Formal Representation and Automated Transformation of Geometric Statements

Xiaoyu Chen

Beihang University, Beijing, China

July 24, 2010

Outline

Start with Geometry Software

Geometry problems (drawing or proving) are specified by applying similar (or same) concepts which are implemented differently in these systems.

Table: Constructive style

Cinderella	GeoGebra
Perpendicular(a;A)	PerpendicularLine[A,a]
Circumcircle(A;B;C)	Circle[A,B,C]
AngleBisector(m;n;A)	AngleBisector[m,n]

Start with Geometry Software

Geometry problems (drawing or proving) are specified by applying similar (or same) concepts which are implemented differently in these systems.

Table: Constructive style

Cinderella	GeoGebra
Perpendicular(a;A)	PerpendicularLine[A,a]
Circumcircle(A;B;C)	Circle[A,B,C]
AngleBisector(m;n;A)	AngleBisector[m,n]

Table: Constraint style

GEOTHER	GeoProof
midpoint(A,B,C)	is_midpoint C A B
parallel(A,B,C,D)	parallel A B C D

Start with Geometry Software

Geometry problems (drawing or proving) are specified by applying similar (or same) concepts which are implemented differently in these systems.

Table: Constructive style

Cinderella	GeoGebra
Perpendicular(a;A)	PerpendicularLine[A,a]
Circumcircle(A;B;C)	Circle[A,B,C]
AngleBisector(m;n;A)	AngleBisector[m,n]

Table: Constraint style

GEOTHER	GeoProof
midpoint(A,B,C)	is_midpoint C A B
parallel(A,B,C,D)	parallel A B C D

The constructions and predicates can be viewed as concepts.

Standardizing Problem Specifications

It is needed to standardize the formats of specifications so that the same specified problems can be processed by different geometry software systems via specific interfaces.

Standardizing Problem Specifications

It is needed to standardize the formats of specifications so that the same specified problems can be processed by different geometry software systems via specific interfaces.

Related work

• Intergeo project offers a common file format for specifying dynamic diagrams. However, the format only works for constructive style.

Standardizing Problem Specifications

It is needed to standardize the formats of specifications so that the same specified problems can be processed by different geometry software systems via specific interfaces.

Related work

• Intergeo project offers a common file format for specifying dynamic diagrams. However, the format only works for constructive style.

• GeoCode is a generic proof scheme standard providing routine codes that can be interfaced with different CAS or provers for proving and DGS for drawing.

More — Macro Constructions

Many systems provide facilities of macro expansions enabling users to customize constructions for use.

More — Macro Constructions

Many systems provide facilities of macro expansions enabling users to customize constructions for use.
However, this functionality

- works internally in the systems;

More — Macro Constructions

Many systems provide facilities of macro expansions enabling users to customize constructions for use.

However, this functionality

- works internally in the systems;
- works for constructive style.

Standardizing Macro Constructions

It is needed to standardize macro constructions so that one can specify problems in terms of customized concepts by defining macros.

Standardizing Macro Constructions

It is needed to standardize macro constructions so that one can specify problems in terms of customized concepts by defining macros.

Related work

- GEOTHER provides a standard form for specifying the entries contained in the predicates routines. However, defined predicates are independent with each other.

Standardizing Macro Constructions

It is needed to standardize macro constructions so that one can specify problems in terms of customized concepts by defining macros.

Related work

- GEOTHER provides a standard form for specifying the entries contained in the predicates routines. However, defined predicates are independent with each other.
- GeoCode provides the facility for users to define new functions in terms of exited functions. However, these functions are defined only in the constructive style.

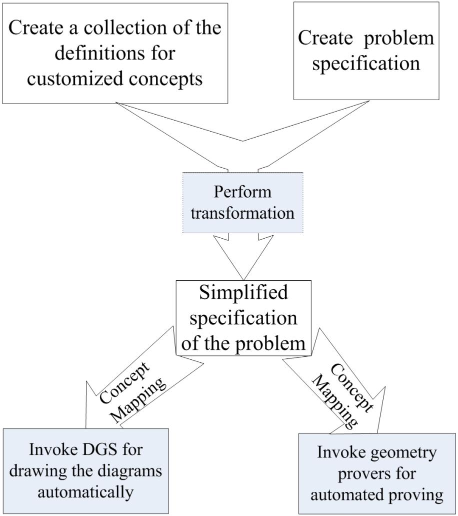
Objectives

- A general geometry programming language is needed in which one can easily and naturally define geometric concepts and specify problems in terms of the customized concepts (for both constructive and constraint type).

