (a b c d : Point) (L M : Line), (onLine a L) ∧ (onLine a M) ∧ (onLine b
    L) ∧ (onLine c M) ∧ (a ≠ b) ∧ (a ≠ c) ∧ ¬(onLine d L) ∧ ¬(onLine d M) ∧ (L ≠ M) ∧ (
    ∠ b:a:c) = (∠ b:a:d) + (∠ d:a:c) → (sameSide b d M) ∧ (sameSide c d L)

sum_angles_onlyif : ∀ (a b c d : Point) (L M : Line), (onLine a L) ∧ (onLine a M) ∧ (
    onLine b L) ∧ (onLine c M) ∧ (a ≠ b) ∧ (a ≠ c) ∧ ¬(onLine d L) ∧ ¬(onLine d M) ∧ (L
    ≠ M) ∧ (sameSide b d M) ∧ (sameSide c d L) → (∠ b:a:c) = (∠ b:a:d) + (∠ d:a:c)

perpendicular_if : ∀ (a b c d : Point) (L : Line), (onLine a L) ∧ (onLine b L) ∧ (between
    a c b) ∧ ¬(onLine d L) ∧ (∠ a:c:d = ∠ d:c:b) → ∠ a:c:d = ∟

perpendicular_onlyif : ∀ (a b c d : Point) (L : Line), (onLine a L) ∧ (onLine b L) ∧ (
    between a c b) ∧ ¬(onLine d L) ∧ (∠ a:c:d = ∟) → ∠ a:c:d = ∠ d:c:b

flat_angle_if : ∀ (a b c : Point), a ≠ b ∧ b ≠ c ∧ (∠ a:b:c = ∟ + ∟) → between a b c

flat_angle_onlyif : ∀ (a b c : Point), between a b c → ∠ a:b:c = ∟ + ∟

equal_angles : ∀ (a b b′ c c′ : Point) (L M : Line), (onLine a L) ∧ (onLine b L) ∧ (onLine
     b′ L) ∧ (onLine a M) ∧ (onLine c M) ∧ (onLine c′ M) ∧ (b ≠ a) ∧ (b′ ≠ a) ∧ (c ≠ a)
    ∧ (c′ ≠ a) ∧ ¬(between b a b′) ∧ ¬(between c a c′) → (∠ b:a:c = ∠ b′:a:c′)

lines_intersect : ∀ (a b c d : Point) (L M N : Line), (onLine a L) ∧ (onLine b L) ∧ (
    onLine b M) ∧ (onLine c M) ∧ (onLine c N) ∧ (onLine d N) ∧ (b ≠ c) ∧ (sameSide a d M)
     ∧ (∠ a:b:c) + (∠ b:c:d) < ∟ + ∟ → ∃ e : Point, (onLine e L) ∧ (onLine e N) ∧ (
    sameSide e a M)

degenerated_area_if : ∀ (a b c : Point) (L : Line),  distinctPointsOnLine a b L ∧ (
    Triangle.area △ a:b:c) = 0 →  onLine c L

degenerated_area_onlyif : ∀ (a b c : Point) (L : Line), distinctPointsOnLine a b L ∧ (
    onLine c L) → (Triangle.area △ a:b:c) = 0

sum_areas_if : ∀ (a b c d : Point) (L : Line),  (onLine a L) ∧ (onLine b L) ∧ (onLine c L)
     ∧ (a ≠ b) ∧ (a ≠ c) ∧ (b ≠ c) ∧ ¬(onLine d L) ∧ (between a c b) → (Triangle.area
    △ a:c:d + Triangle.area △ d:c:b = Triangle.area △ a:d:b)

sum_areas_onlyif : ∀ (a b c d : Point) (L : Line), (onLine a L) ∧ (onLine b L) ∧ (onLine c
     L) ∧ (a ≠ b) ∧ (a ≠ c) ∧ (b ≠ c) ∧ ¬(onLine d L) ∧ (Triangle.area △ a:c:d +
    Triangle.area △ d:c:b = Triangle.area △ a:d:b) → between a c b

parallelogram_area : ∀ (a b c d : Point) (AB CD AC BD : Line), (formParallelogram a b c d
    AB CD AC BD) → Triangle.area △ a:c:d + Triangle.area △ a:d:b = Triangle.area △ b:a:c
     + Triangle.area △ b:c:d

sum_parallelograms_area : ∀ (a b c d e f : Point) (AB CD AC BD : Line), (formParallelogram
     a b c d AB CD AC BD) ∧ (between a e b) ∧ (between c f d) → Triangle.area △ a:c:f +
    Triangle.area △ a:f:e + Triangle.area △ e:f:d + Triangle.area △ e:d:b = Triangle.area
     △ a:c:d + Triangle.area △ a:d:b
```

```
rectangle_area : ∀ (a b c d : Point) (AB CD AC BD : Line), (formParallelogram a b c d AB
    CD AC BD) ∧ (∠ a:c:d = ∟) → (Triangle.area △ a:c:d + Triangle.area △ a:b:d = |(a--b)
    | * |(a--c)|) ∧ (Triangle.area △ b:a:c + Triangle.area △ b:d:c = |(a--b)| * |(a--c)|)
```

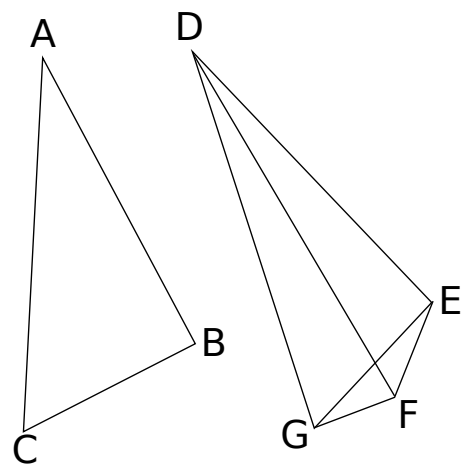
Superposition Rule. Finally, we have a superposition rule to handle Euclid's method of superposition for proving Proposition 4 and Proposition 8 in the first book of *Elements*. The rule follows the idea in Avigad.

```
superposition : ∀ (a b c d g h : Point) (AB BC AC L : Line), (formTriangle a b c AB BC AC)
     ∧ (distinctPointsOnLine d g L) ∧ ¬(onLine h L) → ∃ (b′ c′ : Point) (BC′ AC′ : Line),
     (∠ b:a:c = ∠ b′:d:c′) ∧ (∠ a:c:b = ∠ d:c′:b′) ∧ (∠ c:b:a = ∠ c′:b′:d) ∧ |(a--b)| =
    |(d--b′)| ∧ |(b--c)| = |(b′--c′)| ∧ |(c--a)| = |(c′--d)| ∧ (onLine b′ L) ∧ ¬(between
    b′ d g) ∧ (sameSide c′ h L) ∧ (distinctPointsOnLine b′ c′ BC′) ∧ (distinctPointsOnLine
     d c′ AC′)
```

Axioms Used by the Symbolic Engine. Our symbolic reasoning engine (Sec. 3.3) is based on SMT solvers. Therefore, it always has access to all built-in theories in SMT, e.g., the theory of real numbers. When used for diagrammatic reasoning, it can also use all diagrammatic rules, metric rules, and transfer rules. When used for equivalence checking, it additionally has access to nine simple construction rules: `arbitrary_point`, `distinct_points`, `line_nonempty`, `exists_point_between_points_on_line`, `distinct_point_same_side`, `exists_point_opposite`, `exists_point_on_circle`, `line_from_points`, and `intersection_lines`.

B. An Example of Logical Gaps in Euclid's Proofs Found by Formalization

Below is Euclid's proof of Proposition 24 from Heiberg (2007):



If two triangles have two sides equal to two sides, respectively, but (one) has the angle encompassed by the equal straight-lines greater than the (corresponding) angle (in the other), then (the former triangle) will also have a base greater than the base (of the latter).

Let ABC and DEF be two triangles having the two sides AB and AC equal to the two sides DE and DF, respectively. (That is), AB (equal) to DE, and AC to DF. Let them also have the angle at A greater than the angle at D. I say that the base BC is also greater than the base EF.

For since angle BAC is greater than angle EDF, let (angle) EDG, equal to angle BAC, have been constructed at the point D on the straight-line DE [Prop. 1.23]. And let DG be made equal to either of AC or DF [Prop. 1.3], and let EG and FG have been joined.

Therefore, since AB is equal to DE and AC to DG, the two (straight-lines) BA, AC are equal to the two (straight-lines) ED, DG, respectively. Also the angle BAC is equal to the angle EDG. Thus, the base BC is equal to the base EG [Prop. 1.4]. Again, since DF is equal to DG, angle DGF is also equal to angle DFG

[Prop. 1.5]. Thus, DFG (is) greater than EGF. Thus, EFG is much greater than EGF. And since triangle EFG has angle EFG greater than EGF, and the greater angle is subtended by the greater side [Prop. 1.19], side EG (is) thus also greater than EF. But EG (is) equal to BC. Thus, BC (is) also greater than EF.

Thus, if two triangles have two sides equal to two sides, respectively, but (one) has the angle encompassed by the equal straight-lines greater than the (corresponding) angle (in the other), then (the former triangle) will also have a base greater than the base (of the latter). (Which is) the very thing it was required to show.

Euclid did not cover the case where D and G are on different sides of EF (Fig. A). For this case, we come up with the proof below, which is significantly more complicated than Euclid's proof. Therefore, we argue that Euclid's proof of Proposition 24 has a logical gap.

Similar to what Euclid did, let's construct $\triangle EDG$ s.t., $\angle EDG = \angle BAC$, $|DG| = |DF|$, and $|BC| = |EG|$ (Fig. A). By Proposition 5, we have $\angle DGF = \angle DFG$; let's denote it by α. To prove $|BC| > |EF|$, we only need to prove $|EG| > |EF|$. By Proposition 19, this is further reduced to $\angle EFG > \angle EGF$. Let $x = \angle EFG$ and $y = \angle EGF$. We want to prove $x > y$.

Note that $\angle DGE = \angle DGF + \angle EGF = \alpha + y$. $\angle DFE = 2\pi - \angle DFG - \angle EFG = 2\pi - \alpha - x$. Furthermore, Proposition 17 states that the sum of any two angles in a triangle must be smaller than π. Therefore, any angle in a triangle must also be smaller than π, i.e.,

$$\begin{aligned} \alpha + y &< \pi \\ 2\pi - \alpha - x &< \pi \end{aligned}$$

Simplifying these two inequalities leads to $x > y$. QED.

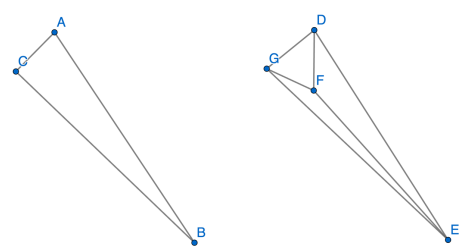


Figure A. The case in Proposition 24 missed by Euclid.

Below is our formal proof of Proposition 24, which explicitly handles the case in Fig. A following the informal proof above.

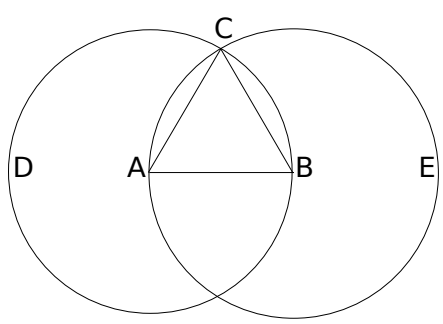
```
theorem proposition_24 : ∀ (a b c d e f : Point) (AB BC AC DE EF DF : Line),
  formTriangle a b c AB BC AC ∧ formTriangle d e f DE EF DF ∧
  (|(a--b)| = |(d--e)|) ∧ (|(a--c)| = |(d--f)|) ∧ (∠ b:a:c > ∠ e:d:f) →
  |(b--c)| > |(e--f)| :=
by
  euclid_intros
  euclid_apply (proposition_23' d e a b c f DE AB AC) as g'
  euclid_apply (line_from_points d g') as DG
  euclid_apply (extend_point_longer DG d g' (a--c)) as g''
  euclid_apply (proposition_3 d g'' a c DG AC) as g
  euclid_apply (line_from_points e g) as EG
  euclid_apply (line_from_points f g) as FG
  euclid_apply (proposition_4 a b c d e g AB BC AC DE EG DG)
  euclid_apply (proposition_5' d g f DG FG DF)
  by_cases (d.sameSide g EF)
  . euclid_assert (∠ d:f:g > ∠ e:g:f)
    euclid_assert (∠ e:f:g > ∠ e:g:f)
    euclid_apply (proposition_19 e f g EF FG EG)
    euclid_finish
```

```
. -- Omitted by Euclid.
  by_cases g.onLine EF
  . euclid_finish
  . euclid_apply (extend_point FG g f) as h
    euclid_assert ¬(g.onLine DF)
    euclid_assert ¬(e.onLine DF)
    euclid_assert (g.opposingSides e DF)
    euclid_assert h.sameSide e DF
    euclid_apply (proposition_13 d f g h DF FG)
    euclid_apply (proposition_13 e f g h EF FG)
    euclid_apply (proposition_17 d g e DG EG DE)
    euclid_apply (proposition_17 d f e DF EF DE)
    euclid_assert (∠ d:g:e < ∟ + ∟)
    euclid_assert (∠ d:f:e < ∟ + ∟)
    euclid_assert (∠ e:f:g + ∠ g:f:d + ∠ d:f:e = ∟ + ∟ + ∟ + ∟)
    euclid_assert (∠ e:f:g > ∠ e:g:f)
    euclid_apply (proposition_19 e f g EF FG EG)
    euclid_finish
```

C. Comparison to Existing Formalization of *Elements*

In this section, we compare our formalizations of *Elements* with existing formalizations, with the goal of demonstrating how our proof automation enables formal proofs that are closer in nature to the original natural language proofs. However, it is important to note that we are making a tradeoff between proof automation and end-to-end certification of proofs. As mentioned in Sec. 2, the results of SMT solvers are not certified or reconstructed as Lean proofs; instead, we introduce an unsound axiom that allows us to admit successful SMT queries to prove any proposition. In practical terms, this means that our proofs are certified up to (i) the soundness of the proof assistant kernel, (ii) any errors in our translation from Lean proof contexts to SMT-LIB queries, and (iii) the soundness of the SMT solvers themselves. By contrast, the other formalizations discussed here only require trusting the proof assistant kernel.

We use Proposition 1 as an example to compare LeanEuclid with existing formalizations of Euclid's *Elements* (Hernandez-Espiet, 2023; Beeson et al., 2019). Below is Euclid's proof of Proposition 1 from Heiberg (2007):



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.5]. Again, since the point B is the center of the circle CAE, BC is equal to BA [Def.5]. 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.

LeanEuclid. Below is our formalization of Proposition 1. Our proof is concise and follows Euclid's proof closely. It is straightforward to map the tactics in our proofs to sentences in Euclid's proof.

