EuclidZ3 – a Computational Proof-Checker for the Language E:

Possible Backend for Interactive Geometric Proof Environments

Kelvin J. Rojas[*]

Carnegie Mellon University

krojas@andrew.cmu.edu

Abstract

This paper is an undergraduate thesis intended as partial fulfillment of a Bachelors of Science degree in Logic and Computation at Carnegie Mellon University. As mentioned in "A Formal System for Euclid's Elements," [Avigad, Dean, Mumma], "it is possible to design a computational proof checker based on [the language E] that takes as input, proofs that look like the ones in the Elements and verifies their correctness against the rules of [language E]." The goal of this thesis project was to implement such a program. The result, EuclidZ3, was implemented in the python programming language using Z3, an automated theorem proving technology by Microsoft [de Moura, Bjørner], as a backend. The program is faithful to the language E's construction, metric, and inference rules. With EuclidZ3 users can construct linear Euclidean proofs and check the validity of construction steps and demonstration steps. EuclidZ3 has no parsing capabilities nor supports suppositional reasoning. However, EuclidZ3 should continue to evolve and the design of the program makes it extensible. Possible applications of the program as a backend for interactive geometric proof making in geometry tutoring software is also discussed.

I. Introduction

Euclid's *Elements* is a collection of geometric definitions, postulates, propositions, and proofs of those propositions. Starting from simple to complex, from demonstrations on how to construct a line to proofs of the pythagorean theorem and its converse, Euclid's *Elements* is a remarkable human achievement.

For millenia, proofs like those presented in the *Elements* represented mathematical rigor. The propositions presented were studied and used without problem. However, in the 19th century new developments in mathematics, led by David Hilbert, rose the bar on mathematical rigor; the sentiment was that appropriate logical analysis of geometric inference should be cast in terms of axioms and rules of inference. Indeed, Euclidean proofs come close to this ideal but it is their use of diagrams that presented an obvious gap in reasoning.

Consider proofs in the *Elements*. All Euclidean proofs maintain a diagram, i.e. configurations of geometric objects like points, lines, etc. These objects are constructed according to various rules and then inferences are made about those objects until some new proposition is shown to follow. For the most part, these inferences are justified by propositions and definitions previously justified.

However, to understand Euclid's *Elements* one requires an intuitive understanding of the diagrams and the implicit inferences Euclid makes about them; some steps implicitly require reading properties from a diagram. For a more detailed description of the shortcomings of Euclidean diagrammatic inference and its significance in the history of mathematics see [Avigad, Dean, Mumma].

It is precisely this reliance on intuitive di-

[*]First, I would like to thank my family and friends, especially those who edit my engrish and cope with my eccentricities: Mom, Allan, Rebecca, Samantha, Catala. I'm indebted also to Dr. Jeremy Avigad, Joel Smith, Dr. Cavalier, Mr. Steve Macissac, and Dr. Peter Spirtes for their boundless patience and guidance.

agrammatic reasoning that offended 19th century mathematicians. If some argument is to deliver mathematical certainty then it should not depend on imprecise drawings, instead certainty should be established by sets of sentences where a sentence is either a premise or derived from some sound rule of inference.

There has been attempts to formalize the *Elements*, most notably are Pasch (1882), Peano (1889), Hilbert (1899), and Tarski (1959). These systems are fully axiomatized and show all the same results as the *Elements*. Yet, proofs in these languages look little like Euclidean proofs.

The language E by [Avigad, Dean, Mumma] is more faithful to the style of Euclidean proofs because it precisely defines the diagrammatic reasoning used in Euclid's *Elements*. The system was shown to be sound and complete and is based on a simple set of inference rules, construction rules, and diagrammatic relations. More details of this language will be discussed in a later section.

[Avigad, Dean, Mumma] mentions the possibility of designing a computational proof checker "based on E that takes, as input, proofs that look like the ones in the *Elements* and verifies their correctness against the rules of the system." The goal of this thesis project is to implement a computational proof checker for the laguage E based on Z3, an automated theorem prover developed by microsoft [de Moura, Bjørner]. EuclidZ3 has been built in the python programming language. In what follows, we will talk about some of the features of the language E, some details about EuclidZ3's implementation, and discuss possible extensions/applications for EuclidZ3.

