# A domain reasoner for geometry exercises

*Stéphane Thibaud*
*student number: 851504350*

Date of presentation: 09-01-2017

Open University of the Netherlands
faculty of Management, Science & Technology
Master Software Engineering

Master's thesis

# A domain reasoner for geometry exercises

*Stéphane Thibaud*
*student number: 851504350*

January 8, 2017

Date of presentation: 09-01-2017

Master's committee
Chairman: prof. dr. J.T. Jeuring
Supervisor: dr. B.J. Heeren

Course code: T75317

# Contents

**Abstract**

With advances in ICT around the world, digital tutors are an increasingly attractive option to provide education to a large audience inexpensively. Important components of a digital tutor include the exercises or a method to generate exercises, an algorithm for finding and verifying solutions to exercises and a method of providing hints to a student while the student is working on an exercise. The domain reasoner models all paths from the proposition(s) of an exercise to the solution(s). This enables the digital tutor to provide hints from any situation the student might encounter. This thesis contributes a method to generate a domain reasoner from an exercise solution in the domain of high school level geometry exercises. The program representing a solution to an exercise is first represented as a directed acyclic graph of which the edges represent steps in the solution. Each edge uses a formal rule to execute a step and human-readable hints are attached to these rules. Since an algorithm for generating solutions from the formal description of an exercises currently exists, this domain reasoner enables the generation of hints from just that formal description of an exercise. The exercise-specific domain reasoner enables feedforward, feedback and worked-out examples as well as a degree of adaptivity to a student's knowledge by providing hints for steps composed of smaller steps.

# List of Figures



# List of Tables

# 1 Introduction

With the advance of access to ICT around the world [26], the target audience of digital educational technology widens. Such educational technology can be used in remote areas and has been shown to have a positive impact on student achievements in the field of mathematics at high school level [9].

According to Van Lehn, a tutorial where one teacher teaches one student intensively, has been shown to increase students' achievements about 0.79 standard deviations above those who have no tutor at their disposal [27]. It would be ideal if every student could be taught by a private teacher, but this is not likely to be affordable for a great number of students. Fortunately, Van Lehn has also demonstrated [27] that a digital mathematics tutor emulating the suggestions that a human tutor would give turns out to be a very effective substitute. Such a digital mathematics tutor would be responsible for monitoring the student's progress and providing hints whenever the student is stuck, just like a human tutor.

The main purpose of this research is to achieve such a digital mathematics tutor that helps students in solving high school level geometry exercises by providing a student with hints on how to arrive at the solution. These hints can have different forms. One type of hint could point out unnecessary (or wrong) steps taken by a student or praise the student after a correct step (feedback). Another type of hint could give a step towards the solution from the current situation (feedforward).

A worked example would give the complete solution to the exercise. The student can learn (acquire) procedural rules/strategies from this and try to solve similar exercises. Although the effectiveness of worked examples alone is disputed, there is consensus that they improve performance when combined with other learning methods [6, 8, 5].

A longer term purpose of this research is to integrate this tutor with an interactive tool that provides drawing functionality and virtual equivalents of pen, compass and ruler, such as GeoGebra [18].

Important components of a digital tutor include the exercises or a method to generate exercises, an algorithm for finding and verifying solutions to exercises and a method of providing hints to a student while the student is working on an exercise. The domain reasoner models all paths from the proposition(s) of an exercise to the solution(s). This enables the digital tutor to provide hints from any situation the student might encounter. This thesis contributes a method to generate a domain reasoner from an exercise solution in the domain of high school level geometry exercises. The program representing a solution to an exercise is first represented as a directed acyclic graph of which the edges represent steps in the solution. Each edge uses a formal rule to execute a step and human-readable hints are attached to these rules. Since an algorithm for generating solutions from the formal description of an exercise currently exists, this domain reasoner enables the generation of hints from just that formal description of an exercise. The exercise-specific domain reasoner enables feedforward, feedback and worked-out examples as well as a degree of adaptivity to a student's knowledge by providing hints for steps composed of smaller steps.

This thesis is organized as follows: the research problem is described in Section 2.1 and the research questions are formulated in Section 2.2. Section 3, puts this thesis in perspective to related research, including the ACT-R the-

ory (Section 3.1) for modeling rational knowledge[1], knowledge tracing (Section 3.1.1), existing program synthesis algorithms (Section 3.2) and the Ideas framework (Section 3.3). Findings that were made during this exploratory research are bundled in Section 4. Section 4.1 formally defines rules and strategies and provides an example of a strategy, which is used throughout this thesis. Section 4.2 introduces the data structures that are used in the domain reasoner. Section 4.3 outlines how hints are represented and used for an exercise. Section 4.3.1 describes how hints can be generated with the domain reasoner. Section 4.3.2 shows how the hints can be made adaptive to a student's knowledge level. Section 4.4 describes how an existing program synthesis algorithm is used with the domain reasoner and Section 4.4.1 shows how an existing program synthesis algorithm can be integrated with the domain reasoner implementation. The findings are validated with several test cases in Section 5 and a discussion of the results, conclusions and suggestions for future research are given in Section 6.

# 2 Research method and objectives

## 2.1 Research context

High school level geometry exercises are often solved in an unstructured way. They are popular in high school as an introduction to geometry, possibly because no exact (algebraic) representation of such exercises is necessary to solve them. An example of such an exercise is that of building a square with a straight edge (an unmarked ruler) and a compass, as shown in Figure 1. The square construction exercise will be referred to throughout this thesis. The sequence of steps that leads to the square of Figure 1 is:

1. Circle $X$ of arbitrary size (with point $A$ in the middle) is drawn.
2. Point $B$ is chosen on circle $X$.
3. Circle $Y$ is drawn around point $B$, that passes through point $A$. Circles $X$ and $Y$ now intersect each other at points $C$ and $D$.
4. A line is drawn through points $C$ and $D$.
5. A line is drawn through points $A$ and $B$.
6. Line intersection $E$ is chosen as the middle of a new circle passing through $A$.
7. The line-circle intersection $F$ and $G$ are labeled.
8. A line $L$ is drawn though points $A$ and $F$, creating line circle intersections $I$ and $K$.
9. A line $L'$ is drawn though points $A$ and $G$, creating line circle intersections $H$ and $J$.
10. Line segments are drawn through $H$, $I$, $J$, $K$ and $H$ respectively, creating the square.

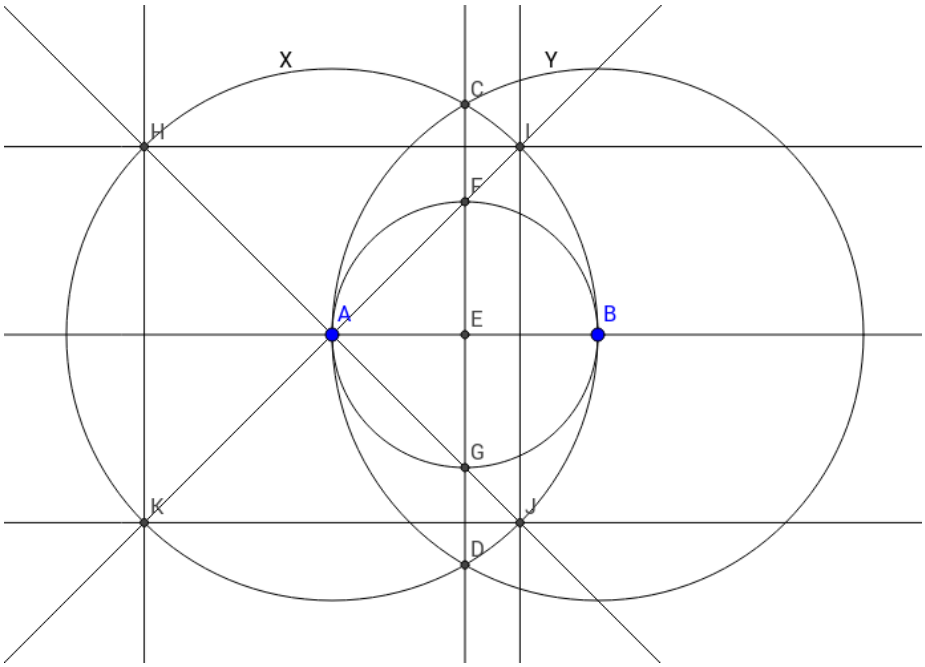


Figure 1: Constructing a square using circles and lines (screenshot from GeoGebra – a mathematics tool that allows drawing of geometric structures).

The absence of an exact (algebraic) representation also means that existing algebra solvers cannot be used directly for such exercises, since the visual representation must first be converted to algebra. Moreover, Gulwani et al. [16] have found that symbolic reasoning does not scale very well for finding geometrical proofs.

For a digital tutor, an important component is the domain reasoner [14, 17]. The domain reasoner supports in finding a solution to a given problem. This is done by tracing a path from given propositions to a solution (model tracing). A description of paths from the propositions to the solutions is required. One approach for building such descriptions is by using the strategy language presented in Section 3.3. A solution of one instance of a problem can be found by using a program synthesis algorithm. A program synthesis algorithm yields a program that can serve as input for the domain reasoner. Such a program synthesis algorithm is presented in Section 3.2. An example of deriving a strategy is presented in Section 4.1.3.

A digital mathematics tutor should capture knowledge that is interesting in more than one exercise. This is the knowledge that the student should learn. The ACT-R theory, that can be used to capture such knowledge, is presented in Section 3.1. Using this theory, it is possible to model the knowledge that the student has likely learned in a process called 'knowledge tracing'. This is presented in Section 3.1.1. A framework that is based on the ACT-R theory, but that has a powerful 'strategy language' to represent procedures, is the Ideas framework, which is presented in Section 3.3.

[1] knowledge which can be obtained through reasoning

## 2.2 Research questions

The main research question for this thesis is: 'How can adaptive hints, feedback and worked examples be generated for high school level geometry exercises?'. To answer that research question several sub-questions need to be answered:

1. Does an available program synthesis algorithm provide adequate (partial) solutions to problems that can be used to generate hints?

2. How can re-usable sub-strategies be distilled from existing program synthesis algorithms?

3. Can variation in granularity of steps in a solution be used to provide adaptivity of hints to target particular (groups of) students?

For the first sub-question, available program synthesis algorithms [22, 16] for solving high school level geometry exercises will be used to establish whether these can be used in an interactive setting where the digital tutor should be able to provide hints. The output from such a program synthesis algorithm should be such that this can be used to generate useful hints. It should, for example, be possible to match steps that a student takes with steps that occur in the solution to provide hints. The program synthesis algorithm by Gulwani et al. [16] will be considered in particular, since its performance for finding proofs to geometric problems is good enough for an interactive setting.

The second sub-question addresses the distillation of procedural and declarative knowledge from the program synthesis algorithm of Gulwani et al. It is well possible that a certain strategy (or a structured combination of procedural rules) works on multiple exercises. If this is the case, it is valuable to build a collection of such strategies, so that they can be used for adaptive hints. It is attempted to directly translate the program synthesis algorithms into strategies by mapping the language constructs onto strategy combinators.

For sub-question three, the use of hints that are adapted to the knowledge of a particular student or a group of students should be investigated. It might be possible to target a particular student by providing a hint for which it is assumed that this provides the student with new information (the student can be assumed to learn new information by a particular hint if (s)he has not completed a step in a mastery learning program yet or when a knowledge tracing model estimates that the knowledge has not yet been learned).

In this thesis, the ability to provide hints for high school level geometry exercises is researched. The hints should be in a textual format, so that they are human-readable. A graphical user interface is out of the scope of this thesis. Adaptivity of hints is considered to the extent of providing hints for different levels of granularity (see Section 4.3.2).

# 3 Related work

## 3.1 ACT-R

The ACT-R (Adaptive Character of Thought – Rational) theory is an attempt to model the rational behavior of human beings. This theory underlies several modern digital tutoring systems including all tutors developed using the Ideas

framework. At the basis of this theory lies the declarative memory and the procedural memory [4]. In the declarative memory, facts are stored. In the procedural memory, procedures using those facts are stored. The theory advocates that complex behavior is the result of using a lot of simple procedures stored in procedural memory (and used with declarative memory). Anderson states that the 'mystical skill of recursive programming' can be modeled by a set of about 500 rules.

### 3.1.1 Knowledge tracing

The rules in procedural memory of the ACT-R theory should be acquired by a student, but acquiring those rules (i.e. 'learning') takes time. To track the progress of a student, a technique called 'knowledge tracing' can be used [10]. With this technique a model of the ideal student (a model of a student that has learned the task at hand perfectly) is used as a reference. The technique can then be used to estimate the probability that an actual student has acquired a rule that the ideal (model) student has acquired (for each rule in the model). The knowledge of knowledge that a student has (the probability of having acquired rules) can be constantly updated based on input from the student. Corbett and Anderson [10] propose a model where a rule is either learned by a student or not learned. The probability that a rule is learned is calculated based on a student's actions.