```
theorem proposition_1 : ∀ (a b : Point) (AB : Line),
  distinctPointsOnLine a b AB →
  ∃ c : Point, |(c--a)| = |(a--b)| ∧ |(c--b)| = |(a--b)| :=
by
  euclid_intros
  euclid_apply circle_from_points a b as BCD
  euclid_apply circle_from_points b a as ACE
  euclid_apply intersection_circles BCD ACE as c
  euclid_apply point_on_circle_onlyif a b c BCD
  euclid_apply point_on_circle_onlyif b a c ACE
  use c
  euclid_finish
```

Hernandez-Espiet (2023). Below is Hernandez-Espiet (2023)'s formalization. In this formalization, all diagrammatic inferences must be handled explicitly. For instance, in the proof of Proposition 1 (`iseqtri_sameside_of_ne`), the proof that circles BCD and ACE intersect "`(circlesInter_of_inside_on_circle bα aβ (inside_circle_of_center aα) (inside_circle_of_center bβ))`" must be supplied directly by the proof author, even though this assertion is left implicit in the text. In contrast, we use the symbolic engine to prove it automatically and implicitly.

```
theorem online_of_circlesinter (aα : CenterCircle a α) (bβ : CenterCircle b β)
    (αβ : CirclesInter α β) : ∃ c L, OnLine a L ∧ OnLine b L ∧ OnCircle c α ∧
    OnCircle c β ∧ ¬OnLine c L := by
rcases line_of_pts a b with ⟨L, aL, bL⟩; rcases not_online_of_line L with ⟨d, dL⟩;
  rcases pt_sameSide_of_circlesInter aL bL dL aα bβ αβ with ⟨c, cdL, cα, cβ⟩;
  exact ⟨c, L, aL, bL, cα, cβ, not_onLine_of_sameSide cdL⟩

theorem DiffSide_of_sameside_DiffSide (abL : SameSide a b L) (acL : DiffSide a c L) :
    DiffSide b c L := by
by_contra h; unfold DiffSide at h; push_neg at h; exact acL.2.2
 (sameSide_trans (sameSide_symm abL) (h (not_onLine_of_sameSide (sameSide_symm abL)) acL
     .2.1))

theorem DiffSide_of_circlesinter (aα : CenterCircle a α) (bβ : CenterCircle b β)
    (αβ : CirclesInter α β) : ∃ c d L, OnLine a L ∧ OnLine b L ∧ OnCircle c α ∧
    OnCircle c β ∧ OnCircle d α ∧ OnCircle d β ∧ DiffSide c d L := by
rcases online_of_circlesinter aα bβ αβ with ⟨c, L, aL, bL, cα, cβ, cL⟩;
rcases diffSide_of_not_onLine cL with ⟨e, eL, ceL⟩; rcases pt_sameSide_of_circlesInter aL
    bL eL
 aα bβ αβ with ⟨d, deL, dα, dβ⟩; exact ⟨c, d, L, aL, bL, cα, cβ, dα, dβ, DiffSide_symm
 (DiffSide_of_sameside_DiffSide (sameSide_symm deL) ⟨eL, cL, not_sameSide_symm ceL⟩)⟩

/-- Euclid I.1, construction of two equilateral Triangles -/
theorem iseqtri_iseqtri_DiffSide_of_ne (ab : a ≠ b) : ∃ c d L, OnLine a L ∧
    OnLine b L ∧ DiffSide c d L ∧ EqTri a b c ∧ EqTri a b d := by
rcases circle_of_ne ab with ⟨α, aα, bα⟩
rcases circle_of_ne (Ne.symm ab) with ⟨β, bβ, aβ⟩
rcases DiffSide_of_circlesinter aα bβ (circlesInter_of_inside_on_circle bα aβ
  (inside_circle_of_center aα) (inside_circle_of_center bβ)) with
  ⟨c, d, L, aL, bL, cα, cβ, dα, dβ, cdL⟩
have ab_ac := (on_circle_iff_length_eq aα bα).mpr cα
have bc_ba := (on_circle_iff_length_eq bβ cβ).mpr aβ
have ab_ad := (on_circle_iff_length_eq aα bα).mpr dα
have bd_ba := (on_circle_iff_length_eq bβ dβ).mpr aβ
exact ⟨c, d, L, aL, bL, cdL, EqTri_of_length_online ab aL bL cdL.1 ab_ac bc_ba,
  EqTri_of_length_online ab aL bL cdL.2.1 ab_ad bd_ba⟩
```

```
/-- Euclid I.1, construction of an equilateral Triangle on the sameside of a point -/
theorem iseqtri_sameside_of_ne (ab : a ≠ b) (aL : OnLine a L) (bL : OnLine b L) (dL : ¬
    OnLine d L):
    ∃ c, ¬OnLine c L ∧ SameSide c d L ∧ EqTri a b c := by
  rcases iseqtri_iseqtri_DiffSide_of_ne ab with ⟨c1, c2, M, aM, bM, c1c2M, eqtri1, eqtri2⟩
  rcases sameSide_or_of_diffSide' dL (by rwa [line_unique_of_pts ab aM bM aL bL] at c1c2M)
    with c1dL | c2dL
  refine ⟨c1, not_onLine_of_sameSide c1dL, c1dL, eqtri1⟩
  refine ⟨c2, not_onLine_of_sameSide c2dL, c2dL, eqtri2⟩

/-- Euclid I.1, construction of a single equilateral Triangle -/
theorem iseqtri_of_ne (ab : a ≠ b) : ∃ c, EqTri a b c :=
  by rcases iseqtri_iseqtri_DiffSide_of_ne ab with ⟨c, -, -, -, -, -, eqtri, -⟩; exact ⟨c,
      eqtri⟩
```

Beeson et al. (2019). Below is Beeson et al. (2019)'s formalization in Coq. Similar to Hernandez-Espiet (2023), it formalizes diagrammatic reasoning manually. As a result, their formal proofs tend to be longer than ours, and it is harder discern the correspondence with Euclid's proofs.

```
Lemma proposition_01 :
   forall A B,
   neq A B ->
   exists X, equilateral A B X /\ Triangle A B X.
Proof.
intros.
let Tf:=fresh in
assert (Tf:exists J, CI J A A B) by (conclude postulate_Euclid3);destruct Tf as [J];
    spliter.
assert (neq B A) by (conclude lemma_inequalitysymmetric).
let Tf:=fresh in
assert (Tf:exists K, CI K B B A) by (conclude postulate_Euclid3);destruct Tf as [K];
    spliter.
let Tf:=fresh in
assert (Tf:exists D, (BetS B A D /\ Cong A D A B)) by (conclude lemma_localextension);
    destruct Tf as [D];spliter.
assert (Cong B A B A) by (conclude cn_congruencereflexive).
assert (OutCirc D K) by (conclude_def OutCirc) .
assert (eq B B) by (conclude cn_equalityreflexive).
assert (InCirc B K) by (conclude_def InCirc ).
assert (Cong A B A B) by (conclude cn_congruencereflexive).
assert (OnCirc B J) by (conclude_def OnCirc ).
assert (OnCirc D J) by (conclude_def OnCirc ).
assert (eq A A) by (conclude cn_equalityreflexive).
assert (InCirc A J) by (conclude_def InCirc ).
let Tf:=fresh in
assert (Tf:exists C, (OnCirc C K /\ OnCirc C J)) by (conclude postulate_circle_circle);
    destruct Tf as [C];spliter.
assert (Cong A C A B) by (conclude axiom_circle_center_radius).
assert (Cong A B A C) by (conclude lemma_congruencesymmetric).
assert (Cong B C B A) by (conclude axiom_circle_center_radius).
assert (Cong B C A B) by (forward_using lemma_congruenceflip).
assert (Cong B C A C) by (conclude lemma_congruencetransitive).
assert (Cong A B B C) by (conclude lemma_congruencesymmetric).
assert (Cong A C C A) by (conclude cn_equalityreverse).
assert (Cong B C C A) by (conclude lemma_congruencetransitive).
assert (equilateral A B C) by (conclude_def equilateral ).
assert (neq B C) by (conclude axiom_nocollapse).
assert (neq C A) by (conclude axiom_nocollapse).
assert (~ BetS A C B).
 {
 intro.
 assert (~ Cong A C A B) by (conclude lemma_partnotequalwhole).
```

```
 assert (Cong C A A C) by (conclude cn_equalityreverse).
 assert (Cong C A A B) by (conclude lemma_congruencetransitive).
 assert (Cong A C C A) by (conclude cn_equalityreverse).
 assert (Cong A C A B) by (conclude lemma_congruencetransitive).
 contradict.
 }
assert (~ BetS A B C).
 {
 intro.
 assert (~ Cong A B A C) by (conclude lemma_partnotequalwhole).
 assert (Cong A B C A) by (conclude lemma_congruencetransitive).
 assert (Cong C A A C) by (conclude cn_equalityreverse).
 assert (Cong A B A C) by (conclude lemma_congruencetransitive).
 contradict.
 }
assert (~ BetS B A C).
 {
 intro.
 assert (~ Cong B A B C) by (conclude lemma_partnotequalwhole).
 assert (Cong B A A B) by (conclude cn_equalityreverse).
 assert (Cong B A B C) by (conclude lemma_congruencetransitive).
 contradict.
 }
assert (~ Col A B C).
 {
 intro.
 assert (neq A C) by (conclude lemma_inequalitysymmetric).
 assert ((eq A B \/ eq A C \/ eq B C \/ BetS B A C \/ BetS A B C \/ BetS A C B)) by (
     conclude_def Col ).
 contradict.
 }
assert (Triangle A B C) by (conclude_def Triangle ).
close.
Unshelve.
all: (exact A).
Qed.
```

D. Autoformalization Prompts

Here, we share the prompt templates used in our experiments.

D.1. Theorem Statement Formalization

To autoformalize theorem statements from *Elements* and UniGeo, we used the prompt shown below. We also prepend the instructions with a header describing the task (i.e., "Your task is to take an English statement of a theorem from Euclidean Geometry and formalize it using Lean 4 programming language, adhering to the following structures and guidelines".

One additional intricacy in formalizing theorem statements from *Elements* is that, unlike in modern mathematics, Euclid does not always state a complete logical theorem statement before beginning a proof. He often begins with an imperative assertion (e.g., Proposition 9: "To cut a given rectilinear angle in half") whose proper logical interpretation is clarified at the conclusion of the proof (e.g., "Thus, angle DAF is equal to angle EAF. Thus, the given rectilinear angle BAC has been cut in half by the straight-line AF"). In other words, to determine the proper logical statement of the theorem, both the beginning *and* the end of the proof must sometimes be considered. During preliminary experiments, we attempted to autoformalize theorem statements by giving the model the entire natural language proof, but we found that the resulting theorem statements were often very poor. Instead, we manually removed the majority of the "proof" portion of the natural language text, leaving only the portions required to obtain a reasonable formalization of the theorem statement. This context was also included as part of the prompt header when formalizing theorems from *Elements*. These abbreviated versions of *Elements* proofs are also available as part of LeanEuclid.