Objectives

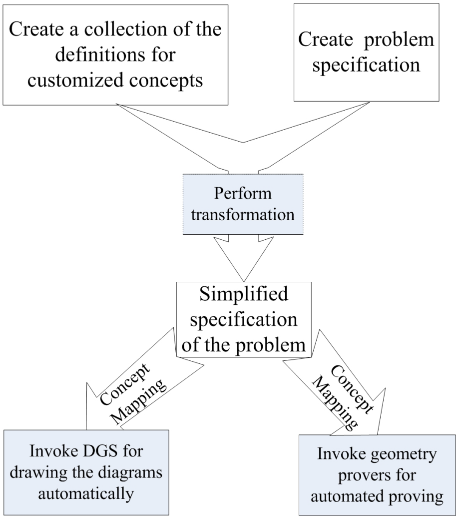
- A general geometry programming language is needed in which one can easily and naturally define geometric concepts and specify problems in terms of the customized concepts (for both constructive and constraint type).
- The facility is needed for transforming the specified problems into the ones that target systems can identify and manipulate via specific interfaces.

Idea



Create a collection of the definitions for customized concepts
Create problem specification
Perform transformation
Simplified specification of the problem
Concept Mapping
Concept Mapping
Invoke DGS for drawing the diagrams automatically
Invoke geometry provers for automated proving

Idea



see Demo

Outline

Concept Symbols

- **Customized concepts**: point, line, intersection, midpoint, area, etc.

Concept Symbols

- **Customized concepts**: point, line, intersection, midpoint, area, etc.
- **Built-in concepts**:
  - **Constants**: 0, π, etc.
  - **Pointers (labels)**: A, B, l, etc.
  - **Types**: Point, Line, Segment, Length, Degree, Number, Boolean, etc.
  - **Algebra concepts**: $times$, $plus$, sin, $squre$, etc.
  - **Set concepts**: $list$, $choose$, $ismember$, etc.
  - **Logic concepts**: and, or, not.

Formalization of Geometric Concepts

- **Abstract concepts**: A ::Point, l ::Line, t ::Triangle, etc.

Formalization of Geometric Concepts

- **Abstract concepts**: A ::Point, l ::Line, t ::Triangle, etc.
- **Entity concepts**:
  - **Geometric objects**: intersection(l::Line,m::Line), perpendicularline(A::Point,l::Line), circumcenter(triangle(A::Point,B::Point,C::Point)), etc.
  - **Geometric quantities**: length(segment(A::Point,B::Point)), ratio(a::GeometricQuantity,b::GeometricQuantity), etc.

Formalization of Geometric Concepts

- **Abstract concepts**: A ::Point, l ::Line, t ::Triangle, etc.
- **Entity concepts**:
  - **Geometric objects**: intersection(l::Line,m::Line), perpendicularline(A::Point,l::Line), circumcenter(triangle(A::Point,B::Point,C::Point)), etc.
  - **Geometric quantities**: length(segment(A::Point,B::Point)), ratio(a::GeometricQuantity,b::GeometricQuantity), etc.
- **Boolean concepts**:
  - **Geometric relations**: parallel(l::Line,m::Line), isin(A::Point,o::Circle), tangent(o::Circle, p::Circle) etc.
  - **Quantity relations**: lt(a::Length,b::Length), equal(c::Degree,d::Degree), etc.

Constructing Geometric Clauses

Clauses are constructed by using instances (of concepts).

- **Reference clauses**: A:=point(), P:=intersection(l,m), etc.

Constructing Geometric Clauses

Clauses are constructed by using instances (of concepts).

- **Reference clauses**: A:=point(), P:=intersection(l,m), etc.
- **Boolean clauses**: perpendicular(l,m), incident(A,l), etc.

Constructing Geometric Clauses

Clauses are constructed by using instances (of concepts).