There are some more recent improvements to this model: Baker, Corbett and Aleven propose to directly categorize slips or guesses using machine learning [7], whereas Pardos and Hefferman [24] propose to alter the Bayesian network's structure by introducing a student node that influences student-specific parameters in the model of Anderson et al.

The knowledge of knowledge acquired can be used to measure how a tutor influences the speed with which a student learns and to train the tutor so that it increases this learning rate.

## 3.2 Existing program synthesis algorithms

The geometry-theorem proving machine as presented by Gelernter [13], is one of the earliest successful attempts to solve geometry problems that involve construction of intermediate steps in a diagram to test for validity of the steps. The theorem proving machine is able to provide a wide range of useful proofs, but as the number of steps (and therefore elements needed in the proof) increases, the performance rapidly decreases. One factor in this is that symmetry in constructions or proofs is not recognized by the machine (e.g., a square's points can be enumerated in a successive way in eight manners). A more recent approach to solving high school level geometry exercises programmatically is the approach found by Gulwani et al. [16]. The approach by Gulwani et al. uses five object

| | Function | Description |
|---|---|---|
| L | = Line(p1,p2) | L is the line joining p1 and p2 (provided p1 $\neq$ p2). |
| C | = Circle(p,r) | C is the circle with center p and radius r. |
| r | = Length(p1,p2) | r is the length of the segment from p1 to p2 (provided p1 $\neq$ p2). |
| p | = LineLineXn(L1,L2) | p is the point that lies at the intersection of L1 and L2 (provided L1, L2 are not parallel) |
| $\overrightarrow{p}$ | = LineCircleXn(L1,C1) | $\overrightarrow{p}$ is the vector containing (1 or 2) points that lie on both L1, C1 (provided they intersect) |
| $\overrightarrow{p}$ | = CircleCircleXn(C1,C2) | $\overrightarrow{p}$ is the vector containing (1 or 2) points that lie on both C1, C2 (provided they intersect) |
| $\overrightarrow{p}$ | = ExplodeAngle(a) | $\overrightarrow{p}$ is the vector of (three) points that define angle a. |

Table 1: Basic library of functions used by the GeoSynth algorithm of Gulwani et al. [16]

| | Function | Description |
|---|---|---|
| L | = PerpendicularBisector2Points(p1,p2) | L is the perpendicular bisector of line joining p1 and p2. |
| p | = MirrorPointLine(p1,L) | p is the reflection of p1 about line L. |
| $\overrightarrow{C}$ | = CircleGivenChordAngle(L,a) | $\overrightarrow{C}$ is a vector of (1 or 2) circles C s.t. L is a chord of C subtending angle a. |
| L | = ConstructLineGivenAngleLinePoint(L1,a,p) | L is at an angle a with L1 at point p (on L1). |
| C | = ConcentricCircle(C1,r) | C is concentric to C1 and at a distance r away from it. |
| L | = PerpendicularToLineThruPoint(p,L1) | L is perpendicular to L1 and passes through p. |
| $\overrightarrow{L}$ | = AngularBisectorLines(L1,L2) | $\overrightarrow{L}$ is the tuple of (two) lines that are angular bisectors of L1 and L2. |
| p | = MidpointGiven2points(p1,p2) | p is the midpoint between p1 and p2. |
| $\overrightarrow{L}$ | = TangentPointToCircle(p,C) | $\overrightarrow{L}$ is the vector of (two) lines that are tangent to C and pass through p. |
| L | = ParallelLine(p,L1) | L is the line parallel to L1 and passing through p. |
| $\overrightarrow{L}$ | = ParallelLineGivenLength(L1,r) | $\overrightarrow{L}$ is the vector of (two) lines that are parallel to L1 and distance r away from it. |

Table 2: Extended library of functions used by the GeoSynth algorithm of Gulwani et al. [16]

types: points, lines, angles, lengths and circles. More formally:

Let $P$ be the set of all points.

Let $L$ be the set of all lines.

Let $A$ be the set of all angles.

Let $D$ be the set of all lengths.

Let $C$ be the set of all circles.

Let $Objects = P \cup L \cup A \cup D \cup C$ be the set of all objects.

Let $I \subseteq Objects$ be the set of input objects for an exercise.

Let $O \subseteq Objects$ be the set of output objects for an exercise.

Gulwani et al. uses an expression language in which the only expressions are executable functions $F_i$ on $Objects$. These functions are defined in a library $Lib$. For GeoSynth, $Lib$ contains functions from a 'basic library' and an 'extended library'. These are shown in Figure 1 and Figure 2 respectively.

A specification language is used in order to describe the exercise. This specification language consists of the following operators:

$$\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \qquad \text{(boolean conjunction)}$$

$$=, \neq : \mathbb{R} \times \mathbb{R} \to \mathbb{B} \qquad \text{(boolean equality/inequality)}$$

$$+, -, *, / : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \qquad \text{(arithmetic)}$$

$$distL : P \times L \to \mathbb{R} \qquad \text{(distance between a point and a line)}$$

$$slope : P \times P \to \mathbb{R} \qquad \text{(slope of the line through two given points)}$$

In addition, set membership $\in$ is used to check for the types of objects and the set equality $=$ is used to conveniently specify sets. A specification is an

expression using the above operators that is of type boolean ($\mathbb{B}$). The input specification $\phi pre$ is a function of input objects $I$, while the output specification $\phi post$ is a function of both input objects $I$ and output objects $O$. The combined specification of an exercise contains both the input specification and the output specification: $\phi spec = (\phi pre(I), \phi post(I, O))$. The GeoSynth algorithm, shown in Figure 2, uses $\phi spec$ and $Lib$ as inputs.

```
GeoSynth(φspec, Lib):
// Input φspec := (φpre, φpost) is the specification
// Input Lib:= ⟨(F_i)_{i=1,2,...}⟩, where F_i implements
               the i-th library function
// Output: A program or "Failure"
Ī_c := Random concrete objects s.t. φpre(Ī_c) holds
Ō_c := Concrete objects s.t. φpost(Ī_c, Ō_c) holds
return GeoSynthRec(Ī_c, Ō_c, Lib, ϵ, φspec);

GeoSynthRec(Ī_c, Ō_c, Lib, P, φspec):
// Input Ī_c, concrete objects we have constructed
// Input Ō_c, concrete objects we wish to construct
// Input P, the program (so far)
if (Ō_c are contained in Ī_c) return(Verify(P, φspec));
forall functions F_i ∈ Lib {
  forall possible choices of arguments args for F_i
         picked from Ī_c {
    newObj := F_i(args);
    if (newObj is Good and
        newObj is not already present in Ī_c) {
       newP := (P; t⃗ := F_i(args));
       newI := Ī_c ∪ {newObj};
       result := GeoSynthRec(newI,Ō_c,Lib,newP,φspec);
       if (result ≠ "Failure") return(result);
    }
  }
}
return "Failure"

Verify(P, φspec):
// Check if P works correctly on a second input
Ī_c := Random concrete objects s.t. φpre(Ī_c) holds
Ō_c := f_P(Ī_c);
if (φpost(Ī_c, Ō_c) == true) return(P);
else return "Failure";
```

Figure 2: The GeoSynth algorithm as presented by Gulwani et al. [16].

As an example, a specification of inputs and outputs of the square construc-

tion exercise using this specification language is:

$$
\begin{aligned}
\phi pre(I) &:= I = \{a, b\} \wedge a, b \in P \\
\phi post(I, O) &= d, e, f, g \in O \\
&\wedge\ d, e, f, g \in P \\
&\wedge\ Line(d, e), Line(e, g), Line(g, f), Line(f, d) \in L \\
&\wedge\ slope(d, e) * slope(e, g) = -1 \\
&\wedge\ slope(e, g) * slope(g, f) = -1 \\
&\wedge\ slope(g, f) * slope(f, d) = -1 \\
&\wedge\ distL(d, Line(e, g)) = distL(e, Line(f, g)) \\
&\wedge\ distL(e, Line(f, g)) = distL(g, Line(f, d)) \\
&\wedge\ distL(g, Line(f, d)) = distL(f, Line(d, e))
\end{aligned}
$$

The pre-condition of this specification states that there are two elements in the input set that are both points. The post condition states that there are four lines and that each line within these four lines is perpendicular to another of the four lines (product of slopes equal to $-1$ means that the lines are perpendicular). In addition, the (shortest) distance between each point and a line it is not located on, should be equal for all four points.

The GeoSynth algorithm starts by selecting a random subset of concrete input objects $\overrightarrow{I_c} \subseteq I$ that satisfy $\phi pre(\overrightarrow{I_c})$ and a subset of concrete output objects $\overrightarrow{O_c} \subseteq O$ that satisfy $\phi post(\overrightarrow{I_c}, \overrightarrow{O_c})$. The algorithm then calls GeoSynthRec with $\overrightarrow{I_c}$, $\overrightarrow{O_c}$, $Lib$, $\phi spec$ and an empty program $P$. It then checks if the output objects are a subset of the input objects ($\overrightarrow{O_c} \subseteq \overrightarrow{I_c}$). If this is the case, a candidate program has been found which is verified (by calling Verify) against a second set of randomly selected input-output objects. If $\overrightarrow{O_c}$ is not a subset of $\overrightarrow{I_c}$, the objects have not yet been constructed using the program and the program is extended by trying all possible library function applications and only adding the resulting objects to $\overrightarrow{I_c}$, which are deemed 'good'. A 'good' object is one that satisfies a set of rules that can be checked by performing a few backward steps from the set of output objects $\overrightarrow{O_c}$ (these are specified in detail by Gulwani et al.). Whenever a 'good' object is added to $\overrightarrow{I_c}$, the program $P$ is extended with the function that was used to construct that object and GeoSynthRec is recursively called to extend the program further (until the solution is reached: $\overrightarrow{O_c} \subseteq \overrightarrow{I_c}$).

This approach of building a program is very fast in comparison with symbolic methods due to the fact that only one problem instance is used to build the program. The program $P$ is therefore not guaranteed to generalize to all problem instances, but the probability that program P does not generalize, rapidly decreases as the number of function applications contained in the program increases. Using the function Verify - which checks for a second problem instance, this probability approaches zero even for programs with few function applications (the probability would be zero if a computer would be able to represent all – infinitely many – real numbers).

### 3.3 Ideas

Ideas is a framework for developing domain reasoners that give intelligent feedback.[2] Using Ideas, domain reasoners for solving linear, quadratic and higher-degree equations, Gaussian elimination, and many other domains have been developed. The strategy language published by Heeren, Jeuring and Gerdes [17] lies at the basis of the Ideas framework. The strategy language is a context-free grammar that can impose an order of application on a set of rules. Where ACT-R means to capture procedural rules, the strategy language captures the order of application of procedural rules. The strategy language is used to model the knowledge needed for solving an exercise or a set of exercises. The following definitions of strategy combinators form the basis of the strategy language:

$language\ (s <*> t) = \{xy \mid x \in language\ s,\ y \in language\ t\}$
strategy concatenation

$language\ (s <|> t) = language\ s \cup language\ t$
strategy choice

$language\ (fix\ f) = language\ (f\ (fix\ f))$
strategy recursion

$language\ (label\ l\ s) = language\ s$
strategy labeling

$language\ (symbol\ r) = \{r\}$
strategy symbol (a symbol is usually a procedural rule)

$language\ succeed = \{\epsilon\}$
strategy success

$language\ fail = \emptyset$
strategy failure

There are additional strategy combinators that can be expressed in the above combinators (i.e.: the above strategy combinators are used to build some more complex strategy combinators). This makes the strategy language a very powerful and rich language.

## 4 Findings

In the previous section, the ACT-R theory, an existing program synthesis algorithm and the Ideas framework were introduced. These form the basis of a domain reasoner for geometry exercises. In this section, the domain reasoner for generating geometry hints based on the Ideas framework and an existing program synthesis algorithm is presented.

[2] Citation from the Ideas website at http://ideas.cs.uu.nl/www/

## 4.1 Rules and strategies

### 4.1.1 Refinement rules

Refinement rules are used to describe a step of an exercise in a formal way. This formal description of steps in an exercise is used to generate hints and to interpret actions by a student. The functions shown in Figure 1 and 2 are refinement rules in the domain of geometry. A refinement rule uses existing objects to produce new objects. For example: the refinement rule *Line* uses two points to produce a line. A refinement rule thus refines a geometric structure: a vague description of a line using two points can be refined to an actual line by using a refinement rule. In the same manner, a point and a length can be refined to a circle by using the *Circle* refinement rule and two circles can be refined to their intersection points by using the $CircleCircleXn$ refinement rule. Refinement rules can be combined to form strategies (see Section 4.1.3).