```
/- Basic Geometric Sorts -/
axiom Point : Type
```

```
axiom Line : Type
axiom Circle : Type
/- Inductive Types for Geometric Entities -/
inductive Angle | right | ofPoints (A B C : Point)
inductive Segment | endpoints (a b : Point)
inductive Triangle | ofPoints (a b c : Point)
/- Notations and Macros for Geometric Entities -/
"|(a-- b)|" means the length of the line segment between point a and point b.
"∠  a:b:c" means the degree of the angle formed by three points a, b, and c.
"∟ " means the right angle.
"△  a:b:c" means the triangle formed from points a, b and c.
"Triangle.area △  a:b:c" means the area of the triangle formed by points a, b and c.
/- Relations and Axioms for Geometric Sorts -/
namespace Point
def onLine (a : Point) (L : Line) -- point a is on line L.
def sameSide (a b : Point) (L : Line) -- point a and b are on the same side of line L.
def opposingSides (a b : Point) (L : Line) -- point a and b are on the opposite sides of
    line L.
def collinear (a b c : Point) -- points a, b and c are collinear.
end Point
namespace Line
def intersectsLine (L M : Line) -- two lines L and M intersect at some point.
def intersectsCircle (L : Line) (C : Circle) -- line L and circle C intersect.
end Line
namespace Circle
def onCircle (a: Point) (C: Circle) -- point a is on circle C.
def insideCircle (a: Point) (C: Circle) -- point a is inside circle C.
def outsideCircle (a: Point) (C: Circle)-- point a is outside circle C.
def isCentre (a: Point) (C: Circle) -- point a is on the center circle C.
def intersectsCircle (C1 C2: Circle) -- circle C1 and C2 intersect.
end Circle
namespace Triangle
def congruent (T1 T2 : Triangle) -- triangle T1 and T2 are congruent.
def similar (T1 T2 : Triangle) -- triangle T1 and T2 are similar.
end Triangle
/- Geometric Relations -/
def distinctPointsOnLine (a b : Point) (L : Line) -- points a and b are distinct and on
    line L.
def twoLinesIntersectAtPoint (AB BC : Line) (b : Point) -- line AB and BC intersect at
    point b.
def between (a b c : Point) -- points a, b and c collinear and cyclically ordered.
def formTriangle (a b c : Point) (AB BC CA : Line) -- point a, b and c form a triangle,
    where point a and b are on line AB, point b and c are on line BC, point a and c are on
     line CA.
def formRectilinearAngle (a b c : Point) (AB BC : Line)  -- point a, b and c form a
    rectilinear angle where a≠b and b≠c.
def formParallelogram (a b c d : Point) (AB CD AC BD : Line) -- point a, b, d and c (in
    clockwise/counterclockwise order) form a parallelogram, where point a and b are on
    line AB,  where point c and d are on line CD,  where point a and c are on line AC,
    where point b and d are on line BD.
def formQuadrilateral (a b c d : Point) (AB CD BC AD : Line)  -- point a, b, c and d (in
    clockwise/counterclockwise order) form a quadrilateral, where point a and b are on
    line AB,  where point c and d are on line CD,  where point b and c are on line BC,
    where point a and d are on line AD.
/-  Guidelines: -/
1. Proposition Format: Your proposition must be of the form <<< ∀  (...) P_1 ∧  P_2 ... ∧
     P_n →  ∃  (...) Q_1 ∧ Q_2 ... ∧ Q_m >>> where where each P_i and Q_i is built from
    the above building blocks using conjunction (∧) disjunction (∨) and negation (¬). Note
     that there may be zero existentially quantified variables.
2. Implication: There can be only a single implication in the formula; either side of the
    implication must be a conjunction of formulae.
3. Numeric Values Restrictions: Denote 90-degree angle by ∟, 180-degree angle by ∟ + ∟,
    etc. Also, when referring to segments, we always mean its length (i.e. |(a--b)|).
4. Quantified Variables: Your quantified variables must be limited to primitive geometric
    types: points, lines, and circles. ALL bound variables must be mentioned at some point
```

```
    .
5. Intermediate Variables: You should never define an intermediate variable inside the
    proposition. For example, "let X := (something);" is NOT allowed.
6. Axioms: You should only use the provided axioms. For example, Line L is parallel to
    line M can be expressed as ¬(L.intersectsLine M). Do not use Line.Parallel L M.
7. Response Format: Present your formalized Lean expression within triple angle brackets
    (<<< Lean expression here >>>). Do not add any annotations/explanations, or markdown
    syntax.
```

D.2. Proof Formalization (*Elements*)

To formalize proofs from *Elements*, we included (1) all inference rules that were used in our manual formalization of *Elements* and (2) all variations of all theorem statements that precede the target theorem. We also provided 5 hardcoded examples (Propositions 2, 6, 12, 32, 42) which were chosen to showcase a variety of proof formats and tactic invocations. The token {PRECEDING_THMS} is substituted at formalization time by the list of visible theorems allowed for the target proposition. For instance, when formalizing a proof of Proposition n in *Elements*, the model is allowed to use proofs of Propositions 1 through $n-1$, and all their variations.

```
/- Basic Geometric Sorts -/
axiom Point : Type
axiom Line : Type
axiom Circle : Type

/- Notations and Macros for Geometric Entities -/
"|(a--b)|" means the length of the line segment between point a and point b.
"∠ a:b:c" means the degree of the angle formed by points a, b, and c.
"∟" means the right angle.
"△ a:b:c" means the triangle formed from points a, b and c.
"Triangle.area △ a:b:c" means the area of the triangle formed by points a, b and c.
/- Relations and Axioms for Geometric Sorts -/
namespace Point
def onLine (a : Point) (L : Line) -- point a is on line L.
def sameSide (a b : Point) (L : Line) -- point a and b are on the same side of line L.
def opposingSides (a b : Point) (L : Line) -- point a and b are on the opposite sides of
    line L.
def collinear : (a b c : Point) -- points a, b and c are collinear.
end Point
namespace Line
def intersectsLine (L M : Line) -- two lines L and M intersect at some point.
def intersectsCircle (L : Line) (C : Circle) -- line L and circle C intersect.
end Line
namespace Circle
def onCircle (a: Point) (C: Circle) -- point a is on circle C.
def insideCircle (a: Point) (C: Circle) -- point a is inside circle C.
def outsideCircle (a: Point) (C: Circle)-- point a is outside circle C.
def isCentre (a: Point) (C: Circle) -- point a is on the center circle C.
def intersectsCircle (C1 C2: Circle) -- circle C1 and C2 intersect.
end Circle

-- Abbreviations --
abbrev distinctPointsOnLine (p q : Point) (L : Line) :=
   P.onLine L ∧ Q.onLine L ∧ P ≠ Q


abbrev Point.opposingSides (a b : Point) (l : Line) :=
  ¬ a.onLine l  ∧ ¬ b.onLine l ∧ ¬ sameSide a b l

abbrev Point.outsideCircle (p c : Point) :=
   ¬ p.insideCircle c ∧ ¬ p.onCircle c

abbrev formTriangle (a b c : Point) (AB BC CA : Line) : Prop :=
  distinctPointsOnLine a b AB ∧
  b.onLine BC ∧ c.onLine BC ∧ c.onLine CA ∧ a.onLine CA ∧
```

```
  AB ≠ BC ∧ BC ≠ CA ∧ CA ≠ AB

abbrev formRectilinearAngle (a b c : Point) (AB BC : Line) :=
  distinctPointsOnLine a b AB ∧ distinctPointsOnLine b c BC

abbrev formParallelogram (a b c d : Point) (AB CD AC BD : Line) : Prop :=
    a.onLine AB ∧ b.onLine AB ∧ c.onLine CD ∧ d.onLine CD ∧ a.onLine AC ∧ c.onLine AC ∧
        distinctPointsOnLine b d BD ∧
    (a.sameSide c BD) ∧ ¬(AB.intersectsLine CD) ∧ ¬(AC.intersectsLine BD)

Your proofs can make use of the following construction rules and inference rules:
Construction Rules:
- Points -
axiom distinct_points : ∀ p : Point, ∃ q : Point, p ≠ q
axiom line_nonempty : ∀ l : Line, ∃ p : Point, p.onLine l
axiom exists_distincts_points_on_line : ∀ l : Line, ∀ p : Point, ∃ p' : Point, p ≠ p' ∧ p
    '.onLine l
axiom exists_point_between_points_on_line : ∀ (L : Line) (b c : Point),
    distinctPointsOnLine b c L → ∃ a : Point, (a.onLine L) ∧ (between b a c)
axiom point_between_points_shorter_than : ∀ (L : Line) (b c : Point) (s : Segment),
    distinctPointsOnLine b c L ∧ (|s| > 0) → ∃ a : Point, (a.onLine L) ∧ (between b a c)
    ∧ (|(b--a)| < s)
axiom extend_point : ∀ (L : Line) (b c : Point), distinctPointsOnLine b c L → ∃ a : Point
    , (a.onLine L) ∧ (between b c a)
axiom extend_point_not_on_line : ∀ (L M : Line) (b c : Point), distinctPointsOnLine b c L
    ∧ L ≠ M → ∃ a : Point, (a.onLine L) ∧ (between b c a) ∧ ¬(a.onLine M)
axiom extend_point_longer : ∀ (L : Line) (b c : Point) (s : Segment), distinctPointsOnLine
     b c L → ∃ a : Point, (a.onLine L) ∧ (between b c a) ∧ (|(c--a)| > s)
axiom point_on_line_same_side : ∀ (L M : Line) (b : Point), ¬(b.onLine L) ∧ (L.
    intersectsLine M) → ∃ a : Point, a.onLine M ∧ a.sameSide b L
axiom exists_point_opposite : ∀ (L : Line) (b : Point), ¬(b.onLine L) → ∃ a : Point, a.
    opposingSides b L
- Lines and Circles -
axiom line_from_points : ∀ (a b : Point), a ≠ b → ∃ L : Line, (a.onLine L) ∧ (b.onLine L)
axiom circle_from_points : ∀ (a b : Point), a ≠ b → ∃ α : Circle, (a.isCentre α) ∧ (b.
    onCircle α)
- Intersections -
axiom intersection_lines : ∀ (L M : Line), L.intersectsLine M → ∃ a : Point, (a.onLine L)
     ∧ (a.onLine M)
axiom intersection_circle_line : ∀ (α : Circle) (L : Line), L.intersectsCircle α → ∃ a :
    Point, (a.onCircle α) ∧ (a.onLine L)
axiom intersections_circle_line : ∀ (α : Circle) (L : Line), L.intersectsCircle α → ∃ (a
    b : Point), (a.onCircle α) ∧ (a.onLine L) ∧ (b.onCircle α) ∧ (b.onLine L) ∧ a ≠ b
axiom intersection_circle_line_between_points : ∀ (α : Circle) (L : Line) (b c :Point), (b
    .insideCircle α) ∧ (b.onLine L) ∧ (c.outsideCircle α) ∧ (c.onLine L) → ∃ a : Point, (
    a.onCircle α) ∧ (a.onLine L) ∧ (between b a c)
axiom intersection_circle_line_extending_points : ∀ (α : Circle) (L : Line) (b c :Point),
    (b.insideCircle α) ∧ distinctPointsOnLine b c L → ∃ a : Point, (a.onCircle α) ∧ (a.
    onLine L) ∧ (between a b c)
axiom intersection_circles : ∀ (α β : Circle), α.intersectsCircle β → ∃ a : Point, (a.
    onCircle α) ∧ (a.onCircle β)
axiom intersection_same_side : ∀ (α β : Circle) (b c d : Point) (L : Line), (α.
    intersectsCircle β) ∧ (c.isCentre α) ∧ (d.isCentre β) ∧ (c.onLine L) ∧ (d.onLine L)
    ∧ ¬(b.onLine L) → ∃ a : Point, (a.onCircle α) ∧ (a.onCircle β) ∧ (a.sameSide b L)
axiom intersection_opposite_side : ∀ (α β : Circle) (b c d : Point) (L : Line), (α.
    intersectsCircle β) ∧ (c.isCentre α) ∧ (d.isCentre β) ∧ (c.onLine L) ∧ (d.onLine L)
    ∧ ¬(b.onLine L) → ∃ a : Point, (a.onCircle α) ∧ (a.onCircle β) ∧ a.opposingSides b L
--- Inference Rules ---
axiom two_points_determine_line : ∀ (a b : Point) (L M : Line),distinctPointsOnLine a b L
    ∧ (a.onLine M) ∧ (b.onLine M) → L = M
axiom superposition : ∀ (a b c d g h : Point) (AB BC AC L : Line), formTriangle a b c AB
    BC AC ∧ distinctPointsOnLine d g L ∧ ¬(h.onLine L) → ∃ (b' c' : Point) (BC' AC' :
    Line), (∠ b:a:c ) = (∠ b':d:c') ∧ (∠ a:c:b) = (∠ d:c':b') ∧ (∠ c:b:a) = (∠ c':b':d)
    ∧ |(a--b)| = |(d--b')| ∧ |(b--c)| = |(b'--c')| ∧ |(c--a)| = |(c'--d)| ∧ b'.onLine L
    ∧ ¬(between b' d g) ∧ c'.sameSide h L ∧ distinctPointsOnLine b' c' BC' ∧
```

```
    distinctPointsOnLine d c' AC'
axiom between_if : ∀ (a b c : Point),between a b c → |(a--b)| + |(b--c)| = |(a--c)|
axiom equal_circles : ∀ (a b c : Point) (α β : Circle),(a.isCentre α) ∧ (a.isCentre β) ∧
    (b.onCircle α) ∧ (c.onCircle β) ∧ |(a--b)| = |(a--c)| → α = β
axiom point_on_circle_if : ∀ (a b c : Point) (α : Circle),(a.isCentre α) ∧ (b.onCircle α)
     ∧ |(a--c)| = |(a--b)| → c.onCircle α
axiom point_on_circle_onlyif : ∀ (a b c : Point) (α : Circle),(a.isCentre α) ∧ (b.onCircle
     α) ∧ (c.onCircle α) → |(a--c)| = |(a--b)|
axiom point_in_circle_if : ∀ (a b c : Point) (α : Circle),(a.isCentre α) ∧ (b.onCircle α)
     ∧ (a--c) < (a--b) → c.insideCircle α
axiom sum_angles_if : ∀ (a b c d : Point) (L M : Line),(a.onLine L) ∧ (a.onLine M) ∧ (b.
    onLine L) ∧ (c.onLine M) ∧ (a ≠ b) ∧ (a ≠ c) ∧¬(d.onLine L) ∧ ¬(d.onLine M) ∧ (L ≠ M
    ) ∧ (∠ b:a:c) = (∠ b:a:d) + (∠ d:a:c) →(b.sameSide d M) ∧ (c.sameSide d L)
axiom sum_angles_onlyif : ∀ (a b c d : Point) (L M : Line), (a.onLine L) ∧ (a.onLine M) ∧
    (b.onLine L) ∧ (c.onLine M) ∧ (a ≠ b) ∧ (a ≠ c) ∧ (d.onLine L) ∧ ¬(d.onLine M) ∧ (L
    ≠ M) ∧ (b.sameSide d M) ∧ (c.sameSide d L) →  (∠ b:a:c) = (∠ b:a:d) + (∠ d:a:c)
axiom perpendicular_if : ∀ (a b c d : Point) (L : Line),(a.onLine L) ∧ (b.onLine L) ∧ (
    between a c b) ∧ ¬(d.onLine L) ∧ (∠ a:c:d = ∠ d:c:b) →(∠ a:c:d) = ∟
axiom sum_areas_if : ∀ (a b c d : Point) (L : Line), (a.onLine L) ∧ (b.onLine L) ∧ (c.
    onLine L) ∧ (a ≠ b) ∧ (a ≠ c) ∧ (b ≠ c) ∧ ¬(d.onLine L) ∧ (between a c b) → (
    Triangle.area △ a:c:d ) + (Triangle.area △ d:c:b) = (Triangle.area △ a:d:b)
axiom rectangle :  (a b c d : Point) (AB CD AC BD : Line),
  formParallelogram a b c d AB CD AC BD ∧ (∠ a:c:d = ∟) →
  (Triangle.area △ a:c:d + Triangle.area △ a:b:d = |(a--b)| * |(a--c)|) ∧ (Triangle.area
      △  b:a:c + Triangle.area △  b:d:c) = |(a--b)| * |(a--c)|
{PRECEDING_THMS}