II. Language E

This section will gloss over details of this language. For a complete listing of the various axioms, definitions, and the rules of inference of the language E, see [Avigad, Dean, Mumma].

In mathematics and computer science a formal language is a set of strings of symbols that may be constrained by rules that are specific to it. The language E is a six-sorted language with sorts for lines, points, circles, angles, areas, and segments. It has various rules, called axioms, that apply to these sorts. These axioms spell out precisely what inferences can be "read off" from the diagram. The language has a set of relations:

- on(a, L): point a is on line L
- same-side(a, b, L): points a and b are on the same side of line L
- between(a, b, c): points a, b, and c are distinct and collinear, and b is between a and c
- on(a,): point a is on circle
- inside(a,): point a is inside circle
- center(a,): point a is the center of circle
- intersects(L , M): line L and M intersect
- intersects(L ,): line L intersects circle
- intersects(,): circles and intersect
- segment(a, b): the length of the line segment from a to b, written ab
- angle(a, b, c): the magnitude of the angle abc, written abc
- area(a, b, c): the area of triangle abc, written abc

The language also has addition, equality, and the less-than relationship that apply to the magnitude sorts, i.e. angles, area, and segments. It also has the constants 0 and "right-angle."

Proofs in E

For the purposes of this project, it's important to understand what proofs are like in E. Theorems in E, the objects one "proves," have the following form:

$$\forall a, L, \alpha (\phi(a, L, \alpha) \rightarrow \exists b, M, \beta \psi(a, b, L, M, \alpha, \beta)) \tag{1}$$

In English this reads as: Given a diagram consisting of some points, a, some lines, L, and some circles, α, satisfying assertions ϕ, one can construct points b, lines M, and circles β, such that the resulting diagram satisfies assertions ψ.

Theorems have assumed objects and assumed properties of those objects. This is represented as the left-hand side of the implication above. Theorems also have desired objects and desired properties/conclusions. This is on the right-hand side. Proofs of these theorems are sequences of steps that transform the diagram through construction of geometric objects or demonstrations of properties of those objects. Success in proving a theorem occurs when we have constructed the required objects and shown that those objects have the desired properties.

To summarize, with no small amount of handwaving, there are two types of steps in a Euclidean proof: construction steps, which introduce new objects into the diagram, and deduction/demonstration steps, which infer facts about objects that have already been introduced. Proofs in E, like in the *Elements*, are largely linear, one step follows the previous, but occasionally the proof can be broken down by cases or a proof by contradiction.

Construction steps are constrained by various pre-conditions and assert various properties to the proofs as post-conditions to successful construction. For example, a construction step like:

```
Let a be a point on L between b and c
```

requires that: b is on L, c is on L, b = c. After construction, the following conclusions are added to the proof: a is on L, a is between b and c. Construction steps assert the existence of new objects with properties asserted. For a complete list see [Avigad, Dean, Mumma].

In demonstration steps we apply various rules of inference to the proof. In the language E there are four types of inference rules: diagrammatic, metric, diagram-metric, and superposition. For the sake of this paper, we can understand these as lists of axioms that allow us to infer, e.g. diagrammatic assertions from the diagrammatic information currently available to us in a proof. Again, for more details see [Avigad, Dean, Mumma].

Demonstration steps do not assert the existence of new objects. Instead they assert properties about existing objects that are direct consequences of the objects and properties already in play.

III. Automated Theorem Proving component

The purpose of EuclidZ3 is to check proofs in the language E. At the heart of checking proofs is what we'll call the proof engine.

The proof engine is given expressions in the language E. It then checks that those expressions are satisfiable given the various assumptions and rules of a given formal language. The proof engine is a kind of blackbox. It is an automated theorem prover that can determine if expressions asserted in the language are direct consequences of the objects and properties in a given context.

EuclidZ3's proof engine is based on an automated theorem proving technology called Z3. It was developed by Microsoft. Z3 is a satisfiability modulo theories solver – SMT solver for short. SMT solvers combine a variety of decision procedures for the provability of universal sentences modulo the combination of disjoint theories whose universal fragments are decidable. For purposes of this paper, this amounts to determining the validity of sentences in the language E given various constraints, e.g. the axioms of the language, any assumptions asserted to the solver.