The strategy language uses the designation 'rewrite rule' instead of 'refinement rule', but this name is based on the equivalence of a term before and a term after application of the rule. In other domains, rules are used to convert one term to an equivalent term. An example can be found in the domain of propositional logic. A very simple rule is that of double negation: $\neg\neg p = p$. Application of the 'double negation' rule yields an equivalent term here. In the geometry domain, a term can be defined as a set of objects *Objects* (as in Section 4.1.2). A refinement rule would then create a new set $Objects'$ for which $Objects \subseteq Objects'$. For example: the $LineCircleXn$ rule from Figure 1 produces the 0, 1 or 2 points in which a given line and circle intersect. The set $Objects'$ can thus contain more objects and the result of applying a rule in the geometry domain is not an equivalent term. The designation 'rewrite rule' is therefore avoided in favor of 'refinement rule' throughout this thesis.

Refinement rules are referred to as 'library functions' by Gulwani et al. [16] (see Figures 1 and 2). Implementations of a subset of the library functions from Gulwani et al. can be found in Appendix A. Appendix B contains the conversion of the library functions to refinement rules as used in the Ideas framework, so that the rules can be used in strategies.

### 4.1.2 Definition of the domain

In the domain of geometry exercises, angles, distances, lines, circles and relative positioning are valued, but absolute positions are not. The domain of high school level geometry exercises has a set of objects *Objects* that are used in the specification of the exercises and their solutions. For every object $o \in Objects$ one of the following three rules holds:

1. $o$ is a point.
2. $o$ is an angle.
3. $o$ is a function on *Objects* (from library *Lib* defined in Section 3.2).

Circles and lines are part of the functions defined in library *Lib*, so they are not explicitly stated in the above rules. A function on *Objects* is called a refinement rule, as it refines a geometric structure by producing new objects from existing objects.

$$r_1 = Length(A, B) \tag{1}$$
$$X = Circle(A, r_1) \tag{2}$$
$$Y = Circle(B, r_1) \tag{3}$$
$$[C, D] = CircleCircleXn(X, Y) \tag{4}$$
$$L_1 = Line(C, D) \tag{5}$$
$$L_2 = Line(A, B) \tag{6}$$
$$E = LineLineXn(L_1, L_2) \tag{7}$$
$$r_2 = Length(A, E) \tag{8}$$
$$Z = Circle(E, r_2) \tag{9}$$
$$[F, G] = LineCircleXn(L_1, Z) \tag{10}$$
$$L_3 = Line(A, F) \tag{11}$$
$$[K, I] = LineCircleXn(L_3, X) \tag{12}$$
$$L_4 = Line(A, G) \tag{13}$$
$$[H, J] = LineCircleXn(L_4, X) \tag{14}$$
$$L_5 = Line(I, J) \tag{15}$$
$$L_6 = Line(J, K) \tag{16}$$
$$L_7 = Line(K, H) \tag{17}$$
$$L_8 = Line(H, I) \tag{18}$$

Figure 3: Drawing a square with a straight edge using the library functions specified by Gulwani et al.

#### 4.1.3 Strategy for the construction of a square

An example of an exercise that can be represented by a strategy is the square construction exercise shown in Figure 1. The library presented by Gulwani et al. [16] can be used to translate the steps of Section 2.1 to a program consisting of precise rules, as shown in Figure 3. Points $A$ and $B$ are given in the specification of the exercise. Lines $L_5$, $L_6$, $L_7$ and $L_8$ are the lines of the square resulting.

There are a couple of things to note about this exercise:

- The circles X and Y can be drawn in any order.
- The square drawn in circle X could also have been drawn in circle Y.
- The size and rotation of the square do not matter.

These observations mean that there are a lot of solutions to such an exercise. In this exercise there are infinitely many solutions, since there are infinitely many ways to rotate and size the square. Even with point $A$ and $B$ as fixed there are still 288 solutions (the number of topological sorts of the graph in Figure 4). It is equally simple to find infinitely many ways in which this problem will not be solved: one could keep on drawing circles (and that will never lead to a square since there are no straight lines).

To model this solution space, a strategy can be found by first building a directed acyclic graph modeling the dependencies between the rule applications

in the program as proposed by Keuning et al. [19]. This is done for the example of drawing a square in Figure 4. Each node represents the program state after an application of a rule (which corresponds to a statement in the example of Figure 3). The edges represent dependencies between applications of rules. It is

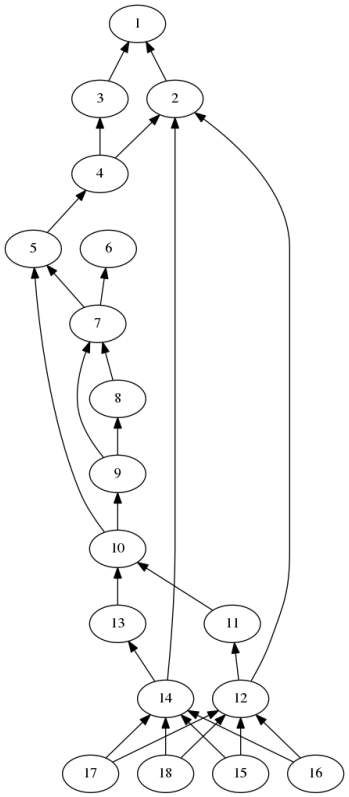


Figure 4: A directed acyclic graph representing the order of statements in the program of Figure 3.

possible to use this graph for the generation of a strategy. All topological sorts of the graph would yield all ways in which the program can be executed without changing the end result. A strategy can be generated from these topological sorts.

Each topological sort is a sequence of rules: this can be expressed in a strategy with the $< * >$ (sequence) strategy combinator. The strategies for each topological sort can then be combined with the $< | >$ (choice) strategy combinator to form one comprehensive strategy by left factorizing the sequences.

## 4.2 Data structures

To represent geometric structures for use with the strategy language of Heeren et al. [17] and the program synthesis algorithm of Gulwani et al. [16], a data structure must be defined for them. Angles, vectors, points, lines, circles and lengths are defined as follows:

```
data Angle = Angle Double
data Vector = Vector Double Double
data Point = Point Double Double
data Line = Line Point Vector
data Circle = Circle Point Double
type Length = Double
```

The coordinates (Double types of a point) are not used for building a strategy or for generating hints, but do provide the ability to draw images of the objects (which could be used for visual hints). These are generalized as objects in order to fit the (more abstract) definition of refinement rules:

```
data Object = AngleObject Angle | PointObject Point
  | LineObject Line | CircleObject Circle | LengthObject Length
```

A refinement rule in the domain of geometry is called a refinement in this implementation. Such a refinement transforms a list of objects into a new list of objects or returns 'nothing' if the it cannot be applied to given objects.

```
type Refinement = [Object] -> Maybe [Object]
```

Objects are labeled so that strategies can be built that reference only the labels of objects. An example is shown in Appendix C.

```
type ObjectLabel = String
type LObject = (Label, Object)
```

An operator is the name of a refinement. A mapping between operators and refinements is found in function 'execute' of Appendix A. An operation is uniquely determined by the operation label. The operation has an operator, a list of object labels used as inputs, a list of object labels used as outputs and it contains references to previous operations in which the input objects were determined.

```
type OperationLabel = Int
type Operation = (OperationLabel, Operator, [OperationLabel],
                  [ObjectLabel], [ObjectLabel])
```

An example of an operation is:

```
(3, "lineLineXn", [1, 2], ["a", "b"], ["c"])
```

This operation is labeled 3, uses the operator lineLineXn to specify a refinement and depends on operations 1 and 2. From the results of these operations it uses lines $a$ and $b$ and produces a point $c$ by applying the lineLineXn (line intersection) refinement.

Finally, a program contains a list of labeled objects (the state) and a list of operations that led to that state:

```
type Program = ([Operation], [LObject])
```

Using these data structures it is possible to represent programs in a simple manner. An example of a program representing the construction of a line from two points is shown below:

```
([
    (2, "lineFromPoints",[0,1],["a","b"],["l"]),
    (1, "specification",[],[],["b"]),
    (0, "specification",[],[],["a"])
  ],[
    ("l",LineObject (Line (Point 0.0 0.0) (Vector 0.0 300.0))),
    ("b",PointObject (Point 0.0 300.0)),
    ("a",PointObject (Point 0.0 0.0))
])
```

### 4.3 Hints

A student working on the square construction exercise might be stuck in one of the steps. These hints can have different forms. One type of hint could point out unnecessary (or wrong) steps taken by a student or praise the student after a correct step (feedback). Another type of hint could give a step towards the solution from the current situation (feedforward).

A worked example would give the complete solution to the exercise. The student can learn (acquire procedural rules/strategies) from this and try to solve similar exercises. Worked examples have been shown effective in improving the performance of students when combined with other learning methods [6, 8, 5]. Section 4.3.1 gives a demonstration of the way these different types of hints can be obtained using the Ideas framework.

Feedback could be given after every action the student applies. If the action fits in the strategy of the exercise, the feedback could simply be affirmative, but when the action of the student deviates from the strategy this may be due to one of the following situations:

1. The student applied a clever step that was not captured in the strategy, but still advances in an optimal[3] way towards a solution.

2. The student applied a step that was not captured in the strategy and advances sub-optimally towards a solution.

3. The student applied a step that was not captured in the strategy and does not advance towards a solution.

4. The student applied a step that was not captured in the strategy and that excludes the possibility of ever attaining a solution.

In situation 1, it would be nice to detect that, although the strategy did not capture the step, it still works towards a solution and the student can be given a compliment. See Section 4.4 for a discussion on this situation.

In situation 2, the digital tutor can give a suggestion for a more optimal step. Although it is not always possible to detect whether a step that was not

[3]The optimal solution can be defined as the solution with the least amount of steps, since a short proof is often preferred over a very long one, but there are other factors that might contribute to the elegance of a proof. For example, it might also be preferable to avoid more complex refinement rules to improve readability. Here, it is simply assumed that the quality of a proof is quantifiable so that there exists a function $e : S \to \mathbb{R}^n$ where $S$ is the set of all possible solutions, $n \in \mathbb{N}^+$ and $e(s)$ is an 'elegance score' that can be optimized. It is a multi-objective optimization when $n > 1$.

captured in the strategy advances optimally or sub-optimally towards a solution (the difference between situation 1 and 2), it is often still possible to make an approximation. When the measure of solution elegance[3] $e(s)$ is the number of steps, the goal is to minimize $e(s)$. When the strategy is first generated, there is a minimum number of steps *in that strategy*. After the student takes a step, a new strategy can be generated that takes the new situation as 'specification'. In this new strategy, the minimum number of steps may be higher than in the earlier strategy (which means we are definitely in situation 2) or it may be equal (which means we are in situation 1 or that the earlier strategy did not contain the optimal solution).

An example of situation 3 is just drawing a random line not passing through any points of the square construction exercise. It is still possible to complete the square in this situation. Feedback could indicate here that the step is unnecessary. Situation 4 cannot actually happen in the domain of geometry exercises, since new objects can always be added. The worst that happen happen is that an unnecessary step is taken (situation 3).

Adaptivity of hints can be provided as well. In the example above, a line is drawn thorugh $A$ and $G$. This line is the perpendicular bisector of the points $I$ and $K$. In this case the student could receive a hint to draw the perpendicular bisector between $I$ and $K$. If the student has not learned about the perpendicular bisector of two points yet, the student could be presented with a hint to draw the line between $A$ and $G$. Whether or not the student knows how to draw a perpendicular line can be assessed with the technique of knowledge tracing (Section 3.1.1) or can be specified manually through intervention of a teacher. The advantage of using knowledge tracing is that a type of hint of the form 'remember what you've learned...' can be given. See Section 4.3.2 for more details on adaptiveness.

### 4.3.1 Providing hints for an exercise

The Ideas framework provides the possibility to specify hints in a script file. Such a file was created for the exercise of constructing a square (see Appendix E). Loading this file while running the Ideas framework as a server makes it possible to request a hint by sending an XML document to the server. This XML specifies the steps that the student has already taken (the state). The server returns an XML with a hint for one of the rules that can be applied from the given state.

To ask for a feedforward hint, a student could push a 'Get Hint' button. Figure 5 gives an example of such a hint in the form of a screen mock-up. After each step, feedback can also be provided. In Figure 6, the student is presented with the feedback 'Correct!' after a circle has been drawn that corresponds with a step in the strategy. When a student is completely stuck, the student could push the 'Solution' button and arrive at a screen in which a worked-out solution is presented (see Figure 7). The actual XML requests that would be constructed by the user interface and the responses that would be generated by the Ideas framework for the hints depicted in Figures 5, 6 and 7 are provided in Appendix F.