--- Proof DSL ---
Your response must be a tactic proof in the LeanEuclid proof DSL. This DSL is built from
    the following tactics (arguments shown in angle-brackets <> ):

* TACTIC: euclid_intros *
 Introduces universally quantified variables and premises of the current goal into the
     proof context. No names required.
* TACTIC: euclid_apply <rule> <args> *
 where <rule> is either a construction rule, inference rule, or other theorem.
 Given a rule <rule> with type of the form ∀ (<args> : Types) ... P -> Q, this tactic
     instantiates <rule> with <args>, and attempts to prove premise P from the local proof
      context using an SMT solver. If successful, propsition Q is added to the proof
     context.
usage examples :
  euclid_apply between_if a b c
  eculid_apply same_side_rfl x y L
* TACTIC: euclid_apply <rule> <args> as X *
Given a rule <rule> with type of the form ∀ (<args> : Types) ... P -> ∃ x . Q(x), this
    tactic instantiates <rule> with <args>, and attempts to prove premise P from the local
     proof context using an SMT solver. If successful, object x and premise Q(x) are added
     to the proof context.
usage examples:
  euclid_apply point_same_side a L as b
  euclid_apply line_from_points p1 p2 as M

NOTE: You can only use `euclid_apply  <rule> <args> as <X>` if the rule produces an
    existential. You should not name any propsotions introduced using `euclid_apply` e,g,
    `euclid_apply <rule> <args> as H1`.
NOTE: It is very important that *all* non-propositional (i.e., universally quantified)
    arguments are provided to the rule when invoking `euclid_apply`.

* euclid_finish *
    Attempts to resolve the proof goal using the current proof context using an SMT solver
        .
* euclid_assert <P> *
    Attempts to prove proposition <P> from the current proof context using an SMT solver.
        Equivalent to "have : <P> := by euclid_finish"
```

```
If you are proving an existentially quantified proposition, you can use the standard Lean
    tactic
` use <X>`
to provide the witness <X> for the quantifier. DO NOT use the tactic `use` if you are not
    proving an existentially quantified proposition.

You can use standard Lean tactics such as <by_cases>, <cases>, <split_ands> and <
    constructor> to structure your proof. But, you should not use imperative Lean tactics,
     such as `rw` or `simp`. You should only use the above declarative tactics.

Your response should be exactly of the following form:

    <tac_1>
    <tac_2>
    ...
    <tac_n>

where each <tac_i> is a Lean tactic adhering to the above guidelines.

DO *NOT* wrap your answer in markdown syntax, e.g. ```lean <contents> ```. It must be
    simply a Lean tactic script that can be inserted into a proof.

Here are some examples. NOTE: You may not necessarily use the propositoins shown in these
    proofs, unless they are also listed above.

Input:
 Informal Proof:
  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

 Proof Setup:
  def example : ∀ (a b c : Point) (BC : Line), distinctPointsOnLine b c BC ∧ a ≠ b → ∃ l
      : Point, |(a--l)| = |(b--c)| :=
  by

 Your Response:
  euclid_intros
  euclid_apply (line_from_points a b) as AB
  euclid_apply (proposition_1 a b AB) as d
  euclid_apply (line_from_points d a ) as AE
  euclid_apply (line_from_points d b ) as BF
  euclid_apply (circle_from_points b c) as CGH
  euclid_apply (intersection_circle_line_extending_points CGH BF b d) as g
  euclid_apply (circle_from_points d g) as GKL
  euclid_apply (intersection_circle_line_extending_points GKL AE a d) as l
  euclid_apply (point_on_circle_onlyif b c g CGH)
  euclid_apply (point_on_circle_onlyif d l g GKL)
  euclid_apply (between_if l a d )
  euclid_apply (between_if g b d )
  use l
```

```
  euclid_finish

Input:
 Informal Proof:
  To draw a straight-line perpendicular to a given infinite straight-line from a given
      point which is not on it.   Let $AB$ be the given infinite straight-line  and $C$
      the given point, which is not on ($AB$). So it is required to draw a  straight-line
       perpendicular to the given infinite straight-line $AB$ from the given point $C$,
      which is not on ($AB$).  For let point $D$ have been taken at random on the other
      side (to $C$) of  the straight-line $AB$, and let the circle $EFG$ have been drawn
      with center $C$ and radius $CD$ [Post.~3], and let the straight-line $EG$ have been
      cut in half at (point) $H$ [Prop.~1.10], and let the straight-lines $CG$, $CH$, and
      $CE$ have been joined. I say that the  (straight-line) $CH$ has been drawn
      perpendicular to the given infinite straight-line $AB$ from the given point $C$,
      which is not on ($AB$).  For since $GH$ is equal to $HE$, and $HC$ (is) common, the
      two (straight-lines) $GH$,  $HC$ are equal to the two (straight-lines) $EH$, $HC$,
      respectively, and the base $CG$ is equal to the base $CE$. Thus, the angle $CHG$ is
      equal to the angle $EHC$ [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, and the former straight-line is called
      a perpendicular to that upon which it stands [Def.~1.10].  Thus, the (straight-line)
       $CH$ has been drawn perpendicular to the given infinite straight-line $AB$ from the
       given point $C$, which is not on  ($AB$). (Which is) the very thing it was required
       to do.

 Proof Setup:
  def example : ∀ (a b c : Point) (AB : Line), distinctPointsOnLine a b AB ∧ ¬(c.onLine AB
      ) → ∃ h : Point, h.onLine AB ∧ (∠ a:h:c) = ∟ ∨ (∠ b:h:c) = ∟ :=

 Your Response:
  euclid_intros
  euclid_apply (exists_point_opposite AB c) as d
  euclid_apply (circle_from_points c d) as EFG
  euclid_apply (intersections_circle_line EFG AB) as (e, g)
  euclid_apply (proposition_10 e g AB) as h
  euclid_apply (line_from_points c g) as CG
  euclid_apply (line_from_points c h) as CH
  euclid_apply (line_from_points c e) as CE
  use h
  euclid_apply (proposition_8 h c g h c e CH CG AB CH CE AB)
  euclid_finish


Input:
 Informal Proof:
 In any triangle,  (if) one of the sides (is) produced  (then) the external angle is equal
      to the (sum of the) two internal and opposite (angles), and the (sum of the) three
     internal angles of the triangle is equal to two right-angles. Let $ABC$ be a triangle
     , and let one of its sides $BC$ have been produced to $D$. I say that the external
     angle $ACD$ is equal to the (sum of the) two internal and opposite angles $CAB$ and
     $ABC$, and the (sum of the) three internal angles of the triangle---$ABC$, $BCA$, and
      $CAB$---is equal to two right-angles.  For let $CE$ have been drawn through point
     $C$ parallel to the straight-line $AB$ [Prop.~1.31].  And since $AB$ is parallel to
     $CE$, and $AC$ has fallen across them, the alternate angles $BAC$ and $ACE$ are equal
      to one another [Prop.~1.29]. Again, since $AB$ is parallel to $CE$, and the straight
     -line $BD$ has fallen across them,  the external angle $ECD$ is equal to the internal
      and opposite (angle) $ABC$ [Prop.~1.29]. But $ACE$ was also shown (to be) equal to
     $BAC$. Thus, the whole angle $ACD$ is equal to the (sum of the) two internal and
     opposite (angles) $BAC$ and $ABC$.  Let $ACB$ have been added to both. Thus, (the sum
      of) $ACD$ and $ACB$ is equal to the (sum of the) three (angles) $ABC$, $BCA$, and
     $CAB$. But, (the sum of) $ACD$ and $ACB$ is equal to two right-angles [Prop.~1.13].
     Thus, (the sum of) $ACB$, $CBA$, and $CAB$ is also equal to two right-angles.  Thus,
     in any triangle,  (if) one of the sides  (is) produced (then) the external angle is
     equal to the (sum of the) two internal and opposite (angles), and the (sum of the)
     three internal angles of the triangle is equal to two right-angles. (Which is) the
```

```
    very thing it was required to show.

 Proof Setup:
  def example: ∀ (a b c d : Point) (AB BC AC : Line), formTriangle a b c AB BC AC ∧ (
      between b c d) → ∠ a:c:d = ∠ c:a:b + ∠ a:b:c ∧ ∠ a:b:c + ∠ b:c:a + ∠ c:a:b = ∟ + ∟
  by

 Your Response:
  euclid_intros
  have : (∠ a:c:d) = (∠ c:a:b) + (∠ a:b:c) := by
    euclid_apply (proposition_31 c a b AB ) as CE
    euclid_apply (point_on_line_same_side BC CE a ) as e
    euclid_apply (proposition_29''' b e a c AB CE AC)
    euclid_apply (proposition_29'''' e a d c b CE AB BC)
    euclid_finish
  constructor
  . assumption
  . euclid_apply (proposition_13 a c b d AC BC)
    euclid_finish



Input:
 Informal Proof:
  To construct a parallelogram equal to a given triangle in a given rectilinear angle. Let
       $ABC$ be the given triangle, and $D$ the given rectilinear angle. So it is required
       to construct a parallelogram equal to triangle $ABC$ in the rectilinear angle $D$.
      Let $BC$ have been cut in half at $E$ [Prop.~1.10], and let $AE$ have been joined.
      And let (angle) $CEF$, equal to angle $D$,  have been constructed at the point $E$
      on the straight-line $EC$ [Prop.~1.23]. And let $AG$ have been drawn through $A$
      parallel to $EC$ [Prop.~1.31], and let $CG$ have been drawn through $C$ parallel to
      $EF$ [Prop.~1.31]. Thus, $FECG$ is a parallelogram. And since $BE$ is equal to $EC$,
       triangle $ABE$ is also equal to triangle $AEC$. For they are on the equal bases,
      $BE$ and $EC$, and between the same parallels, $BC$ and $AG$ [Prop.~1.38]. Thus,
      triangle $ABC$ is double (the area) of triangle $AEC$. And parallelogram $FECG$ is
      also double (the area) of triangle $AEC$. For it has the same base as ($AEC$), and
      is between the same parallels  as ($AEC$) [Prop.~1.41]. Thus, parallelogram $FECG$
      is equal to triangle $ABC$.  ($FECG$) also has the angle $CEF$ equal to the given (
      angle) $D$. Thus, parallelogram $FECG$,  equal to the given triangle $ABC$, has been
       constructed in the angle $CEF$, which is equal to $D$. (Which is) the very thing it
       was required to do.

 Theorem Statement:
  def example : ∀ (a b c d1 d2 d3 : Point) (AB BC AC D12 D23: Line), formTriangle a b c AB
       BC AC ∧ formRectilinearAngle d1 d2 d3 D12 D23 ∧ (∠ d1:d2:d3) > 0 ∧ (∠ d1:d2:d3) <
      ∟ + ∟ → ∃ (f g e c' : Point) (FG EC EF CG : Line), formParallelogram f g e c' FG EC
      EF CG ∧ (∠ c':e:f = ∠ d1:d2:d3) ∧ (Triangle.area △ f:e:c' + Triangle.area △ f:c':g
       = Triangle.area △ a:b:c) :=
  by

 Your Response:
  euclid_intros
  euclid_apply (proposition_10 b c BC) as e
  euclid_apply (line_from_points a e) as AE
  euclid_apply (proposition_23''' e c d2 d1 d3 a BC D12 D23) as f'
  euclid_apply (line_from_points e f') as EF
  euclid_apply (proposition_31 a b c BC) as AG
  euclid_apply (intersection_lines AG EF) as f
  euclid_apply (proposition_31 c e f EF) as CG
  euclid_apply (intersection_lines CG AG) as g
  euclid_assert (formParallelogram f g e c AG BC EF CG)
  euclid_apply (proposition_38 a b e a e c AG BC AB AE AE AC)
  euclid_apply (proposition_41 f e c g a AG BC EF CG AE AC)
  use f, g, e, c, AG, BC, EF, CG
  euclid_finish
```

```
**NOTICE: in the above example, there is exactly one argument to `use` for every
    existentially quantified variables in the theoerem statement.**

Input :
  Informal Proof:
    If a triangle has two angles equal to one another then the sides subtending the equal
        angles will also be equal to one another. Let $ABC$ be a triangle having the angle
         $ABC$ equal to the angle $ACB$. I say that side $AB$ is also equal to side $AC$.
         For if $AB$ is unequal to $AC$ then one of them is greater. Let $AB$ be greater.
        And let $DB$, equal to the lesser $AC$, have been cut off from the greater $AB$ [
        Prop.~1.3].  And let $DC$ have been joined [Post.~1].  Therefore, since $DB$ is
        equal to $AC$, and $BC$ (is) common, the two sides $DB$, $BC$ are equal to the two
         sides $AC$, $CB$, respectively, and the angle $DBC$ is equal to the angle $ACB$.
        Thus, the base $DC$ is equal to the base $AB$, and the triangle $DBC$ will be
        equal to the triangle $ACB$ [Prop.~1.4], the lesser to the greater. The very
        notion (is) absurd [C.N.~5]. Thus, $AB$ is not unequal to $AC$. Thus, (it is)
        equal.  Thus, if a triangle has two angles equal to one another then the sides
        subtending the equal angles will also be equal to one another. (Which is) the very
         thing it was required to show.
  Proof Setup:
    theorem proposition_6 : ∀ (a b c : Point) (AB BC AC : Line), formTriangle a b c AB BC
        AC ∧ (∠ a:b:c = ∠ a:c:b) →(|(a--b)| = |(a--c)|) :=
  Your Response:
    euclid_intros
    by_contra
    by_cases |(a--b)| > |(a--c)|
    . euclid_apply (proposition_3 b a a c AB AC) as d
      euclid_apply (line_from_points d c) as DC
      euclid_apply proposition_4 b d c c a b AB DC BC AC AB BC
      euclid_finish
    . euclid_apply (proposition_3 c a a b AC AB) as d
      euclid_apply (line_from_points d b) as DB
      euclid_apply (proposition_4 c d b b a c AC DB BC AB AC BC)
      euclid_finish
--- End of Instruction ---
```

D.3. Proof Formalization (UniGeo)

To formalize UniGeo proofs, we likewise include those inference rules which were used in our manual formalization, as well as the relevant theorems from *Elements* . Unlike for proofs from *Elements*, we randomly select shots from a separate test set of UniGeo proofs. The guidelines are slightly different from *Elements*, to accommodate the different style of input for UniGeo proofs.