SMT solvers are particularly good at doing linear arithmetic with real numbers. This is ideal for our proof engine because the metric inferences are of the Real sort, e.g. the sum of two angles is less than some other angle. However, SMT solvers don't usually have decision procedures for sets of consequences of arbitrary universal axioms like the diagrammatic axioms of the language E.

However, Z3 is special because it has heuristic methods for instantiating quantifiers. This means that Z3 should be capable of verifying sentences of the language E.

IV. EuclidZ3

This section introduces EuclidZ3.

Recall that all Euclidean proofs maintain a diagram, i.e. configurations of geometric objects like points, lines, etc. These objects are constructed according to various rules and then inferences are made about those objects until some new proposition is shown to follow. EuclidZ3 parallels this. In essence, EuclidZ3 is a proof building program.

The user can construct geometric objects according to rules, assert relationships between those objects, and most importantly check inferences they make about those objects and their properties.

High level overview

In essence, the idea of a proof-checker is to validate steps in a proof. Validating means checking for inconsistencies. For example, if in my proof I assert that P and Q are distinct points then asserting that P equals Q is an inconsistency because the statement "P equals Q" contradicts my assumptions that P and Q are distinct.

As the user constructs new objects or asserts new theorems, EuclidZ3 checks if the steps they take cause a contradiction. Any contradiction that results from a user's activity either contradicts the axioms of the language E or some assumption they have made.

To achieve this, EuclidZ3 uses the program Z3 in the background to check assertions against the rules of the language E.

Implementation details

This section will discuss some of the object oriented design involved in the implementation. Whenever a word like Point is capitalized it refers to a class of objects in the software. If it is not capitalized then it refers to the language E. For example, the LanguageE is a software object but the language E is a formal system for Euclidean geometry.

The EuclidZ3 system is made of two main classes and one abstract class, the LanguageE, Proof, and ConstructableObject. Their relationships, fields, and methods are shown in the object model in the appendix A.

The main class that users will interact with is the Proof class. Using EuclidZ3 is analogous to making proofs in the language E. So, there are construction steps and demonstration steps. The methods of the Proof class reflect this. These are construct(ConstructableObject), hence(ExprRef), assert(Proof), and QED().

To make a new proof object:

```
pc = Proof();
```

Constructing Objects in a Proof

To construct a geometric object in the proof, first the user must configure an instance of a ConstructableObject. There are three concrete implementations of ConstructableObjects, these are Point, Line, and Circle. Each of these objects have certain constructions that are available to them. In the language E, constructions are akin to theorems that you can always apply, given that the construction's preconditions are met.

Here is an example of how to create a line between two points:

```
a = Point("a")
b = Point("b")
L = Line("L")
L.through(a,b)
```

The objects can then be passed to a particular proof:

```
pc.construct(a)
pc.construct(b)
pc.construct(L)
```

When the user enters this construction command, construct(L), the Proof object checks that the prerequisites required by the language E are consequences of the facts already asserted to the proof engine, creates the new objects, and asserts their properties. In the case of constructing L, the prerequisites are that a≠b and the properties asserted in the proof are: a is on L, b is on L. Note that, both

Points a and b are constructed in the proof first. Otherwise, the requirements for constructing L would not be met. Again, a full list of construction requirements and conclusions are available in [Avigad, Dean, Mumma].

The LanguageE core class, expressions, and the Z3 proof engine

Before discussing the details of demonstration steps we'll look at the LanguageE class. This is the core class of the EuclidZ3 system. It is a collection of definitions and functions that allow users to make expressions in the language E. The LanguageE class is where the proof engine resides. Whenever a Proof object is created, a fresh instance of the proof engine, Solver, is created and prepared with all the axioms, definitions, and sorts of the language E. The expressions that users can build using the LanguageE can then be passed to Proof.