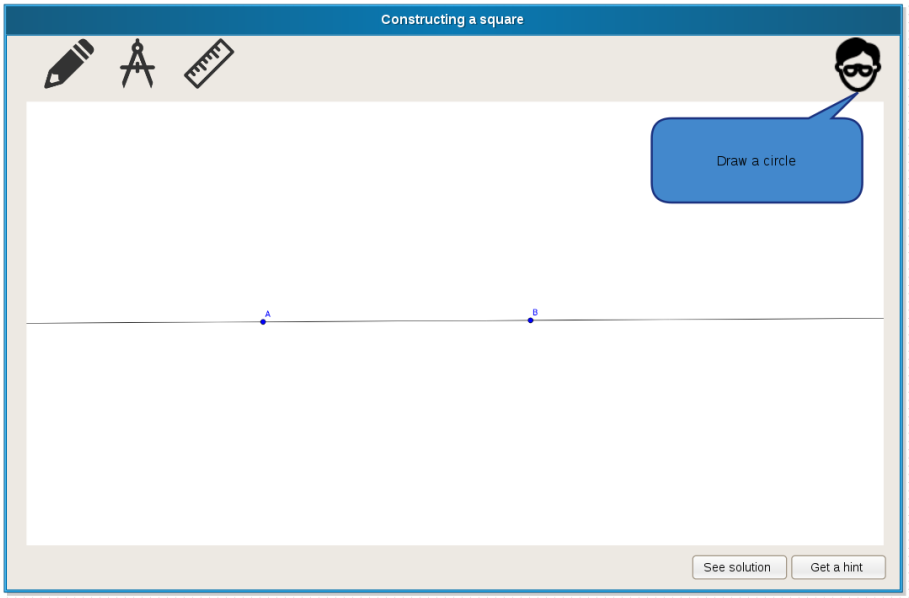


Figure 5: Requesting a feedforward hint.

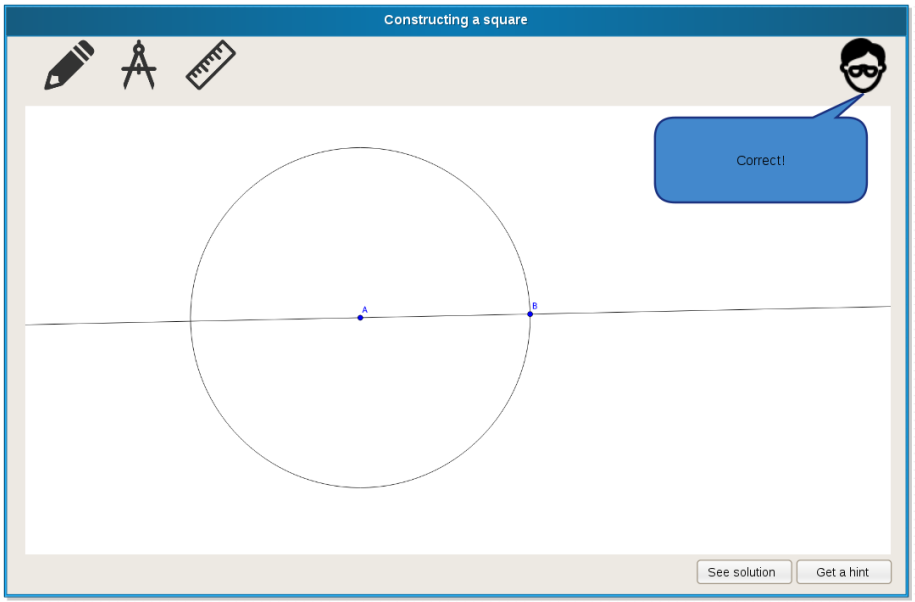


Figure 6: Receiving feedback after a correct step.

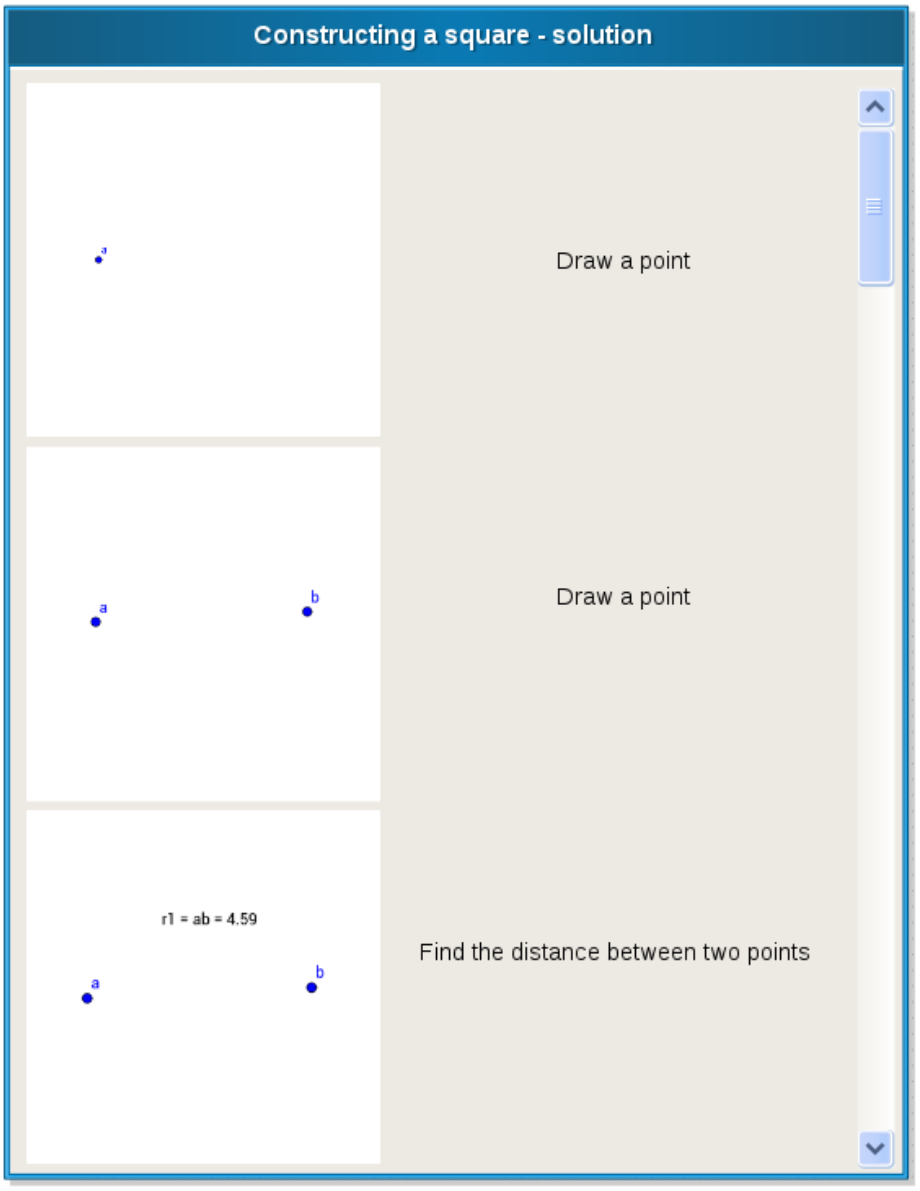


Figure 7: Consulting the solution.

### 4.3.2 Adaptivity of hints

The 'extended library' by Gulwani et al., provides refinement rules (shown in Figure 2) that can be rewritten in the simpler refinement rules of the 'base library'. An example is the function 'PerpendicularBisector2Points' which, given two points, returns a line perpendicular to the line between the two points (see Figure 8). The function 'PerpendicularBisector2Points' can actually be implemented as:

```
perpendicularBisector2Points :: Point -> Point -> Line
perpendicularBisector2Points a b = lineFromPoints d e
    where
        [d, e] = circleCircleXn (Circle a c) (Circle b c)
        c = distance a b
```

Note the use of basic library functions 'distance' (called 'Length' by Gulwani et al.) and 'circleCircleXn'. Since the application of this function is actually equivalent to the application of three simpler functions in terms of the end result (a line perpendicular to a line between 2 points), hints can also be provided for the simpler functions instead of more complex functions. Ideally, the tutor would know which rules a student already masters and which ones are not yet mastered. A hint on the application of a more complex rule can then be avoided depending on a student's knowledge of the rule. This makes the hints adaptive to a student's knowledge.

## 4.4 Use of available program synthesis algorithms

This thesis makes use of the existing domain reasoner by Gulwani et al. to generate strategies. The output of the 'GeoSynth' algorithm by Gulwani et al. is a program consisting of rules. The rules necessary for building a square from two points, have been implemented in Haskell (see Appendix A). A directed acyclic graph between the inputs and outputs of the rules is then built in order to find all paths (topological sorts in the graph) from the propositions to the solution of the exercise. This graph is the strategy. Each path through the graph (from propositions to solution) is a worked-out solution (a solution with all steps leading to it).

### 4.4.1 Integrating an existing algorithm

The GeoSynth algorithm by Gulwani et al. [16] is an example of a program synthesis algorithm that can be integrated with the domain reasoner. The type signature of the GeoSynth algorithm is given:

```
type SpecificationPre = [Object] -> Bool
type SpecificationPost = ([Object], [Object]) -> Bool
type Specification = (SpecificationPre, SpecicationPost)

geoSynth :: Specification -> [LObject] -> Program
geoSynth inputSpec outputSpec inputObjects = ...
```

The Specification type is detailed in Section 3.2. The Haskell implementation of the GeoSynth algorithm is left to the authors of the algorithm, but this signature

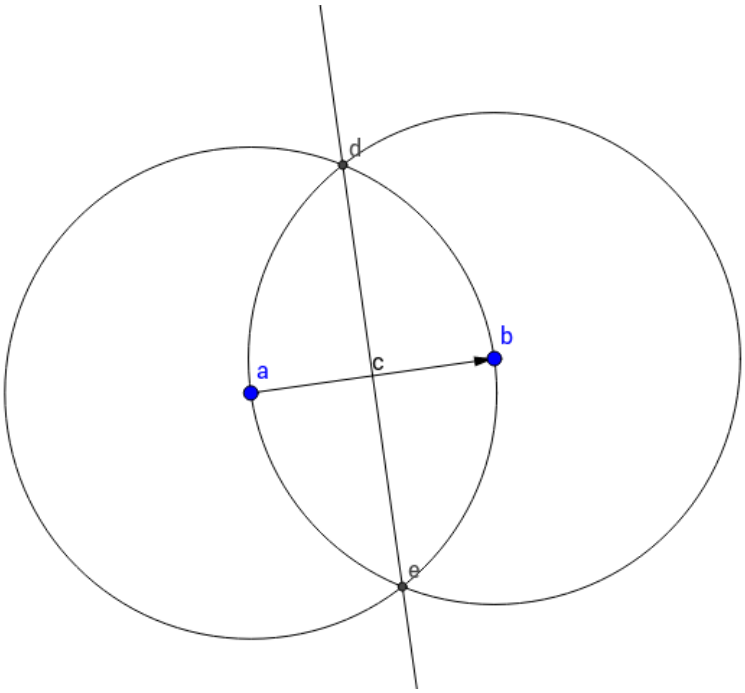


Figure 8: Building 'PerpendicularBisector2Points' with a simple construction.

is enough to show how the algorithm can be integrated. The GeoSynth algorithm returns a program. This program still has to be translated to a strategy:

```
programToStrategy :: Program -> String -> LabeledStrategy Program
programToStrategy (operations, inputObjects) name =
  label name $ dependencyGraph $ graphFromEdges $
  [ (createSpecificationObject l o, l, [])
   | (l, o) <- inputObjects] ++ concatMap makeNodes operations

createSpecificationObject :: Label -> Object -> Rule Program
createSpecificationObject l o = makeRule op $ \(ops, lobjs)
  -> Just ((op, [], [l]):ops, (l, o):lobjs)
    where
      op = "specification"
```

The first function creates a rule from any LabeledObject and the second function creates a strategy from any Program. In this way, a strategy can effectively be built using just an input specification (including labeled objects) and an output specification.

## 4.5 Multiple solutions

A particularity of the square example, is that a second square can be created between points $A$, $F$, $B$ and $G$. This solution is not captured in the strategy generated from the program of Figure 3, since that program does not contain

the rules to draw lines $FB$ and $GB$. A program synthesis algorithm or a human could provide multiple programs that represent different solutions. These programs can then be converted to a strategy that includes all given programs. The ideas framework provides the *choice* function to combine multiple strategies, so that paths through all strategies are accepted. Using the *choice* function the *programsToStrategy* function could be built as follows:

```
programsToStrategy :: [Program] -> String
  -> LabeledStrategy Program
programsToStrategy programs name = label name $
  choice (map (`programToStrategy` "") programs)
```

# 5 Validation

To validate that the correct hints are generated for several different exercises, three of the 25 exercises from Gulwani's paper [16] were selected at random:

1. Construct a regular hexagon inside a circle
2. Construct a triangle given two sides and an included angle
3. Draw arcs of radius $ra$ that are tangent to two given circles

For each of these test cases, a worked-out example, a feedback hint and a feedforward hint is verified. Although the specification of these exercises is not explicitly given in Gulwani's paper, these specifications can be established based on the descriptions of the exercises.