```
    --- Definitions ---
- Basic Geometric Sorts -
axiom Point : Type
axiom Line : Type
axiom Circle : Type

- Inductive Types for Geometric Entities -
inductive Angle | right | ofPoints (A B C : Point)
inductive Segment | endpoints (a b : Point)
inductive Triangle | ofPoints (a b c : Point)

- Notations and Macros for Geometric Entities -
"|(a--b)|" means the length of the line segment between point a and point b.
"∠ a:b:c" means the degree of the angle formed by points a, b, and c.
"∟" means the right angle.
"△ a:b:c" means the triangle formed from points a, b and c.
"Triangle.area △ a:b:c" means the area of the triangle formed by points a, b and c.

- Relations and Axioms for Geometric Sorts -
```

```
namespace Point
def onLine (a : Point) (L : Line) -- point a is on line L.
def sameSide (a b : Point) (L : Line) -- point a and b are on the same side of line L.
def opposingSides (a b : Point) (L : Line) -- point a and b are on the opposite sides of
    line L.
def collinear : (a b c : Point) -- points a, b and c are collinear.
end Point

namespace Line
def intersectsLine (L M : Line) -- two lines L and M intersect at some point.
def intersectsCircle (L : Line) (C : Circle) -- line L and circle C intersect.
end Line

namespace Circle
def onCircle (a: Point) (C: Circle) -- point a is on circle C.
def insideCircle (a: Point) (C: Circle) -- point a is inside circle C.
def outsideCircle (a: Point) (C: Circle)-- point a is outside circle C.
def isCentre (a: Point) (C: Circle) -- point a is on the center circle C.
def intersectsCircle (C1 C2: Circle) -- circle C1 and C2 intersect.
end Circle

namespace Triangle
-- triangle T1 and T2 are congruent (i.e satisfies the SAS, AAS, ASA or SSS condition)
def congruent (T1 T2 : Triangle)

-- triangle T1 and T2 are similar.
def similar (T1 T2 : Triangle)

-- If T1 and T2 are congruent, then the corresponding angles and sides are equal.
theorem congruent_if (T1 T2: Triangle): congruent T1 T2 → match T1,T2 with
  | (Triangle.ofPoints A B C) ,(Triangle.ofPoints D E F) =>
    |(A--B)| = |(D--E)| ∧ |(B--C)| = |(E--F)| ∧ |(A--C)| = |(D--F)| ∧ ∠ A:B:C = ∠ D:E:F
        ∧ ∠ A:C:B = ∠ D:F:E ∧ ∠ B:A:C = ∠ E:D:F

-- If T1 and T2 are similar, then the corresponding angles and the ratio of corresponding
    sides are equal.
theorem similar_if (T1 T2: Triangle): similar T1 T2 → match T1,T2 with
  | (Triangle.ofPoints A B C) ,(Triangle.ofPoints D E F) =>
    |(A--B)| / |(D--E)| = |(B--C)| / |(E--F)| ∧ |(A--B)| / |(D--E)| = |(B--C)| / |(E--F)|
    ∧ |(C--A)| / |(F--D)| = |(A--B)| / |(D--E)| ∧ ∠ A:B:C = ∠ D:E:F
    ∧ ∠ A:C:B = ∠ D:F:E ∧ ∠ B:A:C = ∠ E:D:F

- Geometric Relations -
-- points a and b are distinct and on line L.
def distinctPointsOnLine (a b : Point) (L : Line)

-- line AB and BC intersect at point b.
def twoLinesIntersectAtPoint (AB BC : Line) (b : Point)

-- points a, b and c collinear and cyclically ordered.
def between (a b c : Point)

-- point a, b and c form a triangle, where point a and b are on line AB, point b and c are
     on line BC, point a and c are on line CA.
def formTriangle (a b c : Point) (AB BC CA : Line)

-- point a, b and c form a rectilinear angle where a≠b and b≠c.
def formRectilinearAngle (a b c : Point) (AB BC : Line)

-- point a, b, d, c (in clockwise/counterclockwise order) form a parallelogram, a and b
    are on line AB, c and d are on line CD, a and c are on line AC, b and d are on line BD
def formParallelogram (a b c d : Point) (AB CD AC BD : Line)

-- point a, b, d, c (in clockwise/counterclockwise order) form a quadrilateral, a and b
    are on line AB, c and d are on line CD, a and c are on line AC, b and d are on line BD
```

```
def formQuadrilateral (a b c d : Point) (AB CD AC BD : Line)

--- Axioms ---
- Construction Rules-
axiom extend_point : ∀ (L : Line) (b c : Point), distinctPointsOnLine b c L → ∃ a : Point
    , (a.onLine L) ∧ (between b c a)

- Useful Lemmas from Euclid's Elements Book I-
-- In isosceles triangles the angles at the base equal one another, and, if the equal
    straight lines are produced further, then the angles under the base equal one another.
theorem proposition_5 : ∀ (a b c d e : Point) (AB BC AC : Line), formTriangle a b c AB BC
    AC ∧ (|(a--b)| = |(a--c)|) ∧ (between a b d) ∧ (between a c e) → (∠ a:b:c) = (∠ a:c:b
    ) ∧ (∠ c:b:d) = (∠ b:c:e)
theorem proposition_5' : ∀ (a b c : Point) (AB BC AC : Line), formTriangle a b c AB BC AC
    ∧ (|(a--b)| = |(a--c)|) → ((∠ a:b:c ) = ∠ a:c:b)

-- If a straight line stands on a straight line, then it makes either two right angles or
    angles whose sum equals two right angles.
theorem proposition_13 : ∀ (a b c d : Point) (AB CD : Line), AB ≠ CD ∧
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c d CD ∧ between d b c → ∠ c:b:a +
     ∠ a:b:d = ∟ + ∟

-- If two straight lines cut one another, then they make the vertical angles equal to one
    another.
theorem proposition_15 : ∀ (a b c d e : Point) (AB CD : Line), distinctPointsOnLine a b AB
     ∧ distinctPointsOnLine c d CD ∧ e.onLine AB ∧ e.onLine CD ∧ CD ≠ AB ∧ (between d e c
    ) ∧ (between a e b) → (∠ a:e:c ) = (∠ d:e:b) ∧ (∠ c:e:b ) = (∠ a:e:d)

-- If a straight line falling on two straight lines makes the alternate angles equal to
    one another, then the straight lines are parallel to one another.
theorem proposition_27 : ∀ (a b c d e f : Point) (AB CD EF : Line), distinctPointsOnLine a
     b AB ∧ distinctPointsOnLine c d CD ∧ distinctPointsOnLine e f EF ∧ (between a e b) ∧
    (between c f d) ∧ (b.sameSide d EF) ∧ ∠ a:e:f  = (∠ e:f:d) → ¬(AB.intersectsLine CD)
theorem proposition_27' : ∀ (a d e f : Point) (AB CD EF : Line), distinctPointsOnLine a e
    AB ∧ distinctPointsOnLine f d CD ∧ distinctPointsOnLine e f EF ∧ a.opposingSides d EF
    ∧ ∠ a:e:f = (∠ e:f:d) → ¬(AB.intersectsLine CD)

-- If a straight line falling on two straight lines makes the exterior angle equal to the
    interior and opposite angle on the same side, or the sum of the interior angles on the
     same side equal to two right angles, then the straight lines are parallel to one
    another.
theorem proposition_28 : ∀ (a b c d e f g h : Point) (AB CD EF : Line),
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c d CD ∧ distinctPointsOnLine e f
    EF ∧ (between a g b) ∧ (between c h d) ∧ (between e g h) ∧ (between g h f) ∧ (b.
    sameSide d EF) ∧ (∠ e:g:b = ∠ g:h:d ∨ ∠ b:g:h + ∠ g:h:d = ∟ + ∟) → ¬(AB.
    intersectsLine CD)

-- A straight line falling on parallel straight lines makes the alternate angles equal to
    one another, the exterior angle equal to the interior and opposite angle, and the sum
    of the interior angles on the same side equal to two right angles.
theorem proposition_29 : ∀ (a b c d e f g h : Point) (AB CD EF : Line),
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c d CD ∧ distinctPointsOnLine e f
    EF ∧ (between a g b) ∧ (between c h d) ∧ (between e g h) ∧ (between g h f) ∧ (b.
    sameSide d EF) ∧ ¬(AB.intersectsLine CD) → ∠ a:g:h = ∠ g:h:d ∧ ∠ e:g:b = ∠ g:h:d ∧
    ∠ b:g:h + ∠ g:h:d = ∟ + ∟
theorem proposition_29' : ∀ (a b c d e g h : Point) (AB CD EF : Line),
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c d CD ∧ distinctPointsOnLine g h
    EF ∧ (between a g b) ∧ (between c h d) ∧ (between e g h) ∧ (b.sameSide d EF) ∧ ¬(AB.
    intersectsLine CD) → ∠ a:g:h = ∠ g:h:d ∧ ∠ e:g:b = ∠ g:h:d ∧ ∠ b:g:h + ∠ g:h:d = ∟ +
     ∟
theorem proposition_29'' : ∀ (a b d g h : Point) (AB CD GH : Line), distinctPointsOnLine a
     b AB ∧ distinctPointsOnLine h d CD ∧ distinctPointsOnLine g h GH ∧ (between a g b) ∧
    (b.sameSide d GH) ∧ ¬(AB.intersectsLine CD) → ∠ a:g:h = ∠ g:h:d ∧ ∠ b:g:h + ∠ g:h:d
    = ∟ + ∟
theorem proposition_29''' : ∀ (a d g h : Point) (AB CD GH : Line),distinctPointsOnLine a g
```

```
    AB ∧ distinctPointsOnLine h d CD ∧ distinctPointsOnLine g h GH ∧a.opposingSides d GH
    ∧ ¬(AB.intersectsLine CD) → ∠ a:g:h = ∠ g:h:d
theorem proposition_29'''' : ∀ (b d e g h : Point) (AB CD EF : Line), distinctPointsOnLine
     g b AB ∧ distinctPointsOnLine h d CD ∧ distinctPointsOnLine e h EF ∧ between e g h  ∧
     b.sameSide d EF ∧ ¬(AB.intersectsLine CD) → ∠ e:g:b = ∠ g:h:d
theorem proposition_29''''' : ∀ (b d g h : Point) (AB CD EF : Line), distinctPointsOnLine
    g b AB ∧ distinctPointsOnLine h d CD ∧ distinctPointsOnLine g h EF ∧ b.sameSide d EF
    ∧ ¬(AB.intersectsLine CD) → ∠ b:g:h + ∠ g:h:d = ∟ + ∟

-- In any triangle, if one of the sides is produced, then the exterior angle equals the
    sum of the two interior and opposite angles, and the sum of the three interior angles
    of the triangle equals two right angles.
theorem proposition_32 : ∀ (a b c d : Point) (AB BC AC : Line), formTriangle a b c AB BC
    AC ∧ (between b c d) → ∠ a:c:d = ∠ c:a:b + ∠ a:b:c ∧ ∠ a:b:c + ∠ b:c:a + ∠ c:a:b =
    ∟ + ∟
-- In parallelogrammic areas the opposite sides and angles equal one another, and the
    diameter bisects the areas.
theorem proposition_34 : ∀ (a b c d : Point) (AB CD AC BD BC : Line), formParallelogram a
    b c d AB CD AC BD ∧ distinctPointsOnLine b c BC → |(a--b)| = |(c--d)| ∧ |(a--c)| = |(
    b--d)| ∧ ∠ a:b:d = ∠ a:c:d ∧ ∠ b:a:c  = ∠ c:d:b ∧ Triangle.area △ a:b:c = Triangle.
    area △ d:c:b
theorem proposition_34' : ∀ (a b c d : Point) (AB CD AC BD : Line), formParallelogram a b
    c d AB CD AC BD → |(a--b)| = |(c--d)| ∧ |(a--c)| = |(b--d)| ∧ ∠ a:b:d = ∠ a:c:d ∧ ∠
    b:a:c = ∠ c:d:b

--- Proof DSL ---
Your response must be a tactic proof in the LeanEuclid proof DSL. This DSL is built from
    the following tactics:
You need to formalize each key step (lemma) in the given proof and prove it by 'have <name
    > : <claim> := by <proof>'

1. euclid_intros
It introduces universally quantified variables and premises of the current goal into the
    proof context. No names required.

2. euclid_apply <rule> <args>
If <rule> has the form ∀ (<args> : Types) ... P -> Q, it instantiates <rule> with <args>,
    and attempts to prove premise P from the local proof context using the above axioms.
    If successful, propsition Q is added to the proof context.
usage examples :
  euclid_apply proposition_13 A B C D AB CD -- This proves that ∠ C:B:A + ∠ A:B:D = ∟ + ∟

If <rule> has the form ∀ (<args> : Types) ... P -> ∃ x . Q(x), this tactic instantiates <
    rule> with <args>, and attempts to prove premise P from the local proof context using
    the above axioms. If successful, object x and premise Q(x) are added to the proof
    context.
usage examples:
  euclid_apply extend_point L a b as c -- This creates a point c on line L such that b is
      between a and c.