Building expressions for EuclidZ3 can be tricky. In its current implementation, there is no wrapper for Z3 commands. For example, if a user wants to assert that the point a is not on the line L:

```
lang = LanguageE()
pc.hence( Not(lang.OnLine(a,L))
```

This means that logical components of expressions like Not, Implies, etc., depend on Z3 functions. For most purposes, And(), Implies(), Not(), Or(), are sufficient. For their documentation see Z3 python bindings.

Demonstration steps with hence

Here is an example, note that # denotes a comment in python. We want to show that if some line L and M both go through 2 points a and b, then those two lines are the same.

```
pc = Proof()

##  let a be a distinct point"
a = Point("a")
pc.construct(a)

## let b be a distinct point"
b = Point("b")
pc.construct(b)

## let L be a distinct line
## through a and b"
L = Line("L")
L.through(a,b)
pc.construct(L)

## hence a == b , Proof should
## complain because this is
## is false.
pc.hence(a == b)

## let m be the line through a and
## b"
M = Line("M")
M.through(a,b)
pc.construct(M)

## hence M does not equal L"
## Proof should complain because
## it follows logically that
## L == M
pc.hence(Not(L == M))
```

When a user enters hence A, the proof-checker checks that A is a consequence of the facts already asserted and, if so, asserts it explicitly to the proof engine, to speed up subsequent checks.

QED and finishing a Proof

Success in proving a theorem occurs when we have constructed the desired objects and shown that they have the claimed properties. Q.E.D. is an initialism of the Latin phrase quod erat demonstrandum, meaning "which had to be proven." It marks the end of a proof.

When the user calls QED(), Proof checks that the negation of the theorems conclusion is inconsistent with the facts that have been asserted thus far. In this way, QED checks that the desired conclusions and objects follow from the hypothesis. Once QED is called on a proof, it is marked as proven and no furter steps can be applied to it.

The Not-so-good and missing features of EuclidZ3

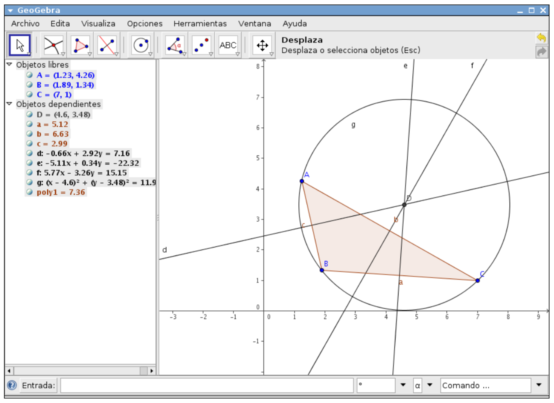
Proofs in the language E begin with a theorem that is to be proven. The theorem has assumed objects and properties, and desired objects and properties. EuclidZ3 works in a different way. The Proof object is treated as a theorem in itself. It is like a theorem that hasn't been proven yet. To assert a theorem to the current proof is a bulky endeavor at the current implementation of EuclidZ3.

Calling the assert method on a Proof object requires a Proof as an argument. This Proof object represents the proof of a theorem. So, to assert a theorem in EuclidZ3 is to assert the proof of that theorem.

When the user asserts a theorem,the proofchecker declares the new objects (points, lines, and circles) to the proof engine, asserts the assumptions to the proof engine, and stores the conclusion.

This implementation of asserting theorems is a gap in representation between the software and the domain of formal languages. This flaw is likely to be overhauled entirely in EuclidZ3's next iteration.

V. Possible Applications to Geometry Tutoring Software



```
A = (1.23, 4.26)
B = (1.89, 1.34)
C = (7, 1)
D = (4.6, 3.48)
a = 5.12
b = 6.63
c = 2.99
d: -0.66x + 2.92y = 7.16
e: -5.11x + 0.34y = -22.32
f: 5.77x - 3.26y = 15.15
g: (x - 4.6)² + (y - 3.48)² = 11.9
poly1 = 7.36
```

Figure 1: *Geogebra – Interactive Geometry Software*

By leveraging new media like visualizations to teach mathematics, the learning experience can become more dynamic, interactive, and ultimately make concepts more tangible to students. This is the claim of interactive tutors like the mathematics software platform Geogebra. I will refer to interactive and visual education software like Geogebra collectively as interactive geometry software (IGS). At the core, these systems are essentially digital playgrounds for drawing and manipulating geometric objects on a screen.