In this section, programs have been established manually based on the description of the exercises. An instance of the specification of the exercise is generated as follows: given coordinates are chosen at random in the interval 0 to 1000 (except for the first point, for which the origin is always chosen), lengths are chosen at random in the interval 1 to 1000 and angles are chosen at random in the interval 0 to $\pi$. Random combinations of numbers are chosen until the exercise constraints are satisfied (the constraints are given in the input specification of the exercise). Although this is a naive way to find random exercise instances, the exercise instances were quickly found by using this method in practice, due to the limited number of constraints.

- For exercise 1, length $r = 267$, point $m = (0, 0)$ and point $p = (145, 66)$ and circle $x = Circle(m, r)$ are given.
- For exercise 2, angle $alpha = 1.95809317826$, length $d_1 = 828$ and length $d_2 = 631$ are given.
- For exercise 3, point $a = (0, 0)$, point $b = (507, 157)$, length $r_1 = 629$, length $r_2 = 142$, length $r_a = 817$, circle $c1 = Circle(a, r_1)$ and $c2 = Circle(b, r_2)$ are given.

## 5.1 Worked-out examples

In order to generate worked-out examples, a program for each exercise was established manually using the functions from Gulwani's basic library [16] and

the
*ConcentricCircle* function from Gulwani's extended library. Since the program is just one solution, the sequences of hints in the worked-out examples do not necessarily match the order of the statements in the program. The programs can be found in Appendix D. The following are worked-out examples that resulted from the domain reasoner:

### 5.1.1 Construct a regular hexagon inside a circle

1. Draw a circle $x$ with radius $r$ around point $m$
2. Construct a line $l$ between point $m$ and point $p$
3. Find the intersection points $a$ and $b$ between line $l$ and circle $x$
4. Draw a circle $y$ with radius $r$ around point $a$
5. Draw a circle $z$ with radius $r$ around point $b$
6. Find the intersection points $c$ and $d$ between circle $x$ and circle $y$
7. Find the intersection points $e$ and $f$ between circle $x$ and circle $z$
8. Construct a line $lh_2$ between point $a$ and point $d$
9. Construct a line $lh_5$ between point $b$ and point $f$
10. Construct a line $lh_1$ between point $c$ and point $a$
11. Construct a line $lh_3$ between point $d$ and point $e$
12. Construct a line $lh_4$ between point $e$ and point $b$
13. Construct a line $lh_6$ between point $f$ and point $c$

The last steps are shown in Figure 9.

### 5.1.2 Construct a triangle given two sides and an included angle

1. Make three points $b$, $a$ and $c$ that define angle *alpha*
2. Draw a circle $x$ with radius $d_1$ around point $a$
3. Draw a circle $y$ with radius $d_2$ around point $a$
4. Construct a line *lab* between point $a$ and point $b$
5. Find the intersection points $d$ and $f$ between line *lab* and circle $x$
6. Construct a line *lac* between point $a$ and point $c$
7. Find the intersection points $e$ and $g$ between line *lac* and circle $y$
8. Construct a line *lde* between point $d$ and point $e$

The last steps are shown in Figure 10.

### 5.1.3 Draw arcs of radius $r_a$ that are tangent to two given circles

1. Draw a circle $c_1$ with radius $r_1$ around point $a$
2. Draw a circle $c_2$ with radius $r_2$ around point $b$
3. Draw a circle $cc_1$ concentric to circle $c_1$ and at distance $r_a$ away from it
4. Draw a circle $cc_2$ concentric to circle $c_2$ and at distance $r_a$ away from it
5. Find the intersection points $c$ and $d$ between circle $cc_1$ and circle $cc_2$
6. Draw a circle $a_1$ with radius $r_a$ around point $c$
7. Draw a circle $a_2$ with radius $r_a$ around point $d$

The last steps are shown in Figure 11.

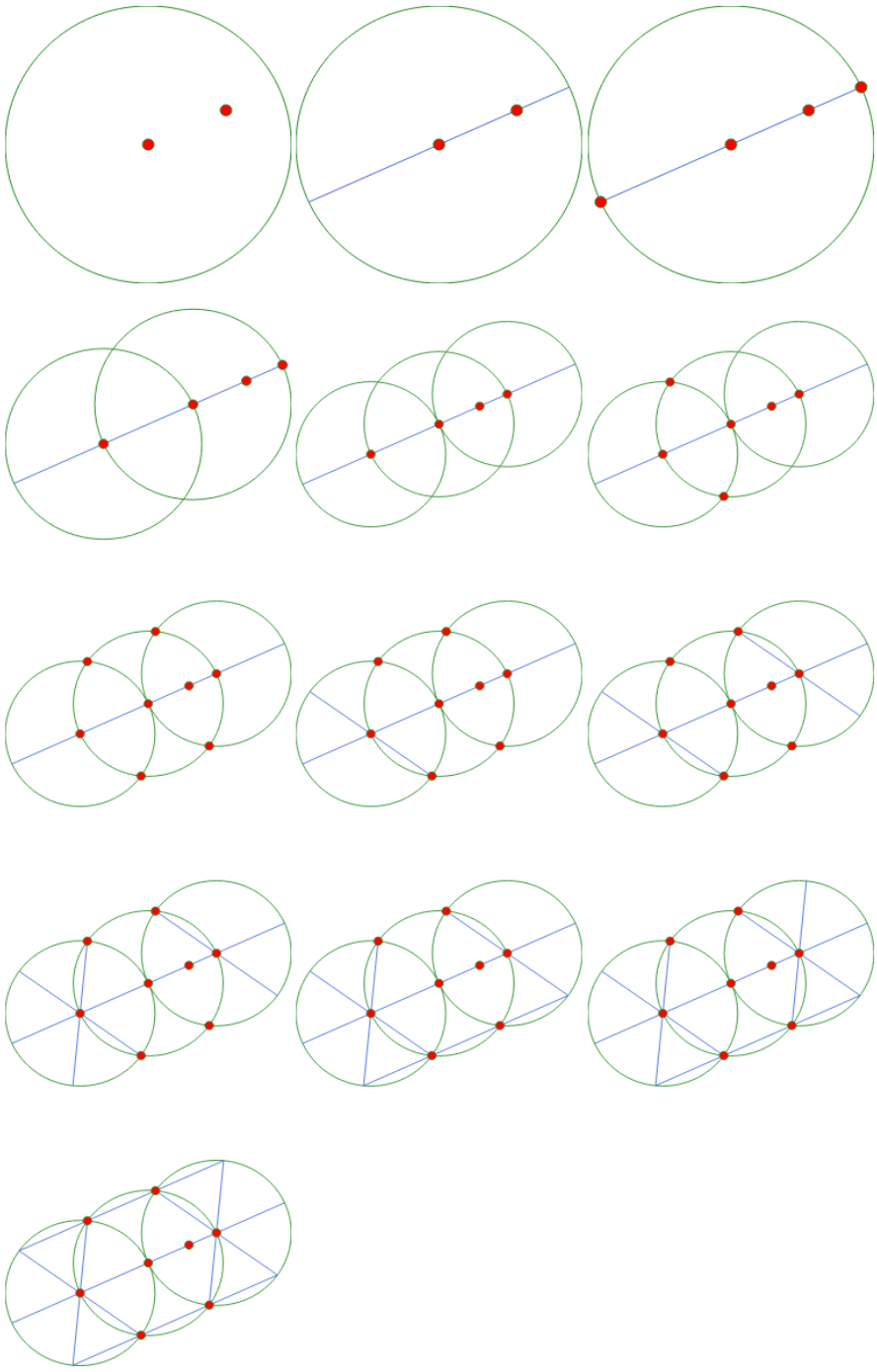

Figure 9: Construction of a regular hexagon inside a circle (worked-out example generated by the domain reasoner)

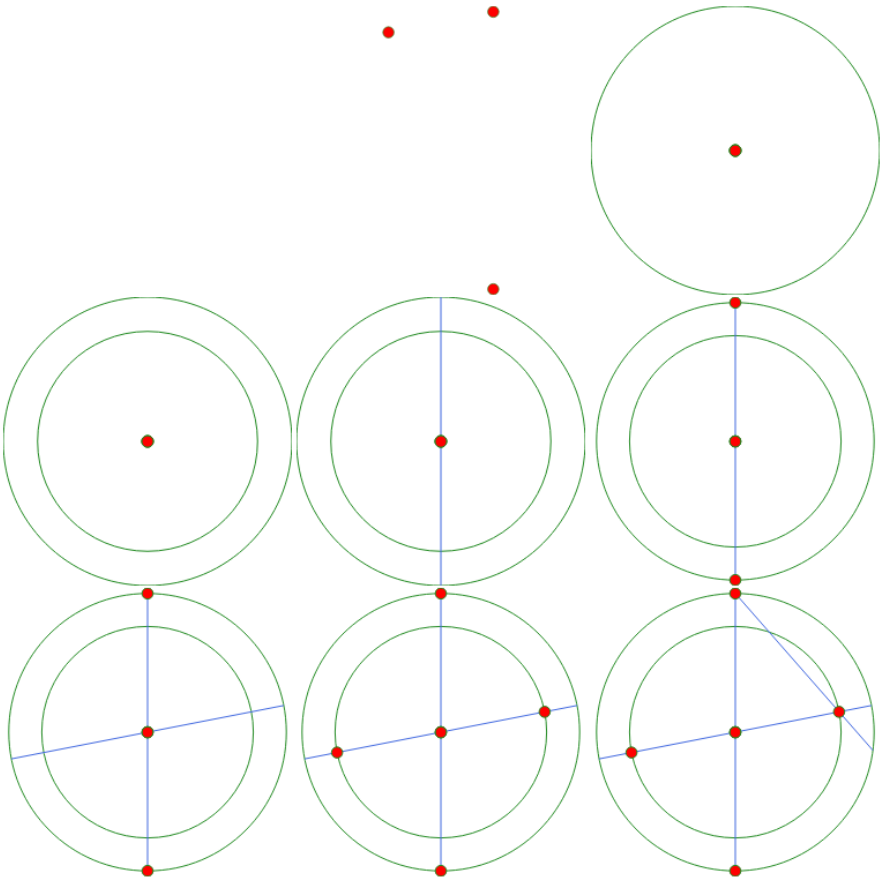

Figure 10: Construction of a triangle given two sides and an included angle (worked-out example generated by the domain reasoner)

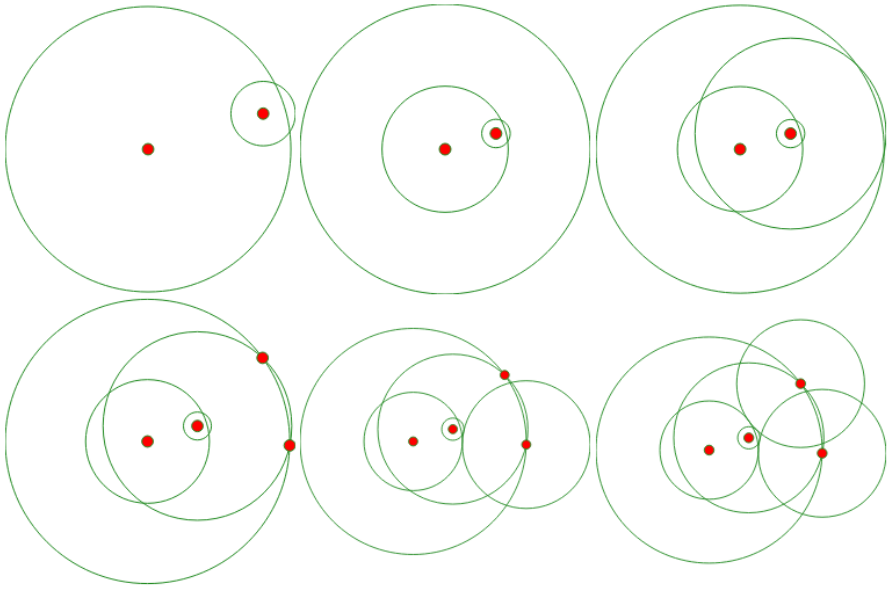

Figure 11: Drawing arcs of radius $r_a$ that are tangent to two given circles (worked-out example generated by the domain reasoner)

## 5.2 Test cases

The three exercises used above, are used to test feedforward and feedback hints as well. The following steps are followed twice for each exercise (once for feedback and once for feedforward):

1. Number the statements the program from 1 to $n$.
2. Choose the number of already executed statements $e$ (in the interval 0 to $n$) randomly.
3. Execute $e$ steps. Choose every applicable rule randomly (generating a random path of length $e$).
4. For the feedback tests: flip a coin to determine whether an incorrect statement is appended to the executed statements (the incorrect statement is arbitrarily chosen). Ensure that there is at least a test for positive feedback and one for negative feedback (in a randomized way).

This resulted in a diverse set of tests covering several situations. For the feedforward hints, the following three situations can be distinguished:

- The student has executed a step (Table 3).
- The student has not executed any steps (Table 4).
- The student has executed all steps (Table 5).

For the feedback hints, a hint on a correct step (see Table 8) and hints on incorrect steps are given (see Tables 6 and 7).