3. euclid_assert <P>
It attempts to prove proposition <P> from the current proof context and the above axioms.

4. euclid_finish
It attempts to resolve the proof goal using the current proof context and the above axioms
    .

5. use <X>
This tactic proves an existentially quantified proposition by providing the witness <X>
    for the quantifier.

6. have <name> : <claim> := by <proof>
This tactic proves a lemma <claim>. The <name> is optional, and you need to use
    euclid_finish at the end of <proof>. If <claim> is already in premise, you can use
    directly use.
```

```
In case the <proof> only contains euclid_finish, you can directly use euclid_assert <claim
    > for simplicity.

--- Guidelines ---
1. Response Format: Present your proof within THREE angle brackets by <<<Your Proof>>>. Do
     not add any annotations/explanations in your proof. Do not use markdown syntax.
2. You should not use other imperative Lean tactics, such as 'rw' or 'simp'. You should
    only use the above declarative tactics.
--- End of Instruction ---
```

E. Theorem Statement Autoformalization

E.1. Successful Formalizations

As mentioned in the paper, some of the successful predictions proven correct by E3 were actually syntactic matches with their ground truth formalizations; these cases are not interesting, since existing syntactic evaluation techniques can identify these are correct. What is interesting are cases where the prediction is *not* a syntactic match with its ground truth. Even a small deviation (e.g., writing $\angle ABC$ rather than $\angle BAC$) may or may not affect the semantics of the result, depending on the context. It is these cases for which E3 offers an improvement over syntactic or manual evaluation methods. We highlight some such cases here.

Proposition 8, GPT-4V, 5-shot. Consider Proposition 8 of *Elements*, whose abridged version used for autoformalization of theorem statements reads as follows:

If two triangles have two sides equal to two sides, respectively, and also have the base equal to the base, then they will also have equal the angles encompassed by the equal straight-lines. Let ABC and DEF be two triangles having the two sides AB and AC equal to the two sides DE and DF, respectively. (That is) AB to DE, and AC to DF. Let them also have the base BC equal to the base EF. I say that the angle BAC is also equal to the angle EDF ... So the angle BAC will also coincide with angle EDF, and will be equal to it [C.N. 4]. Thus, if two triangles have two sides equal to two side, respectively, and have the base equal to the base, then they will also have equal the angles encompassed by the equal straight-lines

The formalization generated by GPT-4 from this text is as follows, alongside our ground truth formalization:

```
prediction :
    ∀ (a b c d e f : Point) (AB AC BC DE DF EF : Line),
    formTriangle a b c AB AC BC ∧ formTriangle d e f DE DF EF ∧
    |(a--b)| = |(d--e)| ∧ |(a--c)| = |(d--f)| ∧  |(b--c)| = |(e--f)| →
    (∠ a:b:c = ∠ d:e:f)

ground :
    ∀ (a b c d e f : Point) (AB BC AC DE EF DF : Line),
    formTriangle a b c AB BC AC ∧ formTriangle d e f DE EF DF ∧
    |(a--b)| = |(d--e)| ∧ |(a--c)| = |(d--f)| ∧ |(b--c)| = |(e--f)| →
    (∠ b:a:c) = (∠ e:d:f)
```

Syntactically, there are four differences between the two propositions: in each of the `formTriangle` clauses, we have swapped two sets of lines $BC \mapsto AC$ and $EF \mapsto DF$, and in the postcondition, we have also swapped points $b \mapsto a$ and $e \mapsto d$. This is a good example of a set of simple perturbations whose validity may not be immediately obvious to a human grader. However, E3 is easily able to prove that these propositions are indeed equivalent.

Proposition 10, GPT-4, 1-shot. Consider Proposition 10 of *Elements*:

"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... 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."

When formalizing this statement, one has to specify that point D lies halfway between A and B. Our ground truth formalization formalizes this by asserting that D lies between A and B, and the length $|AD|$ is equal to the length $|BD|$.

```
ground :
    ∀ (a b : Point) (AB : Line),
    distinctPointsOnLine a b AB →
    ∃ d : Point, between a d b ∧ |(a--d)| = |(d--b)|
```

However, there is another way to specify that D lies exactly halfway between A and B, as suggested by GPT-4 – namely, by specifying that, in addition to $|AD| = |BD|$, we have $|AD| + |BD| = |AB|$:

```
prediction:
    ∀ (a b : Point) (AB : Line),
    distinctPointsOnLine a b AB →
    ∃ d : Point, |(a--d)| = |(b--d)| ∧ |(d--b)| + |(a--d)| = |(a--b|
```

E3 is able to fill the gap between the two formulations.

Proposition 20, GPT-4, 1-shot. Consider Proposition 20 of *Elements*:

In any triangle, (the sum of) two sides taken together in any (possible way) is greater than the remaining (side). For let ABC be a triangle. I say that in triangle ABC (the sum of) two sides taken together in any (possible way) is greater than the remaining (side)... (So), (the sum of) BA and AC (is greater) than BC, (the sum of) AB and BC than AC, and (the sum of) BC and CA than AB. Thus, (the sum of) BA and AC is greater than BC. Similarly, we can show that (the sum of) AB and BC is also greater than CA, and (the sum of) BC and CA than AB. Thus, in any triangle, (the sum of) two sides taken together in any (possible way) is greater than the remaining (side)

In the text, Euclid explicitly mentions each combination of distinct line segments explicitly. However, when formalizing the theorem statement, it is not necessary to do so. All combinations are covered by the following concise formulation:

```
ground :
    ∀ (a b c : Point) (AB BC AC : Line),
    formTriangle a b c AB BC AC →
    |(b--a)| + |(a--c)| > |(b--c)|)
```

Since this simplification is not obvious to the model, it will instead attempt to enumerate each for the combinations mentioned:

```
prediction :
    ∀ (a b c : Point) (AB BC CA : Line),
    formTriangle a b c AB BC CA →
    (|(a--b)| + |(b--c)| > |(a--c)|) ∧
    (|(a--b)| + |(a--c)| > |(b--c)|) ∧
    (|(b--c)| + |(a--c)| > |(a--b)|)
```

Using only a syntactic similarity metric, one might judge the prediction poorly since it nearly doubles the character length of the ground truth formalization. However, E3 correctly identifies them as semantically equivalent.

E.2. False Negatives

As mentioned in the paper, E3 is occasionally unable to verify reasonable predictions. Based on our sample size, we estimate the false negative rate to be approximately 15%. Here, we share the false negatives we found while manually evaluating the results from GPT-4 and GPT-4V ($k = 5$) in Experiment #1.

Proposition 1, Both Models, 5-shot. Two of our false negatives were the same formalization of Proposition 1, whose text reads:

"To construct an equilateral triangle on a given finite straight-line. Let AB be the given finite straight-line... 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".

Our ground truth formalization is as follows:

```
ground:
    ∀ (a b : Point) (AB : Line), distinctPointsOnLine a b AB →
    ∃ c : Point, |(c--a)| = |(a--b)| ∧ |(c--b)| = |(a--b)|
```

The prediction which E3 is unable to prove equivalent is

```
prediction:
    ∀  (a b : Point) (AB : Line), distinctPointsOnLine a b AB →
    ∃  (c : Point) (AC BC : Line), formTriangle a b c AB BC AC ∧
        |(a-- b)| = |(b-- c)| ∧ |(a-- c)| = |(a-- b)|
```

During equivalence checking, E3 is unable to apply the construction rules required to form lines AC and BC.

Proposition 11, Text-based, 5-shot. Another false negative came from Proposition 11, which reads:

"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.

Thus, the straight-line CF has been drawn at right-angles to the given straight-line AB from the given point C on it."

Our ground truth formalization is as follows:

```
ground :
    ∀  (a b c : Point) (AB : Line), distinctPointsOnLine a b AB ∧  between a c b →
    ∃  f : Point, ¬(f.onLine AB) ∧  ∠ a:c:f = ∟
```

This formalization reflects the diagram in the souce material, in which the point C around which the right angle is constructed indeed lies between A and B. However, the only real requirement in the formalization is to ensure that there is a right angle against line AB.

The prediction which is rejected by E3 is as follows:

```
prediction:
    ∀ (a b c : Point) (AB : Line), distinctPointsOnLine a b AB ∧ c.onLine AB
    → ∃ (f : Point), (∠ a:c:f) = ∟ ∧ (∠ b:c:f) = ∟
```

Which captures the essence of the theorem. Both theorems are actually true, and are each reasonable formalizations of the given statement, but E3 is unable to prove their equivalence. In addition to revealing the limitations of the tool, this also reflects the difficulty in building a dataset which can anticipate all possible ways of formalizing a natural language theorem.

E.3. False Positive

As mentioned in the paper, we did not encounter indications of soundness bugs in our experiment results. However, we did encounter one instance of a proposition which was proven correct by E3, despite being an "incorrect" formalization of the given proposition. The instance is from the UniGeo/Congruent dataset, which amounts to proving that, with respect to the diagram in Fig. B, if $|TU| = |VW|$ and TU is parallel to VW, then the triangles formed by points TUV and TVW are congruent.

Our ground truth formalization is as follows:

```
ground :
    ∀ (T U V W : Point) (TU UV TV VW TW : Line),
    formTriangle T U V TU UV TV ∧ formTriangle T V W TV VW TW ∧
    U.opposingSides W TV ∧ |(T--U)| = |(V--W)| ∧¬ TU.intersectsLine VW →
    △ T:U:V).congruent (△ V:W:T)
```

One of the predictions generated by GPT-4v is given as follows:

```
prediction :
```

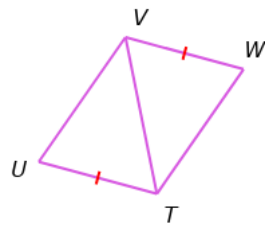


Figure B. Diagrammatic input for UniGeo proof/Congruent Thm02

```
    ∀ (U V W T : Point) (UV VT TW WT : Line),
    formQuadrilateral U V W T UV VT TW WT ∧
    |(T--U)| = |(V--W)| ∧ ¬(UV.intersectsLine WT) →
    (△ T:U:V).congruent (△ V:W:T)
```

In this case, the formalization does not quite line up with the text of the proposition; the statement assumes a quadrilateral rather than two triangles, and the premises refer to UV and WT being parallel rather than TU and VW, despite still assuming $TU = VW$. Of course, the statement is still true, and in fact E3 is able to prove they are equivalent; however, this is not quite a "faithful" formalization of the text.

E.4. Unsatisfiable Statements

E3 did not identify any unsatisfiable predictions during our final rounds of experiments, but we did encounter a few such cases during preliminary experiments and testing; we include them here for the interested reader, as well as to emphasize the importance of rigorous semantic validation.

Proposition 11, GPT-4, 5-shot. Consider the following:

```
prediction :
    ∀  (a b c : Point) (AB : Line),
    c.onLine AB →
    ∃  (CF : Line),  c.onLine CF ∧  ¬ (AB.intersectsLine CF) ∧  ∠  a:c:b = ∟
```

E3 correctly identifies this proposition as unsatisfiable, since, if c is on CF, and c is on AB, then obviously these lines must intersect.

Proposition 46, GPT-4, 5-shot. Consider the following:

```
prediction:
∀  (a b : Point) (AB : Line), distinctPointsOnLine a b AB →  ∃  (c d : Point) (AC BD :
    Line), formQuadrilateral a d c b AB BD AC BD ∧ |(a-- b)| = |(b-- c)| ∧  |(c-- d)| = |(
    d-- a)| ∧ ∠  a:b:c = ∟  ∧ ∠  b:c:d = ∟  ∧  ∠  c:d:a = ∟  ∧  ∠  d:a:b = ∟
```

E3 correctly identifies this proposition as unsatisfiable. The cause is that, in the application of `formQuadrilateral`, the line BD is given as an argument twice. However, `formQuadrilateral` requires that all its inputs are distinct objects.

Proposition 48, GPT-4, 5-shot. Consider the following prediction, which is stating the converse direction of the Pythagorean theorem:

```
prediction:
    ∀ (a b c : Point) (AB AC BC : Line),
    formTriangle a b c AB AC BC ∧
    |(b--c)| * |(b--c)| = |(a--b)| * |(a--b)| + |(a--c)| * |(a--c)| →
    ∠ a:b:c = ∟
```

Obviously, if $|BC|^2 = |AB|^2 + |AC|^2$, then BC must be the hypotenuse of the triangle, so the right angle cannot be around point B. E3 is able to identify this impossibility.

E.5. The Approximate Logical Equivalence Procedure

As shown in Fig. 3, the approximate equivalence checker attempts to quantify how "semantically close" one theroem statement is to another. The intention is to allow us to distinguish "close" formalizations from "junk" ones. We provide a more detailed description of how this procedure works.

We assume we are given a ground truth formula T_{gt} and a prediction T_{gt} such that the two formulas agree on the quantity and types of their bound variables. The goal is to find a unification ρ between bound variables in T_{gt} and T_{gt} with which we can prove the equivalence of the preconditions and postconditions of the two formulae. Obviously, the number of potential unifications is factorial in the number of variables, so we fix some upper bound n on the number of unifications we consider, and rank candidates using a string similarity heuristic.

Having fixed a unification ρ of variables in T_{pred} with variables in T_{gt}, let T'_{pred} be the result of renaming the variables in T_{pred} with variables in T_{gt} according to ρ. Assuming T'_{pred} and T_{gt} are both well-formed statements in E, we can extract the sets $\{p_i\}$ and $\{q_i\}$ containing the pre/post-conditions of T_{gt}, respectively, and likewise the set $\{r_i\}$ and $\{s_i\}$ containing the pre/postconditions of T'_{pred}. We then perform a four-step process: (1) Assuming $\bigwedge_i p_i$ try to prove each r_j (2) Assuming $\bigwedge_i r_i$, try to prove each p_j (3) Assuming $\bigwedge_i q_i$, try to prove each s_j (4) Assuming $\bigwedge_i s_i$, try to prove each q_j.

If steps (1) and (2) are both successful (meaning that the preconditions of the formulae were proven equivalent), then the preconditions are included as additional assumptions for steps (3) and (4). For each step, `E3` records how many obligations were able to be proved. The results are then aggregated for each of the n unifications of bound variables, which can then be evaluated by any number of metrics. For instance, Fig. 3 compares the formalization instances using the ratio of clauses solved across steps (1)–(4).

F. Proof Autoformalization

F.1. Quantitative Results

As mentioned in the paper, only two proofs from *Elements* were able to be formalized correctly out-of-the-box. For the remaining 43 proofs, we wished to quantify how "good" the proofs are by computing the degree of modification introduced during the repair process, as measured by the Levenshtein ratio. The results of this process for the (incomplete) proofs autoformalized by GPT-4 and GPT-4V are shown in Fig. C and Fig. D, respectively. Obviously, this is not a perfect metric – for instance, a tactic invoking `proposition_6` may only differ in a single character with one invoking `proposition_7`. However, we find that these kinds of major conceptual errors, where the model chooses an entirely irrelevant theorem or rule, are quite rare. The majority of modifications necessary to repair proofs (as shown by the exemplars below) are simply rearranging arguments to tactics or modifying one variant of a theorem for another (e.g., `proposition_9` versus the variant `proposition_9`′).

F.2. Correctly Formalized Proofs from *Elements*

The same two proofs – Proposition 1 and Proposition 17 – were correctly formalized out-of-the-box by GPT-4 and GPT4-V. Both of these are relatively simple proofs. Nevertheless, they serve as good exemplars of how LeanEuclid's proof language supports simple, elegant and faithful formalizations of the natural language proof.

Proposition 1, GPT-4V. Natural language proof:

"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."

Autoformalized proof:

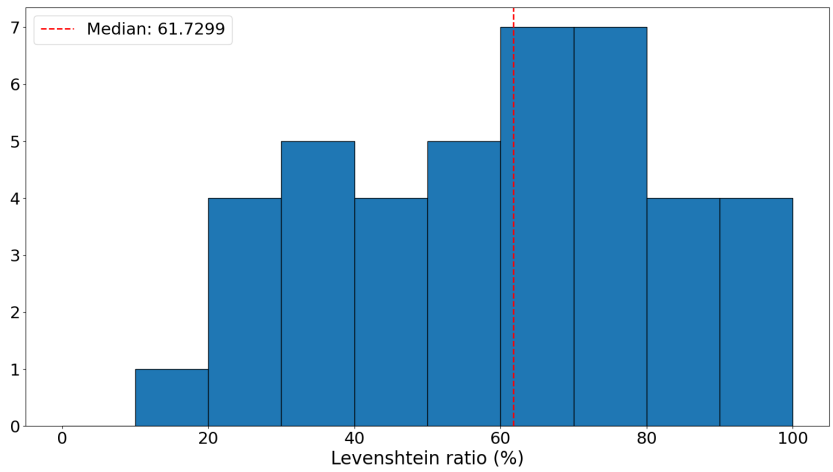


Figure C. Levenshtein ratio between autoformalized and repaired *Elements* proofs generated by GPT-4 (5-shot).

```
theorem prop1_prediction : ∀ (a b : Point) (AB : Line), distinctPointsOnLine a b AB → ∃ c
     : Point, |(c--a)| = |(a--b)| ∧ |(c--b)| = |(a--b)|:=
by  euclid_intros
    euclid_apply circle_from_points a b as BCD
    euclid_apply circle_from_points b a as ACE
    euclid_apply intersection_circles BCD ACE as c
    euclid_apply line_from_points c a as CA
    euclid_apply line_from_points c b as CB
    euclid_apply point_on_circle_if a b c BCD
    euclid_apply point_on_circle_if b a c ACE
    use c
    euclid_finish
```

Proposition 17, GPT-4V. Natural language proof:

"For any triangle, (the sum of) two angles taken together in any (possible way) is less than two right-angles. Let ABC be a triangle. I say that (the sum of) two angles of triangle ABC taken together in any (possible way) is less than two right-angles. For let BC have been produced to D. And since the angle ACD is external to triangle ABC, it is greater than the internal and opposite angle ABC [Prop. 1.16].

Let ACB have been added to both. Thus, the (sum of the angles) ACD and ACB is greater than the (sum of the angles) ABC and BCA. But, (the sum of) ACD and ACB is equal to two right-angles [Prop. 1.13]. Thus, (the sum of) ABC and BCA is less than two right-angles. Similarly, we can show that (the sum of) BAC and ACB is also less than two right-angles, and further (that the sum of) CAB and ABC (is less than two right-angles). Thus, for any triangle, (the sum of) two angles taken together in any (possible way) is less than two right-angles. (Which is) the very thing it was required to show."

Autoformalized proof:

```
theorem prop_17_prediction : ∀ (a b c : Point) (AB BC AC : Line), formTriangle a b c AB BC
     AC → (∠ a:b:c) + (∠ b:c:a) < ∟ + ∟:=
by  euclid_intros
    euclid_apply extend_point BC b c as d
    euclid_apply proposition_16 a b c d AB BC AC
    euclid_apply proposition_13 a c b d AC BC
    euclid_finish
```

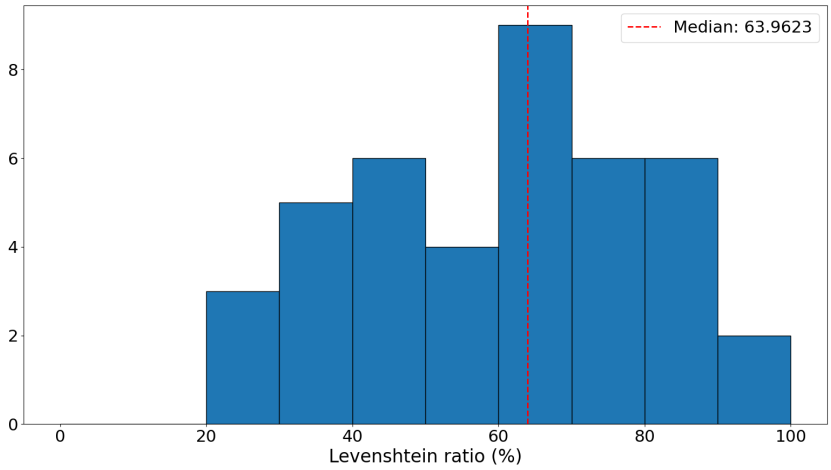


Figure D. Levenshtein ratio between autoformalized and repaired *Elements* proofs generated by GPT-4V (5-shot).

F.3. Partiallly Correct Autoformalized Proofs from *Elements*

Here, we showcase some attempts at autoformalizing proofs from *Elements* which, despite being incorrect, require only minor modifications to be repaired. These exemplars demonstrate that, in many cases, the steps required to repair the proof do not require creative insight. Moreover, we believe this process is quite amenable to automation.

Proposition 3, GPT-4. Natural language proof:

"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."

Autoformalized Proof:

```
def prop3_prediction :  ∀ (a b c0 c1 : Point) (AB C : Line),
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c0 c0 C  ∧ a ≠ c0 ∧
    |(a--b)| > |(c0--c1)| →
        ∃ e, between a e b ∧ |(a--e)| = |(c0--c1)| :=
by
    euclid_intros
    euclid_apply (proposition_2 a c0 c1 C) as d
    euclid_apply (circle_from_points a d) as DEF
    euclid_apply (intersection_circle_line DEF AB) as (e, f)
    euclid_apply (point_on_circle_onlyif a d e DEF)
    euclid_apply between_if a e b
    use e
    euclid_finish
```

To prove the precondition for `between_if`, we need the fact `between a e b`, which is not provable from the preceding tactics. To repair this proof, we can use a stronger version of `intersection_circle_line` on Line 9.

Repaired Proof:

```
def prop3_repaired :  ∀ (a b c0 c1 : Point) (AB C : Line),
```

```
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine c0 c0 C  ∧ a ≠ c0 ∧
    |(a--b)| > |(c0--c1)| →
        ∃ e, between a e b ∧ |(a--e)| = |(c0--c1)| :=
by
    euclid_intros
    euclid_apply (proposition_2 a c0 c1 C) as d
    euclid_apply (circle_from_points a d) as DEF
    euclid_apply (intersection_circle_line_between_points DEF AB a b) as e
    euclid_apply (point_on_circle_onlyif a d e DEF)
    euclid_apply between_if a e b
    use e
    euclid_finish
```

Proposition 5, GPT-4. Natural language proof:

"For isosceles triangles, the angles at the base are equal to one another, and if the equal sides are produced then the angles under the base will be equal to one another. Let ABC be an isosceles triangle having the side AB equal to the side AC, and let the straight-lines BD and CE have been produced in a straight-line with AB and AC (respectively) [Post. 2]. I say that the angle ABC is equal to ACB, and (angle) CBD to BCE. For let the point F have been taken at random on BD, and let AG have been cut off from the greater AE, equal to the lesser AF [Prop. 1.3]. Also, let the straight-lines FC and GB have been joined [Post. 1]. In fact, since AF is equal to AG, and AB to AC, the two (straight-lines) FA, AC are equal to the two (straight-lines) GA, AB, respectively. They also encompass a common angle, FAG. Thus, the base FC is equal to the base GB, and the triangle AFC will be equal to the triangle AGB, and the remaining angles subtended by the equal sides will be equal to the corresponding remaining angles [Prop. 1.4]. (That is) ACF to ABG, and AFC to AGB. And since the whole of AF is equal to the whole of AG, within which AB is equal to AC, the remainder BF is thus equal to the remainder CG [C.N. 3]. But FC was also shown (to be) equal to GB. So the two (straight-lines) BF, FC are equal to the two (straight-lines) CG, GB, respectively, and the angle BFC (is) equal to the angle CGB, and the base BC is common to them. Thus, the triangle BFC will be equal to the triangle CGB, and the remaining angles subtended by the equal sides will be equal to the corresponding remaining angles [Prop. 1.4]. Thus, FBC is equal to GCB, and BCF to CBG. Therefore, since the whole angle ABG was shown (to be) equal to the whole angle ACF, within which CBG is equal to BCF, the remainder ABC is thus equal to the remainder ACB [C.N. 3]. And they are at the base of triangle ABC. And FBC was also shown (to be) equal to GCB. And they are under the base. Thus, for isosceles triangles, the angles at the base are equal to one another, and if the equal sides are produced then the angles under the base will be equal to one another. (Which is) the very thing it was required to show."

Autoformalized Proof:

```
def prop5_prediction : ∀ (a b c d e : Point) (AB BC AC : Line), formTriangle a b c AB BC
    AC ∧ |(a--b)| = |(a--c)| ∧ between a b d ∧ between a c e → ∠a:b:c = ∠a:c:b ∧ ∠c:b:d
    = ∠b:c:e :=
by  euclid_intros
    euclid_apply (exists_point_between_points_on_line AB b d) as f
    euclid_apply (proposition_3 e a a f AC AB) as g
    euclid_apply (line_from_points f c) as FC
    euclid_apply (line_from_points g b) as GB
    euclid_apply proposition_4 f a c g a b FC AC GB AB AC AB
    euclid_apply (between_if f b a)
    euclid_apply (between_if g c a)
    euclid_apply (proposition_4 b f c c g b AB FC BC GB BC AB)
    use f, g, FC, GB
    euclid_finish
```

In order to prove the preconditions for `proposition_3`, we need to know $|ba| < |ce|$, which can be obtained via a stronger version of `exists_point_between_points_on_line` on Line 2. We also need to rearrange some of the arguments on in the invocations of `proposition_3` and `proposition_4`. Finally, we can remove the invocation of tactic `use` entirely, since it is inapplicable in this proof context.

Repaired Proof:

```
def prop5_repaired : ∀ (a b c d e : Point) (AB BC AC : Line), formTriangle a b c AB BC AC
    ∧ |(a--b)| = |(a--c)| ∧ between a b d ∧ between a c e → ∠a:b:c = ∠a:c:b ∧ ∠c:b:d =
    ∠b:c:e  :=
by  euclid_intros
    euclid_apply (point_between_points_shorter_than AB b d (c--e)) as f
    euclid_apply (proposition_3 a e f a AC AB) as g
    euclid_apply (line_from_points f c) as FC
    euclid_apply (line_from_points b g) as GB
    euclid_apply proposition_4 a f c a g b AB FC AC AC GB AB
    euclid_apply (between_if f b a)
    euclid_apply (between_if g c a)
    euclid_apply (proposition_4 f b c g c b AB BC FC AC BC GB)
    euclid_finish
```

Proposition 7, GPT-4. Natural language proof:

On the same straight-line, two other straight-lines equal, respectively, to two (given) straight-lines (which meet) cannot be constructed (meeting) at a different point on the same side (of the straight-line), but having the same ends as the given straight-lines. For, if possible, let the two straight-lines AC, CB, equal to two other straight-lines AD, DB, respectively, have been constructed on the same straight-line AB, meeting at different points, C and D, on the same side (of AB), and having the same ends (on AB). So CA is equal to DA, having the same end A as it, and CB is equal to DB, having the same end B as it. And let CD have been joined [Post. 1]. Therefore, since AC is equal to AD, the angle ACD is also equal to angle ADC [Prop. 1.5]. Thus, ADC (is) greater than DCB [C.N. 5]. Thus, CDB is much greater than DCB [C.N. 5]. Again, since CB is equal to DB, the angle CDB is also equal to angle DCB [Prop. 1.5]. But it was shown that the former (angle) is also much greater (than the latter). The very thing is impossible. Thus, on the same straight-line, two other straight-lines equal, respectively, to two (given) straight-lines (which meet) cannot be constructed (meeting) at a different point on the same side (of the straight-line), but having the same ends as the given straight-lines. (Which is) the very thing it was required to show.

Autoformalized Proof:

```
def prop7_prediction : ∀ (a b c d : Point) (AB AC CB AD DB CD : Line),
    distinctPointsOnLine a b AB ∧ distinctPointsOnLine a c AC ∧ distinctPointsOnLine c b
    CB ∧ distinctPointsOnLine a d AD ∧ distinctPointsOnLine d b DB ∧ Point.sameSide c d AB
     ∧ distinctPointsOnLine c d CD ∧ Point.sameSide a b CD ∧ Point.sameSide d b AC ∧ |(a
    --c)| = |(a--d)| ∧ |(c--b)| = |(d--b)| → False :=
by  euclid_intros
    euclid_apply (line_from_points c d) as CD
    euclid_apply proposition_5 c a d AC AD CD
    euclid_apply proposition_5 c b d CB DB CD
    by_cases (∠c:a:d) > (∠ d:c:b)
    . euclid_apply proposition_5' c a d b AC AD CD AB
      euclid_finish
    . by_cases (∠ c:b:d) > (∠ d:c:b)
      . euclid_apply proposition_5' c b d a CB DB CD AB
        euclid_finish
      . euclid_finish
```

The rule `proposition_5` takes five points and three lines as argument, whereas the variant `proposition_5'` takes three of each. It is easiest to fix the fourth and fifth tactics by instead invoking the latter theorem (and tweaking the arguments). The `euclid_apply` within the case analysis also need repairing, since they are using too many arguments. These tactics are actually redundant, since `euclid_finish` can complete the proof. Nevertheless, simply ommitting the last point and line from each tactic is sufficient to make the proof go through.

Repaired Proof:

```
def prop7_repaired : ∀ (a b c d : Point) (AB AC CB AD DB CD : Line), distinctPointsOnLine
    a b AB ∧ distinctPointsOnLine a c AC ∧ distinctPointsOnLine c b CB ∧
    distinctPointsOnLine a d AD ∧ distinctPointsOnLine d b DB ∧ Point.sameSide c d AB ∧
    distinctPointsOnLine c d CD ∧ Point.sameSide a b CD ∧ Point.sameSide d b AC ∧ |(a--c)|
     = |(a--d)| ∧ |(c--b)| = |(d--b)| → False :=
```

```
by
    euclid_intros
    euclid_apply line_from_points c d as CD'
    euclid_apply proposition_5' a c d AC CD AD
    euclid_apply proposition_5' b c d CB CD DB
    by_cases (∠  c:a:d) > (∠  d:c:b)
    . euclid_apply proposition_5' c a d AC AD CD
      euclid_finish
    . by_cases (∠  c:b:d) > (∠  d:c:b)
        . euclid_apply proposition_5' c b d CB DB CD
          euclid_finish
        . euclid_finish
```

Proposition 18, GPT-4V. Natural language proof:

"In any triangle, the greater side subtends the greater angle. For let ABC be a triangle having side AC greater than AB. I say that angle ABC is also greater than BCA. For since AC is greater than AB, let AD be made equal to AB [Prop. 1.3], and let BD have been joined. And since angle ADB is external to triangle BCD, it is greater than the internal and opposite (angle) DCB [Prop. 1.16]. But ADB (is) equal to ABD, since side AB is also equal to side AD [Prop. 1.5]. Thus, ABD is also greater than ACB. Thus, ABC is much greater than ACB. Thus, in any triangle, the greater side subtends the greater angle. (Which is) the very thing it was required to show."

Autoformalized Proof:

```
def prop18_prediction : ∀ (a b c : Point) (AB BC AC : Line), formTriangle a b c AB BC AC
    ∧ (|(a --c)| > |(a--b)|) → (∠ a:b:c > ∠ b:c:a):=
by  euclid_intros
    euclid_apply proposition_3 a c a b AC AB as d
    euclid_apply line_from_points b d as BD
    euclid_apply proposition_16 a b d c AB BD AC
    euclid_apply proposition_5 a b d AB BD as H1
    euclid_apply proposition_6 a b d AB BD AC
    euclid_finish
```

The second tactic does not provide the right number of arguments to `proposition_3`, but it does for `proposition_3'`. The preconditions of the invocation of `proposition_16` are not provable from the preceding steps, but we can permute points and a, b, c to recover a valid tactic. The invocation of `proposition_5` has the same problem as `proposition_3` , and is replaced with `proposition_5'`. Also, the keyword "`as`" can only be used for constructive theorems, which `proposition_5` is not. We can simply drop "`as H1`" and the proof goes through.

Repaired Proof:

```
def prop18_repaired : ∀ (a b c : Point) (AB BC AC : Line), formTriangle a b c AB BC AC ∧
    (|(a --c)| > |(a--b)|) → ((∠ a:b:c) > ∠ b:c:a):=
    euclid_intros
    euclid_apply proposition_3' a c a b AC AB as d
    euclid_apply line_from_points b d as BD
    euclid_apply proposition_16 b c d a BC AC BD
    euclid_apply proposition_5' a b d AB BD AC
    euclid_apply proposition_6 a b d AB BD AC
    euclid_finish
```

Proposition 34, GPT-4V. Natural language proof:

"In parallelogrammic figures the opposite sides and angles are equal to one another, and a diagonal cuts them in half. Let $ACDB$ be a parallelogrammic figure, and BC its diagonal. I say that for parallelogram $ACDB$, the opposite sides and angles are equal to one another, and the diagonal BC cuts it in half. For since AB is parallel to CD, and the straight-line BC has fallen across them, the alternate angles ABC and BCD are equal to one another [Prop. 1.29]. Again, since AC is parallel to BD, and BC has fallen across them, the alternate angles ACB and CBD are equal to one another [Prop. 1.29].

So ABC and BCD are two triangles having the two angles ABC and BCA equal to the two (angles) BCD and CBD, respectively, and one side equal to one side—the (one) by the equal angles and common to them, (namely) BC. Thus, they will also have the remaining sides equal to the corresponding remaining (sides), and the remaining angle (equal) to the remaining angle [Prop. 1.26]. Thus, side AB is equal to CD, and AC to BD. Furthermore, angle BAC is equal to CDB. And since angle ABC is equal to BCD, and CBD to ACB, the whole (angle) ABD is thus equal to the whole (angle) ACD. And BAC was also shown (to be) equal to CDB. Thus, in parallelogrammic figures the opposite sides and angles are equal to one another. And, I also say that a diagonal cuts them in half. For since AB is equal to CD, and BC (is) common, the two (straight-lines) AB, BC are equal to the two (straight-lines) DC, CB, respectively. And angle ABC is equal to angle BCD. Thus, the base AC (is) also equal to DB, and triangle ABC is equal to triangle BCD [Prop. 1.4]. Thus, the diagonal BC cuts the parallelogram $ACDB$ in half. (Which is) the very thing it was required to show."

Autoformalized Proof:

```
def prop34_prediction : ∀ (a b c d : Point) (AB CD AC BD BC : Line), formParallelogram a b
     c d AB CD AC BD ∧ distinctPointsOnLine b c BC →
    |(a--b)| = |(c--d)| ∧ |(a--c)| = |(b--d)| ∧
    ∠ a:b:d = ∠ a:c:d ∧ ∠ b:a:c = ∠ c:d:b ∧
    Triangle.area △ a:b:c = Triangle.area △ d:c:b :=
by  euclid_intros
    euclid_apply proposition_29 a b c d AB CD AC BD
    euclid_apply proposition_29 c d a b CD AB BD AC
    euclid_apply proposition_26 a b c d a c b d AB BC AC BD AB CD
    euclid_apply proposition_4 a b c d c b AB BC AC BD CD BC
    euclid_finish
```

Similar to previous cases, the first invocation of `proposition_29` on is not well-formed, since `proposition_29` requires more arguments than the model has supplied. However, the variant `proposition_29'''` accepts the number and quantity of variables provided, so we can swap the rule and permute its arguments to make the tactic go through. The next invocation is repaired in the same fashion. Too few arguments are given to `proposition_265`, but there are no variants of this theorem, so we must infer which arguments can be supplied from the proof context. The preconditions of the tactic using `proposition_4` cannot be proved as-is. As a matter of fact, this tactic is unnecessary, since, the proof can be completed already by `euclid_finish`. However, if we want to repair it, we can simply permute a few of its arguments.

Repaired Proof:

```
def prop34_prediction :  ∀ (a b c d : Point) (AB CD AC BD BC : Line), formParallelogram a
    b c d AB CD AC BD ∧ distinctPointsOnLine b c BC →
    |(a--b)| = |(c--d)| ∧ |(a--c)| = |(b--d)| ∧
    ∠ a:b:d = ∠ a:c:d ∧ ∠ b:a:c = ∠ c:d:b ∧
    Triangle.area △ a:b:c = Triangle.area △ d:c:b :=
by
    euclid_intros
    euclid_apply proposition_29''' a d b c AB CD BC
    euclid_apply proposition_29''' a d c b AC BD BC
    euclid_apply proposition_26 a b c d c b AB BC AC CD BC BD
    euclid_apply (proposition_4 b a c c d b AB AC BC CD BD BC)
    euclid_finish
```

F.4. Autoformalized Proofs from UniGeo

Here, we showcase two examples of correctly autoformalized proofs from the UniGeo dataset.

Congruent/Thm07, GPT-4V, 5-shot. The diagrammatic input for this theorem is shown in Fig. E. The theorem asserts that if $|WX| = |ZY|$ and WX is parallel to ZY, then triangle WYZ is congruent to triangle YWX. The UniGeo proof

makes the following inferences:

$$
\begin{aligned}
& WX \parallel XY \wedge |WX| = |YZ| \\
\therefore\ & \angle WYZ = \angle XWY \\
& |WY| = |WY| \\
\therefore & \triangle WYZ \cong \triangle YWX
\end{aligned}
$$

GPT-4V correctly translates this semiformal proof to LeanEuclid as follows:

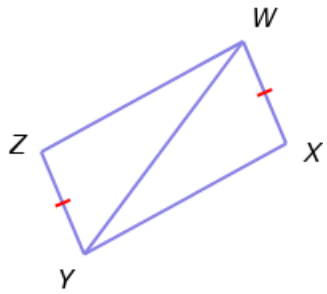


Figure E. Diagrammatic input for UniGeo/Congruent/Thm07

```
theorem prediction_thm07:
     ∀ (W X Y Z : Point) (WX XY WY YZ WZ : Line),
    formTriangle W X Y WX XY WY  ∧
    formTriangle W Y Z WY YZ WZ  ∧
    X.opposingSides Z WY  ∧ |(W --X)| = |(Y --Z)|  ∧
     ¬ WX.intersectsLine YZ  → ( △ W:Y:Z).congruent ( △ Y:W:X):=
by
    euclid_intros
    have :  ∠W:Y:Z =  ∠X:W:Y := by
        euclid_apply proposition_29''' X Z W Y WX YZ WY
        euclid_finish
    have : |(W --Y)| = |(W --Y)| := by euclid_finish
    have :  (△W:Y:Z).congruent  △Y:W:X := by euclid_finish
    euclid_finish
```

Parallel/Thm06, GPT-4V, 5-shot. The diagrammatic input for this proposition is shown in Fig. F. The theorem asserts that if $\angle STW$ and $\angle TWV$ are supplementary, then SU is parallel to VX. The UniGeo proof makes the following inferences:

$$
\begin{aligned}
& \angle STW \text{ and } \angle TWV \text{ are supplementary} \\
\therefore\ & \angle STW + \angle TWV = 180^\circ \\
\therefore\ & \angle RTS + \angle STW = 180^\circ \\
\therefore\ & \angle STW + \angle TWV = \angle RTS + \angle STW \\
\therefore\ & \angle TWV = \angle RTS \\
\therefore\ & SU \parallel VX
\end{aligned}
$$

GPT-4V correctly translates this semiformal proof to LeanEuclid as follows:

```
theorem prediction_thm06:
     ∀ (S U V X R Y T W : Point) (SU VX RY : Line), distinctPointsOnLine S U SU  ∧
    distinctPointsOnLine V X VX  ∧
    distinctPointsOnLine R Y RY  ∧
    twoLinesIntersectAtPoint SU RY T  ∧
    between S T U  ∧ between R T W  ∧
    twoLinesIntersectAtPoint VX RY W  ∧
```

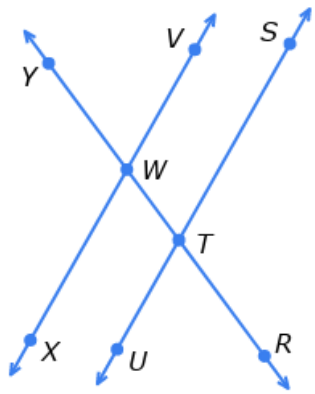


Figure F. Diagrammatic input for UniGeo/Parallel/Thm06

```
    between V W X  ∧ between T W Y  ∧
    V.sameSide S RY  ∧ U.sameSide X RY  ∧
     ∠ S:T:W +  ∠ T:W:V =  ∟ +  ∟  →  ¬ VX.intersectsLine SU:=
by
  euclid_intros
  have :  ∠ R:T:S +  ∠ S:T:W  = ∟+∟:=
  by
    euclid_apply proposition_13 S T R W SU RY
    euclid_finish
  have :  ∠S:T:W + ∠T:W:V = ∠ R:T:S +  ∠ S:T:W := by euclid_finish
  have :  ∠T:W:V =  ∠ R:T:S  := by euclid_finish
  have :  ¬ VX.intersectsLine SU := by
    euclid_apply proposition_15 R W S U T RY SU
    euclid_apply proposition_27' U V T W SU VX RY
    euclid_finish
  euclid_finish
```