The EuclidZ3 proofbuilding plugin

A proofchecking program for the language E like EuclidZ3 might work as a plugin for IGS with public Application Programming Interface (API).

An API is a set of routines, protocols, and tools for building software applications, in this case, an IGS with a public API means that software components are available to any software developer.

These software components can then be customized and their behaviours modified. A developer with access to an IGS's API could write a "plugin" – an extension/customization of the original program. A silly example might be a plugin that changes the color of the lines drawn in the IGS depending on the time of day.

As a plugin, EuclidZ3 could be a fruitful backend for interactive proof environments. Essentially, by communicating to the IGS the various constraints on geometric objects that have been asserted to EuclidZ3, the IGS can display geometries on the screen. In addition, all the capabilities of the IGS are available. Users should be able to make proofs that inform diagrams, and make diagrams that inform proofs.

Teaching proof based mathematics

Such a platform for interactive geometric proof making would be a valuable pedagogical tool.

There are many theories that attempt to describe how students learn geometry. In 1957 as part of their doctoral dissertations, Dina van

Hiele-Geldof and Pierre van Hiele (wife and husband), described how students learn geometry. At the core of the van Hiele's model of learning are four levels of comprehension, summarized in appendix B.

This model for learning has greatly influenced geometry curricula throughout the world through emphasis on analyzing properties and classification of shapes at early grade levels. However, with the ubiquitousness of computers and the fact that the age for digital literacy gets younger and younger, it's possible to take this a step further and expose children to proof-based concepts earlier than ever before in history.

Programs like EuclidZ3 can enable interactive geometrical proof-making programs and in turn supports the development of 2nd and 3rd levels of apprehension for students learning geometry.

Possible design of plugin

A natural extension point for EuclidZ3 is in the abstract class ConstructableObject. In one implementation, ConstructableObjects might queue themselves – passing it's own label, sort (point, line, circle, etc), and any relevant graphical constraints – to a renderer that arranges the various ConstructableObjects into a datastructure that represents a geometric diagram the IGS's api can understand. What results is that as the proof is built in the EuclidZ3 plugin, objects are constructed in the IGS.

Building the infrastructure required to interface EuclidZ3 with an IGS would be straightforward since public API have clear methods for sending messages between plugins and their frameworks. The real challenge would be in designing an algorithm which arranges geometric objects in a diagram given relationships between those objects. Ironically, the difficulty lies in the vagueness of the descriptions of diagrams.

However, such algorithms for arranging geometric objects according to relationships amongst those objects exist. For example, many computer-aided design programs like SolidWorks or AutoCAD employ similar algorithms to correctly display geometry dynamically. That is, they display geomtries such that the geometric objects obey constraints placed on them – like intersections, verticality, etc – as constraints and other geometric objects are continually added.

VI. Future Work

The following is a list of features that would greatly improve EuclidZ3:

- Creating a visual cue for proofs that require suppositional reasoning, like proof by cases or proof by contradiction. Support of suppositional reasoning is not user friendly for EuclidZ3. It would greatly improve the user's experience to be able to visualize proof trees.
- Overhauling the Proof class. Creating a Theorem class with fields for: assumed objects and conclusions, desired objects and conclusions. The Theorem class can store justifications, which could be sets of other Theorems that justify them.
- Saving Proofs of theorems to persistant storage. This is called cacheing and it has the potential to significantly improve performance since EuclidZ3 would have to call Z3, its proof engine and computational bottle neck, only on unproven theorems.
- Parsing proofs directly from text. EuclidZ3 does not parse Euclidean proofs.