### 5.2.1 Feedforward test cases

The following tables show test cases with the objects/statements given in the exercise in the left column. The steps that the student has executed are in the 'executed statment column and the domain reasoner's suggestion for a next step are in the 'hint' column. The validity of that hint is specified in the last column.

Table 3: Construct a regular hexagon inside a circle

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [m,p,r] = specification()<br>[x] = circle(m,r) | [l] = lineFromPoints(m,p) | Find the intersection points 'a' and 'b' between line 'l' and circle 'x' | Yes |

Table 4: Construct a triangle given two sides and an included angle

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [alpha,d1,d2] = specification() | - | Make three points 'b', 'a' and 'c' that define angle 'alpha' | Yes |

Table 5: Draw arcs of radius $r_a$ that are tangent to two given circles

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [a,b,r1,r2,ra] = specification()<br>[c1] = circle(a,r1)<br>[c2] = circle(b,r2) | [cc2] = concentricCircle(c2,ra)<br>[cc1] = concentricCircle(c1,ra)<br>[c,d] = circleCircleXn(cc1,cc2)<br>[a1] = circle(c,ra)<br>[a2] = circle(d,ra) | Sorry, no hint available. | Yes: the exercise is already solved, so no hint can be given. |

### 5.2.2 Feedback test cases

In the following test cases, the feedback is given on the last step (highligted in bold). In an interactive tutor the hints would be shown immediately after the student executed the step, so that the student is informed of the correctness of the last step.

Table 6: Construct a regular hexagon inside a circle - feedback

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [m,p,r] = specification()<br>[x] = circle(m,r) | [l] = lineFromPoints(m,p)<br>[a,b] = lineCircleXn(l,x)<br>[y] = circle(a,r)<br>[c,d] = circleCircleXn(x,y)<br>[lh2] = lineFromPoints(a,d)<br>[z] = circle(b,r)<br>[lh1] = lineFromPoints(c,a)<br>[e,f] = circleCircleXn(x,z)<br>[lh5] = lineFromPoints(b,f)<br>**[d] = concentricCircle(x,r)** | Incorrect | Yes |

Table 7: Draw arcs of radius $ra$ that are tangent to two given circles - feedback

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [a,b,r1,r2,ra] = specification()<br>[c1] = circle(a,r1)<br>[c2] = circle(b,r2) | [cc1] = concentricCircle(c1,ra)<br>[cc2] = concentricCircle(c2,ra)<br>**[l] = lineFromPoints(a,b)** | Incorrect | Yes |

Table 8: Construct a triangle given two sides and an included angle - feedback

| **Given statements/objects** | **Executed statements** | **Hint** | **Valid?** |
|---|---|---|---|
| [alpha,d1,d2] = specification() | **[b,a,c] = explodeAngle(alpha)** | Correct! | Yes |

# 6 Conclusions and future work

In this thesis, a method for generating hints has been demonstrated using the square construction exercise and validated using three other exercises. It has been assumed that solutions can be found using the GeoSynth algorithm by Gulwani et al. The program synthesis algorithm by Gulwani et al. [16] is very well suited for generating solutions to geometry exercises, which can be used in the geometry domain reasoner. This is demonstrated with the example exercise of constructing a square in Section 4.1.3 and the integration of the GeoSynth algorithm in Section 4.4.1. However, their method might not always find a solution to an exercise when using their 'goodness' measure (and without it, the exhaustive search might not be feasible). This limitation also limits the ability to generate hints for all exercises. Gulwani et al. [16] specifies a list of 25 test cases on which a solution can be found by using the extended library. The square construction exercise in particular, is not part of the 25 test cases that Gulwani et al. has established. It is assumed that GeoSynth is able to find the solution, but there is a possibility that it would not be. This would not be a real threat to the validity of the ability to generate hints using the program synthesis algorithm (sub-question 1), since the method used to generate hints relies on the generic assumption that a program can be represented as a graph, meaning that hints could still be generated for other programs – even those generated in the same format by other program synthesis algorithms.

Adaptive hints, worked examples and feedback can be generated by converting programs generated by a program synthesis algorithm to strategies used in the geometry domain reasoner. A strategy combines rules and these rules can be coupled to hints. From any state in the process of solving an exercise, rules that can be applied are determined using model tracing. A hint for a rule that can be applied from the current state is a feedforward hint. The hints are made adaptive by choosing a next-step rule based on the student's knowledge. There is often a choice in next-step rules, since many complex rules can be constructed using more simple rules, as demonstrated in Section 4.3.2. In this way, granularity of steps in a solution can be varied to provide hints that target specific students (sub-question 3). Worked examples are generated by generating hints from the start (the propositions) of the exercise to a solution. Model tracing also enables feedback hints by evaluating the next-step rules available from a previous state and verifying that the step that the student has taken is part of these next-step rules.

Furthermore, it is possible to translate a specification of an exercise in the form of propositions to a program that can be used in the domain reasoner by making use of the GeoSynth program synthesis algorithm, as shown in Section

4.4.1. A program generated by the algorithm of Gulwani et al. can in turn be transformed into a strategy (demonstrated by the programToStrategy function in Section 4.4.1). Since the program synthesis algorithm is integrated with the domain reasoner by the programToStrategy function, the strategies used in the program synthesis algorithm are effectively used in the domain reasoner (subquestion 2). The strategies produced by the programToStrategy function are used to trace the progress of a student in an exercise (model tracing). This means that hints can be generated from math exercises that are specified in as much detail as in the textbooks. It suffices to translate the textbook description of the exercises in a formal definition using the specification language of Gulwani et al. This thesis therefore presents a method for generating hints based on given propositions describing an exercise and a predicate on the end state. When there are multiple different solutions (including different statements), as in the square example, the strategy generated based on a program found by the GeoSynth program synthesis algorithm might not capture all valid solutions. Future research on program synthesis algorithms able to find multiple solutions is required. It is also possible to manually provide different programs in this case (as demonstrated in Section 4.5).

The hints that are generated suggest the application of a rule, present feedback on the application of a rule or give a complete derivation. Hints can take many more different forms: there could be natural language dialogues between the student and the intelligent tutor [12], hints could be proposed or withheld automatically [20] or the student could be requested to explain the reasoning behind a certain action [1]. Such more advanced hints could be implemented and carefully chosen based on research on the effectiveness of those hints in this specific domain. The effectiveness of the feedforward, feedback and worked-out examples, have not been tested on the specific domain of high school level geometry exercises either, so it would be interesting to do a population study on which these types of hints are empirically tested. Mostow J. and Beck J. [23] suggest a data mining approach for intelligent tutors that could be used to determine the effectiveness of several intelligent tutor implementations that use different types of hints.

The adaptivity of hints is an additional challenge. One form of adaptivity has been shown: alternating between hints for a complex rule and hints for more simple rules depending on a student's knowledge. It is assumed that the intelligent tutor is aware of the knowledge of a student, but a knowledge tracing technique is required to get an approximation of a student's knowledge (see Section 3.1.1). Using this approximation and the adaptive hints, the effectiveness of the intelligent tutor should be evaluated to really know what the effect of the combination of adaptive hints and the knowledge tracer is on a student's performance. There are many other forms of adaptivity in hints as well. It is possible to use a part of an already constructed geometric structure (a subset of constructed objects) so that, for example, hints of the form 'What can you conclude when length AB and length AC are equal?' can be given [21] or to use conversational dialogues between intelligent tutor and student [15].

A related opportunity for future research is in finding re-occurring patterns beyond the ones used in the extended library by Gulwani et al. [16]. This can be done by analyzing the commonality between (or patterns in) a large collection of strategies for exercises. When a complex rule is composed of several simpler rules, a hint can be given for the complex rule or several hints can be given for

the simpler rules. It could also be chosen to start with a hint for a complex rule and provide the possibility to 'drill down' to hints of the more simple rules as presented by Roll et al. [25]. Although it has been demonstrated that hints at the executive level (for the most simple rules) are not as effective as hints for a higher level (linked to more complex rules) [11, 2], further research can be done on the choice of the most appropriate hint at the right moment. This can be done, for example, by knowledge tracing or by fitting the types of hints to a mastery learning program.

Finally, a generator of geometry exercises would make a hint generator even less reliant on exercise-specific input. A semi-automated generator seems to exist (see Alvin et al. [3]). According to the authors, the input figures used in their method could also be generated and this would truly make a digital tutor independent of exercise-specific input.

# Glossary

**algorithm** Strategy followed in calculations.

**Bayesian belief network** Statistical model that represents random variables and their conditional dependencies in a directed acyclic graph.

**declarative knowledge** Knowledge of facts.

**digital mathematics tutor** A tutor for mathematics in the form of a computer program.

**domain** Sphere of activity or knowledge.

**domain reasoner** Program that is able to reason within and give feedback for a specific domain.

**framework** Reusable set of libraries or classes for a software system.

**grammar** A set of rules for producing strings that belong to a formal language.

**hint** An indication for going forward in an exercise or feedback on completed steps.

**model tracing** Model tracing is used for digital tutors in order to relate the steps a student takes with respect to a model of possible steps for an exercise.

**procedural knowledge** Knowledge of functions of declarative and procedural knowledge.

**process mining** The extraction of strategies or rules from event logs.

**recursion** Programming of functions that are defined in terms of themselves.

**rule** Smallest unit of procedural knowledge.
Often represented as an if-statement in the ACT-R theory [4].

**strategy** Expert knowledge captured in the strategy language defined in Heeren et al. [17].

**strategy combinator** Operator on the domain of strategies as defined by Heeren et al. [17].

**XML** EXtensible Markup Language - a language that was designed to be both machine- and human-readable.

## References


[1] V.A.W.M.M. Aleven and K.R. Koedinger. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive science*, 26(2):147–179, 2002.

[2] V.A.W.M.M. Aleven, B. McLaren, I. Roll, and K. Koedinger. Toward meta-cognitive tutoring: A model of help seeking with a cognitive tutor. *International Journal of Artificial Intelligence in Education*, 16(2):101–128, 2006.

[3] C. Alvin, S. Gulwani, R. Majumdar, and S. Mukhopadhyay. Synthesis of geometry proof problems. In *AAAI*, pages 245–252, 2014.

[4] J.R. Anderson. Act: A simple theory of complex cognition. *American Psychologist*, 51(4):355, 1996.

[5] J.R. Anderson, J.M. Fincham, and S. Douglass. The role of examples and rules in the acquisition of a cognitive skill. *Journal of experimental psychology: learning, memory, and cognition*, 23(4):932, 1997.

[6] R.K. Atkinson, S.J. Derry, A. Renkl, and D. Wortham. Learning from examples: Instructional principles from the worked examples research. *Review of educational research*, 70(2):181–214, 2000.

[7] R.S.J.d. Baker, A.T. Corbett, and V. Aleven. More accurate student modeling through contextual estimation of slip and guess probabilities in Bayesian knowledge tracing. In Beverley P. Woolf, Esma Aïmeur, Roger Nkambou, and Susanne Lajoie, editors, *Intelligent Tutoring Systems*, volume 5091 of *Lecture Notes in Computer Science*, pages 406–415. Springer Berlin Heidelberg, 2008.

[8] W.M. Carroll. Using worked examples as an instructional support in the algebra classroom. *Journal of Educational Psychology*, 86(3):360, 1994.

[9] A.C.K. Cheung and R.E. Slavin. The effectiveness of educational technology applications for enhancing mathematics achievement in k-12 classrooms: A meta-analysis. *Educational Research Review*, 9:88–113, 2013.

[10] A.T. Corbett and J.R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, 4(4):253–278, 1994.

[11] S. Dutke and T. Reimer. Evaluation of two types of online help for application software. *Journal of computer assisted learning*, 16(4):307–315, 2000.

[12] M.W. Evens, R. Chang, Y.H. Lee, L.S. Shim, C.W. Woo, Y. Zhang, J.A. Michael, and A.A. Rovick. Circsim-tutor: An intelligent tutoring system using natural language dialogue. In *Proceedings of the fifth conference on Applied natural language processing: Descriptions of system demonstrations and videos*, pages 13–14. Association for Computational Linguistics, 1997.

[13] H. Gelernter. Computers & thought. chapter 'Realization of a Geometry-theorem Proving Machine', pages 134–152. MIT Press, Cambridge, MA, USA, 1995.

[14] G. Goguadze, A.G. Palomo, and E. Melis. Interactivity of exercises in activemath. In *ICCE*, pages 109–115, 2005.

[15] A.C. Graesser, K. VanLehn, C.P. Rosé, P.W. Jordan, and D. Harter. Intelligent tutoring systems with conversational dialogue. *AI magazine*, 22(4):39, 2001.

[16] S. Gulwani, V.A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. *SIGPLAN Not.*, 46(6):50–61, June 2011.

[17] B. Heeren and J. Jeuring. Feedback services for stepwise exercises. *Science of Computer Programming*, 88:110–129, 2014. Software Development Concerns in the e-Learning Domain.