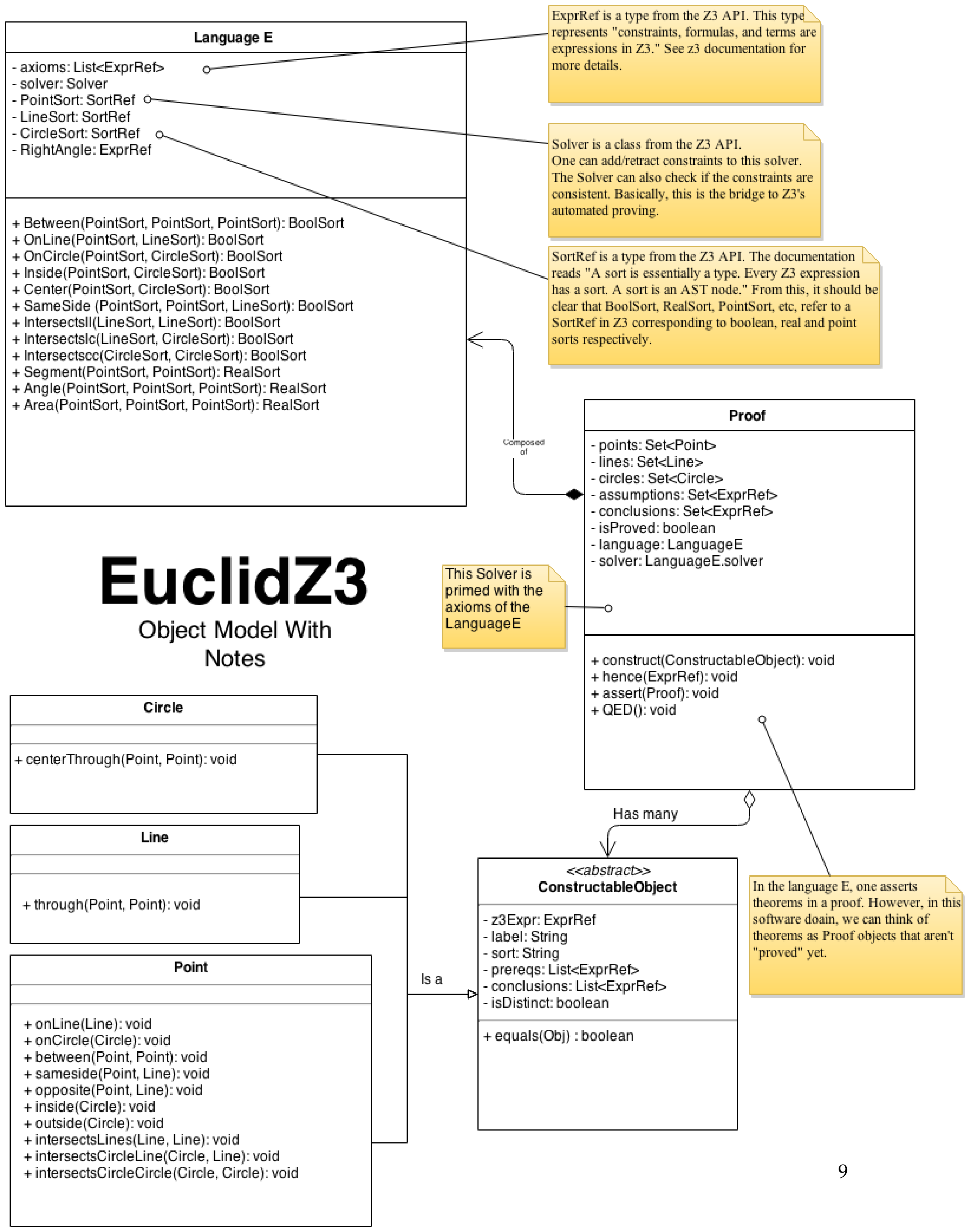
References

[Avigad, Dean, Mumma] Jeremy Avigad, Edward Dean, John Mumma (2009). A Formal System For Euclid's *Elements*. *The Review of Symbolic Logic*, vol 2 #4, December 2009

[de Moura, Bjørner] de Moura, L. M., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS) 2008. Berlin: Springer

[van Hiele] van Hiele, Pierre (1985) [1959]. The Childs Thought and Geometry, Brooklyn, NY: City University of New York, pp. 243252