[18] M. Hohenwarter and K. Jones. BSRLM geometry working group: ways of linking geometry and algebra, the case of geogebra. *Proceedings of the British Society for Research into Learning Mathematics*, 27(3):126–131, 2007.

[19] H. Keuning, B. Heeren, and J. Jeuring. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*, pages 43–54. ACM, 2014.

[20] K.R. Koedinger and V.A.W.M.M. Aleven. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review*, 19(3):239–264, 2007.

[21] N. Matsuda and K. VanLehn. Modeling hinting strategies for geometry theorem proving. In *International Conference on User Modeling*, pages 373–377. Springer, 2003.

[22] N. Matsuda and K. VanLehn. Gramy: A geometry theorem prover capable of construction. *Journal of Automated Reasoning*, 32(1):3–33, 2004.

[23] J. Mostow and J. Beck. Some useful tactics to modify, map and mine data from intelligent tutors. *Natural Language Engineering*, 12(02):195–208, 2006.

[24] Z.A. Pardos and N.T. Heffernan. Modeling individualization in a Bayesian networks implementation of knowledge tracing. In Paul De Bra, Alfred Kobsa, and David Chin, editors, *User Modeling, Adaptation, and Personalization*, volume 6075 of *Lecture Notes in Computer Science*, pages 255–266. Springer Berlin Heidelberg, 2010.

[25] I. Roll, V.A.W.M.M. Aleven, B.M. McLaren, and K.R. Koedinger. Improving students' help-seeking skills using metacognitive feedback in an intelligent tutoring system. *Learning and Instruction*, 21(2):267–280, 2011.

[26] ITU (International Telegraph Union). The world in 2013 - ict facts and figures. `http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013-e.pdf`, 2013.

[27] K. VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011.

# Appendices

## A Implementation of library functions

```
module GeoLibrary ( lineLineXn, lineCircleXn, circleCircleXn,
  explodeAngle,  perpendicularBisector2Points, parallelLine,
  parallelLineGivenLength, perpendicularToLineThruPoint,
  mirrorPointLine, concentricCircle, midpointGiven2Points,
  angularBisectorLines, distance, lineFromPoints )
where

import Data.Maybe (fromJust)
import DataTypes

infixl 6 |+|
(|+|) :: Vector -> Vector -> Vector
Vector a b |+| Vector c d = Vector (a+c) (b+d)

infixl 8 |.|
(|.|) :: Vector -> Vector -> Double
(Vector a b) |.| (Vector c d) = a * c + b * d

infixl 7 |*|
(|*|) :: Double -> Vector -> Vector
c |*| Vector a b = Vector (c*a) (c*b)

infixl 6 |-|
(|-|) :: Vector -> Vector -> Vector
a |-| b = a |+| ((-1) |*| b)

p2v :: Point -> Vector
p2v (Point px py) = Vector px py

v2p :: Vector -> Point
v2p (Vector vx vy) = Point vx vy

vectorLength :: Vector -> Double
vectorLength (Vector a b) = sqrt (a^2 + b^2)

normalizeVector :: Vector -> Vector
normalizeVector v = (1 / vectorLength v) |*| v

evaluateLine :: Line -> Length -> Point
evaluateLine (Line p v) l =
  v2p (p2v p |+| l |*| normalizeVector v)

lineLineXn :: Line -> Line -> Maybe Point
lineLineXn (Line p1 (Vector xx xy)) (Line p2 y)
  | yd /= 0 =
```

```
46      Just (v2p (p2v p2 |+| ((p2v p1 |-| p2v p2) |.| d / yd) |*| y))
47    | otherwise = Nothing
48    where
49      yd = y |.| d
50      d = Vector (-xy) xx
51
52  lineCircleXn :: Line -> Circle -> [Point]
53  lineCircleXn (Line a b) (Circle c r)
54      | e == 0 = [evaluateLine (Line a b) (d / f)]
55      | e > 0 = [evaluateLine (Line a b) ((d - e) / f) ,
56                 evaluateLine (Line a b) ((d + e) / f)]
57      | otherwise = []
58      where
59          f = 2
60          (d, e) = (((-2) |*| (p2v a |-| p2v c)) |.| bn,
61                    sqrt (((2 |*| (p2v a |-| p2v c)) |.| bn)^2 - 4
62                    * (vectorLength (p2v a |-| p2v c)^2 - r^2)))
63          bn = normalizeVector b
64
65
66  circleCircleXn :: Circle -> Circle -> [Point]
67  circleCircleXn (Circle a ra) (Circle b rb) =
68      lineCircleXn radicalAxisLine (Circle a ra)
69          where
70              radicalAxisLine =
71                Line radicalAxisPoint (Vector ((-1)*aby) abx)
72              Vector abx aby = p2v b |-| p2v a
73              radicalAxisPoint = v2p (p2v a
74                |+| radicalAxisDistance
75                  |*| normalizeVector (p2v b |-| p2v a))
76              radicalAxisDistance = (d + (ra^2 - rb^2) / d) / 2
77              d = vectorLength (p2v b |-| p2v a)
78
79  explodeAngle :: Angle -> [Point]
80  explodeAngle (Angle a) =
81    [Point 0 0, Point 0 1, Point (cos a) (sin a)]
82
83  perpendicularBisector2Points :: Point -> Point -> Line
84  perpendicularBisector2Points a b = lineFromPoints d e
85      where
86          [d, e] = circleCircleXn (Circle a c) (Circle b c)
87          c = distance a b
88
89  parallelLine :: Line -> Point -> Line
90  parallelLine (Line a t) c
91      | abs ((p2v g |-| p2v c) |.|  t)
92        > abs ((p2v h |-| p2v c) |.| t) = Line c (p2v g |-| p2v c)
93      | otherwise = Line c (p2v h |-| p2v c)
94      where
95          [g, h] = circleCircleXn x y
```

```
96          y = Circle f (vectorLength (p2v d |-| p2v e))
97          (d:l1, e:l2, f:l3) = (lineCircleXn (Line a t) z,
98            lineCircleXn l z, lineCircleXn l x)
99          l = Line a (p2v c |-| p2v a)
100         z = Circle a 1
101         x = Circle c 1
102
103 parallelLineGivenLength :: Line -> Length -> [Line]
104 parallelLineGivenLength l r = [parallelLine l c, parallelLine l d]
105     where
106         [c, d] = lineCircleXn f x
107         x = Circle (fromJust e) r
108         e = lineLineXn f l
109         f = perpendicularBisector2Points
110           (evaluateLine l 0) (evaluateLine l 1)
111
112 perpendicularToLineThruPoint :: Line -> Point -> Line
113 perpendicularToLineThruPoint l = parallelLine m
114     where
115         m = perpendicularBisector2Points
116           (evaluateLine l 0) (evaluateLine l 1)
117
118 mirrorPointLine :: Point -> Line -> Point
119 mirrorPointLine a l = v2p (p2v a |+| 2 |*| (p2v b |-| p2v a))
120     where
121         b = fromJust
122           (lineLineXn (perpendicularToLineThruPoint l a) l)
123
124 concentricCircle :: Circle -> Length -> Circle
125 concentricCircle (Circle m r) o = Circle m (r + o)
126
127 midpointGiven2Points :: Point -> Point -> Point
128 midpointGiven2Points a b = v2p ((1/2) |*| (p2v a |+| p2v b))
129
130 angularBisectorLines :: Line -> Line -> [Line]
131 angularBisectorLines (Line a d1) (Line b d2) = [Line c d3,
132                                                 Line c d4]
133     where
134         c = fromJust (lineLineXn (Line a d1) (Line b d2))
135         d3 = normalizeVector (d1 |+| d2)
136         d4 = normalizeVector (d1 |-| d2)
137
138 distance :: Point -> Point -> Length
139 distance p1 p2 = vectorLength (p2v p2 |-| p2v p1)
140
141 lineFromPoints :: Point -> Point -> Line
142 lineFromPoints p1 p2 = Line p1 (p2v p2 |-| p2v p1)
```

# B Implementation of conversion from library functions to rules

```
module Rules (createRule, createSpecification, getRuleIds)
where
import Control.Applicative
import Data.List (intercalate, sort, sortBy, sortOn)
import Data.Maybe (fromJust, isJust, isNothing, fromMaybe)
import DataTypes
import GeoLibrary
import Ideas.Common.Id (describe)
import Ideas.Common.Rule.Abstract (Rule, makeRule)

objLineLineXn :: Refinement
objLineLineXn [LineObject l1, LineObject l2] = Just $ maybe []
  (replicate 1 . PointObject) (lineLineXn l1 l2)
objLineLineXn _ = Nothing

objLineCircleXn :: Refinement
objLineCircleXn [LineObject l, CircleObject c] =
  Just $ map PointObject (lineCircleXn l c)
objLineCircleXn _ = Nothing

objCircleCircleXn :: Refinement
objCircleCircleXn [CircleObject c1, CircleObject c2] =
  Just $ map PointObject (circleCircleXn c1 c2)
objCircleCircleXn _ = Nothing

objPerpendicularToLineThruPoint :: Refinement
objPerpendicularToLineThruPoint [LineObject l, PointObject p] =
  Just [LineObject $ perpendicularToLineThruPoint l p]
objPerpendicularToLineThruPoint _ = Nothing

objPerpendicularBisector2Points :: Refinement
objPerpendicularBisector2Points [PointObject p1, PointObject p2] =
  Just [LineObject $ perpendicularBisector2Points p1 p2]
objPerpendicularBisector2Points _ = Nothing

objDistance :: Refinement
objDistance [PointObject p1, PointObject p2] =
  Just [LengthObject $ distance p1 p2]
objDistance _ = Nothing

objLineFromPoints :: Refinement
objLineFromPoints [PointObject p1, PointObject p2] =
  Just [LineObject $ lineFromPoints p1 p2]
objLineFromPoints _ = Nothing

objCircle :: Refinement
```

```haskell
objCircle [PointObject p, LengthObject d] =
  Just [CircleObject $ Circle p d]
objCircle _ = Nothing

objConcentricCircle :: Refinement
objConcentricCircle [CircleObject c, LengthObject o] =
  Just [CircleObject $ concentricCircle c o]

objExplodeAngle :: Refinement
objExplodeAngle [AngleObject a] =
  Just . map PointObject $ explodeAngle a
objExplodeAngle _ = Nothing

selectObjects :: [ObjectLabel] -> [LObject] -> Maybe [Object]
selectObjects (l:ls) e = do
    o <- lookup l e
    os <- selectObjects ls e
    return (o:os)
selectObjects [] e = Just []

execute :: Operator -> Refinement
execute o = fromMaybe (const Nothing) (lookup o [
    ("lineLineXn", objLineLineXn),
    ("lineCircleXn", objLineCircleXn),
    ("circleCircleXn", objCircleCircleXn),
    ("perpendicularToLineThruPoint",
      objPerpendicularToLineThruPoint),
    ("distance", objDistance),
    ("lineFromPoints", objLineFromPoints),
    ("circle", objCircle),
    ("perpendicularBisector2Points",
      objPerpendicularBisector2Points),
    ("explodeAngle", objExplodeAngle),
    ("concentricCircle", objConcentricCircle)
  ])

makeRuleId :: Operator -> [ObjectLabel] -> [ObjectLabel] -> String
makeRuleId op is os = op ++ "--" ++ intercalate "-" is
  ++ "--" ++ intercalate "-" os

getRuleIds :: Program -> [String]
getRuleIds (operations, _) =
  map (\(_, o, _, is, os) -> makeRuleId o is os) operations

createRule :: Operation -> Rule Program
createRule (l, op, ols, i, o) =
  makeRule (makeRuleId op i o) $ \(ops, lobjs) -> do
    inObjects <- selectObjects i lobjs
    outObjects <- execute op inObjects
    return ((l, op, ols, i, o):ops, zip o outObjects ++ lobjs)
```

```
97
98  createSpecification :: OperationLabel -> [LObject] -> Rule Program
99  createSpecification l lobjs =
100   makeRule op $ \_ ->
101     Just ([(l, op, [], [], map fst lobjs)], lobjs)
102       where
103         op = "specification"
```

# C Implementation of strategy generation

```
1   module Strategies where
2   import Data.Graph
3   import DataTypes
4   import Ideas.Common.Context hiding (Context)
5   import Ideas.Common.Library hiding (Context)
6   import Ideas.Common.Strategy.Abstract
7   import Ideas.Common.Strategy.Combinators hiding (repeat)
8   import Rules
9
10
11  makeNode :: [LObject] -> Operation
12    -> (Rule Program, OperationLabel, [OperationLabel])
13  makeNode spec o@(l, _, is, _, _)
14    | l == 0 = (createSpecification l spec, l, [])
15    | otherwise = (createRule o, l, is)
16
17  geoSynth :: Specification -> Specification -> [LObject] -> Program
18  geoSynth inputSpec outputSpec inputObjects = ...
19
20  -- This function only works if no renaming of labeled objects took
21  -- place.
22  -- For example: if a point is labeled X,
23  -- the label X cannot be used
24  -- for a different point later on.
25  programToStrategy :: Program -> String -> LabeledStrategy Program
26  programToStrategy (operations, inputObjects) name =
27    label name $ dependencyGraph $ graphFromEdges $
28      map (makeNode inputObjects) operations
29
30  programsToStrategy :: [Program] -> String
31    -> LabeledStrategy Program
32  programsToStrategy programs name =
33    label name $ choice (map (`programToStrategy` "") programs)
```

# D Implementation of geometry programs

```
1   module Programs where
2   import DataTypes
```

```haskell
import qualified Data.Map as Map
import Data.List (nub)
import Data.Maybe (fromJust)

type Statement = (Operator, [ObjectLabel], [ObjectLabel])
type LabeledStatement = (OperationLabel, Statement)

numberStatements :: [Statement] -> [LabeledStatement]
numberStatements = zip [0..]

getDefinitions :: [LabeledStatement]
  -> Map.Map ObjectLabel OperationLabel
getDefinitions [] = Map.empty
getDefinitions ((i, (op, is, os)):xs) = Map.union
  (Map.fromList (zip os (repeat i))) (getDefinitions xs)

createOperation ::
  (LabeledStatement, Map.Map ObjectLabel OperationLabel)
  -> Operation
createOperation ((i, (op, is, os)), m) = (i, op, nub $
  map (fromJust . flip Map.lookup m) is, is, os)

createOperations :: [Statement] -> [Operation]
createOperations stmts = map createOperation statementsWithDefs
  where
    labeledStatements = numberStatements stmts
    definitions = [getDefinitions (take i labeledStatements)|
                   i <- [0..length labeledStatements]]
    statementsWithDefs = zip labeledStatements definitions

createProgram :: [LObject] -> [Statement] -> Program
createProgram spec stmts = (
  createOperations (("specification", [], map fst spec):stmts),
  spec
  )

square :: Program
square = createProgram
    [ ("a", PointObject $ Point 0 0)
    , ("b", PointObject $ Point 300 0)
    ]
    -- Elements above are given
    [ ("distance", ["a","b"], ["r1"])
    , ("circle", ["a", "r1"], ["x"])
    , ("circle", ["b", "r1"], ["y"])
    , ("circleCircleXn", ["x", "y"], ["c", "d"])
    , ("lineFromPoints", ["c", "d"], ["l1"])
    , ("lineFromPoints", ["a", "b"], ["l2"])
    , ("lineLineXn", ["l1", "l2"], ["e"])
    , ("distance", ["a", "e"], ["r2"])
```

```
      , ("circle", ["e", "r2"], ["z"])
      , ("lineCircleXn", ["l1", "z"], ["f", "g"])
      , ("lineFromPoints", ["a", "f"], ["l3"])
      , ("lineCircleXn", ["l3", "x"], ["k", "i"])
      , ("lineFromPoints", ["a", "g"], ["l4"])
      , ("lineCircleXn", ["l4", "x"], ["h", "j"])
      , ("lineFromPoints", ["i", "j"], ["l5"])
      , ("lineFromPoints", ["j", "k"], ["l6"])
      , ("lineFromPoints", ["k", "h"], ["l7"])
      , ("lineFromPoints", ["h", "i"], ["l8"])
      ]

hexagon :: Program
hexagon = createProgram
  [ ("m", PointObject $ Point 0 0)
  , ("p", PointObject $ Point 145 66)
  , ("r", LengthObject 267)
  ]
  [ ("circle", ["m", "r"], ["X"])
  -- Elements above are given
  , ("lineFromPoints", ["m", "p"], ["l"])
  , ("lineCircleXn", ["l", "X"], ["a", "b"])
  , ("circle", ["a", "r"], ["Y"])
  , ("circle", ["b", "r"], ["Z"])
  , ("circleCircleXn", ["X", "Y"], ["c", "d"])
  , ("circleCircleXn", ["X", "Z"], ["e", "f"])
  , ("lineFromPoints", ["c", "a"], ["lh1"])
  , ("lineFromPoints", ["a", "d"], ["lh2"])
  , ("lineFromPoints", ["d", "e"], ["lh3"])
  , ("lineFromPoints", ["e", "b"], ["lh4"])
  , ("lineFromPoints", ["b", "f"], ["lh5"])
  , ("lineFromPoints", ["f", "c"], ["lh6"])
  ]

triangleGiven2SidesAndIncludedAngle :: Program
triangleGiven2SidesAndIncludedAngle = createProgram
  [ ("alpha", AngleObject $ Angle 1.95809317826)
  , ("d1", LengthObject 828)
  , ("d2", LengthObject 631)
  ]
  -- Elements above are given
  [ ("explodeAngle", ["alpha"], ["b", "a", "c"])
  , ("lineFromPoints", ["a", "b"], ["lab"])
  , ("lineFromPoints", ["a", "c"], ["lac"])
  , ("circle", ["a", "d1"], ["X"])
  , ("circle", ["a", "d2"], ["Y"])
  , ("lineCircleXn", ["lab", "X"], ["d", "f"])
  , ("lineCircleXn", ["lac", "Y"], ["e", "g"])
  , ("lineFromPoints", ["d", "e"], ["lde"])
  ]
```

```haskell

tangentArcsToTwoCircles :: Program
tangentArcsToTwoCircles = createProgram
  [ ("a", PointObject $ Point 0 0)
  , ("b", PointObject $ Point 507 157)
  , ("r1", LengthObject 629)
  , ("r2", LengthObject 142)
  , ("ra", LengthObject 817)
  ]
  [ ("circle", ["a", "r1"], ["c1"])
  , ("circle", ["b", "r2"], ["c2"])
  -- Elements above are given
  , ("concentricCircle", ["c1", "ra"], ["cc1"])
  , ("concentricCircle", ["c2", "ra"], ["cc2"])
  , ("circleCircleXn", ["cc1", "cc2"], ["c", "d"])
  , ("circle", ["c", "ra"], ["a1"])
  , ("circle", ["d", "ra"], ["a2"])
  ]
```

# E Implementation of hint generation

```
supports eval.square
feedback same = @ok
feedback noteq = @incorrect
feedback unknown = @incorrect
feedback ok = {Correct!}
feedback buggy = {}
feedback detour = {}
feedback wrongrule = {}
feedback hint = {}
feedback step = @expected
feedback label = {}

string incorrect = Incorrect

text specification----a-b = {}
text distance--a-b--r1 =
  {Find the distance 'r1' between point 'a' and point 'b'}
text circle--a-r1--x =
  {Draw a circle 'x' with radius 'r1' around point 'a'}
text circle--b-r1--y =
  {Draw a circle 'y' with radius 'r1' around point 'b'}
text circlecirclexn--x-y--c-d =
  {Find the intersection points 'c' and 'd' between circle 'x'
   and circle 'y'}
text linefrompoints--c-d--l1 =
  {Construct a line 'l1' between point 'c' and point 'd'}
text linefrompoints--a-b--l2 =
  {Construct a line 'l2' between point 'a' and point 'b'}
```

```
text linelinexn--l1-l2--e =
  {Find the intersection point 'e' between line l1 and line 'l2'}
text distance--a-e--r2 =
  {Find the distance 'r2' between point 'a' and point 'e'}
text circle--e-r2--z =
  {Draw a circle 'z' with radius 'r2' around point 'e'}
text linecirclexn--l1-z--f-g =
  {Find the intersection points 'f' and 'g'
   between line 'l1' and circle 'z'}
text linefrompoints--a-f--l3 =
  {Construct a line 'l3' between point 'a' and point 'f'}
text linecirclexn--l3-x--k-i =
  {Find the intersection points 'k' and 'i'
   between line 'l3' and circle 'x'}
text linefrompoints--a-g--l4 =
  {Construct a line 'l4' between point 'a' and point 'g'}
text linecirclexn--l4-x--h-j =
  {Find the intersection points 'h' and 'j'
   between line 'l4' and circle 'x'}
text linefrompoints--i-j--l5 =
  {Construct a line 'l5' between point 'i' and point 'j'}
text linefrompoints--j-k--l6 =
  {Construct a line 'l6' between point 'j' and point 'k'}
text linefrompoints--k-h--l7 =
  {Construct a line 'l7' between point 'k' and point 'h'}
text linefrompoints--h-i--l8 =
  {Construct a line 'l8' between point 'h' and point 'i'}
```

# F Example requests and responses

## F.1 Feedforward

Request:

```
<request exerciseid="eval.square" service="textual.onefirsttext"
         encoding="string" source="testxml">
    <state>
        <prefix>
            [0, 0, 0, 2]
        </prefix>
        <expr>
            ([
                (6,"lineFromPoints",[0],["a","b"],["l2"]),
                (1,"distance",[0],["a","b"],["r1"]),
                (0,"specification",[],[],["a","b"])
            ],[
                ("l2",LineObject
                  (Line (Point 0.0 0.0) (Vector 0.0 300.0))),
                ("r1",LengthObject 300.0),
                  ("a",PointObject (Point 0.0 0.0)),
```

```
                ("b",PointObject (Point 0.0 300.0))
            ])
        </expr>
    </state>
</request>
```

Response:

```
<reply result="ok" version="1.4 (8775)">
  <message>
    Draw a circle 'x' with radius 'r1' around point 'a'
  </message>
  <state>
    <prefix>
      [0,0,0,2,0]
    </prefix>
    <expr>
      ([
        (2,"circle",[0,1],["a","r1"],["x"]),
        (6,"lineFromPoints",[0],["a","b"],["l2"]),
        (1,"distance",[0],["a","b"],["r1"]),
        (0,"specification",[],[],["a","b"])
      ],[
        ("x",CircleObject (Circle (Point 0.0 0.0) 300.0)),
        ("l2",LineObject
          (Line (Point 0.0 0.0) (Vector 0.0 300.0))),
        ("r1",LengthObject 300.0),
        ("a",PointObject (Point 0.0 0.0)),
        ("b",PointObject (Point 0.0 300.0))
      ])
    </expr>
  </state>
</reply>
```

## F.2 Feedback

Request:

```
<request exerciseid="eval.square" service="textual.feedbacktext"
         encoding="string" source="testxml">
    <state>
        <prefix>
          [0, 0, 0]
        </prefix>
        <expr>
            ([
              (1,"distance",[0],["a","b"],["r1"]),
              (0,"specification",[],[],["a","b"])
            ],[
              ("r1",LengthObject 300.0),
              ("a",PointObject (Point 0.0 0.0)),
```

```
            ("b",PointObject (Point 0.0 300.0))
          ])
      </expr>
  </state>
  <expr>
      ([
        (2,"circle",[0,1],["a","r1"],["x"]),
        (1,"distance",[0],["a","b"],["r1"]),
        (0,"specification",[],[],["a","b"])
      ],[
        ("x",CircleObject (Circle (Point 0.0 0.0) 300.0)),
        ("r1",LengthObject 300.0),
        ("a",PointObject (Point 0.0 0.0)),
        ("b",PointObject (Point 0.0 300.0))
      ])
  </expr>
</request>
```

Response:

```
<reply result="ok" version="1.4 (8775)">
  <message accept="true">
    Correct!
  </message>
  <state>
    <prefix>
      [0,0,0,0]
    </prefix>
    <expr>
      ([
        (2,"circle",[0,1],["a","r1"],["x"]),
        (1,"distance",[0],["a","b"],["r1"]),
        (0,"specification",[],[],["a","b"])
      ],[
        ("x",CircleObject (Circle (Point 0.0 0.0) 300.0)),
        ("r1",LengthObject 300.0),
        ("a",PointObject (Point 0.0 0.0)),
        ("b",PointObject (Point 0.0 300.0))
      ])
    </expr>
  </state>
</reply>
```

## F.3 Worked-out example

Request:

```
<request exerciseid="eval.square" service="textual.derivationtext"
         encoding="string" source="testxml">
    <state>
        <prefix>
```

```
            []
        </prefix>
        <expr>
            ([], [])
        </expr>
    </state>
</request>
```

Response (shortened):

```
<reply result="ok" version="1.4 (8775)">
  <list>
    <elem ruletext="specification">
      <expr>
        ([
          (0,"specification",[],[],["a","b"])
        ],[
          ("a",PointObject (Point 0.0 0.0)),
          ("b",PointObject (Point 0.0 300.0))
        ])
      </expr>
    </elem>
    <elem ruletext="Find the distance 'r1' between
      point 'a' and point 'b'">
      <expr>
        ([
          (1,"distance",[0],["a","b"],["r1"]),
          (0,"specification",[],[],["a","b"])
        ],[
          ("r1",LengthObject 300.0),
          ("a",PointObject (Point 0.0 0.0)),
          ("b",PointObject (Point 0.0 300.0))
        ])
      </expr>
    </elem>
    ...
  </list>
</reply>
```