A. EuclidZ3 Object Model


Language E
- axioms: List<ExprRef>
- solver: Solver
- PointSort: SortRef
- LineSort: SortRef
- CircleSort: SortRef
- RightAngle: ExprRef
+ Between(PointSort, PointSort, PointSort): BoolSort
+ OnLine(PointSort, LineSort): BoolSort
+ OnCircle(PointSort, CircleSort): BoolSort
+ Inside(PointSort, CircleSort): BoolSort
+ Center(PointSort, CircleSort): BoolSort
+ SameSide (PointSort, PointSort, LineSort): BoolSort
+ Intersectsll(LineSort, LineSort): BoolSort
+ Intersectslc(LineSort, CircleSort): BoolSort
+ Intersectscc(CircleSort, CircleSort): BoolSort
+ Segment(PointSort, PointSort): RealSort
+ Angle(PointSort, PointSort, PointSort): RealSort
+ Area(PointSort, PointSort, PointSort): RealSort
Composed of
Proof
- points: Set<Point>
- lines: Set<Line>
- circles: Set<Circle>
- assumptions: Set<ExprRef>
- conclusions: Set<ExprRef>
- isProved: boolean
- language: LanguageE
- solver: LanguageE.solver
+ construct(ConstructableObject): void
+ hence(ExprRef): void
+ assert(Proof): void
+ QED(): void
EuclidZ3
Object Model With Notes
Circle
+ centerThrough(Point, Point): void
Line
+ through(Point, Point): void
Point
+ onLine(Line): void
+ onCircle(Circle): void
+ between(Point, Point): void
+ sameside(Point, Line): void
+ opposite(Point, Line): void
+ inside(Circle): void
+ outside(Circle): void
+ intersectsLines(Line, Line): void
+ intersectsCircleLine(Circle, Line): void
+ intersectsCircleCircle(Circle, Circle): void
Is a
Has many
<<abstract>>
ConstructableObject
- z3Expr: ExprRef
- label: String
- sort: String
- prereqs: List<ExprRef>
- conclusions: List<ExprRef>
- isDistinct: boolean
+ equals(Obj) : boolean

ExprRef is a type from the Z3 API. This type represents "constraints, formulas, and terms are expressions in Z3." See z3 documentation for more details.

Solver is a class from the Z3 API. One can add/retract constraints to this solver. The Solver can also check if the constraints are consistent. Basically, this is the bridge to Z3's automated proving.

SortRef is a type from the Z3 API. The documentation reads "A sort is essentially a type. Every Z3 expression has a sort. A sort is an AST node." From this, it should be clear that BoolSort, RealSort, PointSort, etc, refer to a SortRef in Z3 corresponding to boolean, real and point sorts respectively.

This Solver is primed with the axioms of the LanguageE

In the language E, one asserts theorems in a proof. However, in this software doain, we can think of theorems as Proof objects that aren't "proved" yet.

B.
Van Hiele Levels of Geometric Learning

These descriptions are courtesy of wikipedia.

Level 0 Visualization

The focus of a child's thinking is on individual shapes, which the child is learning to classify by judging their holistic appearance. Children simply say, "That is a circle," usually without further description.

Level 1 Analysis

The shapes become bearers of their properties. The objects of thought are classes of shapes, which the child has learned to analyze as having properties. A person at this level might say, "A square has 4 equal sides and 4 equal angles. Its diagonals are congruent and perpendicular, and they bisect each other." The properties are more important than the appearance of the shape. However, at this level, properties do not overlap and children will often introduce erroneous properties. All reasoning at this level is inductive; students learn properties from many examples.

Level 2 Deduction

Students at this level understand the meaning of deduction. The object of thought is deductive reasoning (simple proofs), which the student learns to combine to form a system of formal proofs (Euclidean geometry). Learners can construct geometric proofs at a secondary school level and understand their meaning. They understand the role of undefined terms, definitions, axioms and theorems in Euclidean geometry. However, students at this level believe that axioms and definitions are fixed, rather than arbitrary, so they cannot yet conceive of non-Euclidean geometry. Geometric ideas are still understood as objects in the Euclidean plane.

Level 3 Rigor

At this level, geometry is understood at the level of a mathematician. Students understand that definitions are arbitrary and need not actually refer to any concrete realization. The object of thought is deductive geometric systems, for which the learner compares axiomatic systems.