- **Reference clauses**: A:=point(), P:=intersection(l,m), etc.
- **Boolean clauses**: perpendicular(l,m), incident(A,l), etc.
- **Compound clauses**:
  - **Nesting**: collinear(foot(D,line(A,B)),foot(D,line(A,C)),foot(D,line(B,C)));
  - **Give**: give(triangle(A,B,C));
  - **Configuration**: configuration(E:=intersection(line(A,B),line(C,D)), F:=intersection(line(A,C),line(B,D)));
  - **Declare**: declare(A::Point,B::Point,l::Line);
  - **Logic**: and(parallel(l,m),incident(A,l));
  - **List**: {A;B;C},{point();point();midpoint(segment(A,B))};
  - **Set**: choosediff(A;B;C,2));
  - **Algebra**: times(2,length(segment(A,B))).

Formalization of Geometry Definitions

Format

Definition(Target concept, Return body, Nondegeneracy condition)

Formalization of Geometry Definitions

Format

Definition(Target concept, Return body, Nondegeneracy condition)

For example,

• Definition(intersection(l::Line,m::Line), **[**A::Point **where** and(incident(A,l), incident(A,m))**]**, intersect(l,m))

Formalization of Geometry Definitions

Format

Definition(Target concept, Return body, Nondegeneracy condition)

For example,

- Definition(intersection(l::Line,m::Line), **[**A::Point **where** and(incident(A,l), incident(A,m))**]**, intersect(l,m))
- Definition(completequadrilateral(A::Point,B::Point,C::Point,D::Point, E::Point,F::Point), **[**configuration(E:=intersection(line(A,B),line(C,D)), F:=intersection(line(A,C),line(B,D)))**]**, null)

Formalization of Geometry Definitions

Format

Definition(Target concept, Return body, Nondegeneracy condition)

For example,

- Definition(intersection(l::Line,m::Line), **[**A::Point **where** and(incident(A,l), incident(A,m))**]**, intersect(l,m))
- Definition(completequadrilateral(A::Point,B::Point,C::Point,D::Point, E::Point,F::Point), **[**configuration(E:=intersection(line(A,B),line(C,D)), F:=intersection(line(A,C),line(B,D)))**]**, null)
- Definition(diagonal(completequadrilateral(A::Point,B::Point,C::Point,D::Point, E::Point,F::Point)),{**[**segment(A,D)**]**;**[**segment(B,C)**]**; **[**segment(E,F)**]**}, null)

Formalization of Geometry Problems

Format

Problem(Name, Problem type, Hypothesis, Objective)

Formalization of Geometry Problems

Format

Problem(Name, Problem type, Hypothesis, Objective)

For example,

- Problem(Simson,Theorem,assume(A:=point(),B:=point(),C:=point(), D:=point(),incident(D,circumcircle(triangle(A,B,C)))), show(collinear(foot(D,line(A,B)),foot(D,line(A,C)),foot(D,line(B,C)))))

Formalization of Geometry Problems

Format

Problem(Name, Problem type, Hypothesis, Objective)

For example,

- Problem(Simson,Theorem,assume(A:=point(),B:=point(),C:=point(), D:=point(),incident(D,circumcircle(triangle(A,B,C)))), show(collinear(foot(D,line(A,B)),foot(D,line(A,C)),foot(D,line(B,C)))))
- Problem(Pappus,Theorem,assume(declare(C::Point,F::Point,P::Point, Q::Point,R::Point),A:=point(),B:=point(),D:=point(),E:=point(), give(Pappus(A,B,C,D,E,F,P,Q,R))), show(collinear(P,Q,R)))

Outline

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]
•Def_4: intersection(l::Line,m::Line) $\triangleq$ [A::Point **where** incident(A,l) $\wedge$ incident(A,m)]

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]
•Def_4: intersection(l::Line,m::Line) $\triangleq$ [A::Point **where** incident(A,l) $\wedge$ incident(A,m)]

➢ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_1}$ foot(D,line(E,F))

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]
•Def_4: intersection(l::Line,m::Line) $\triangleq$ [A::Point **where** incident(A,l) $\wedge$ incident(A,m)]

➢ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_1}$ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_2}$

[intersection(perpendicularline(D,line(E,F)),line(E,F))]

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]
•Def_4: intersection(l::Line,m::Line) $\triangleq$ [A::Point **where** incident(A,l) $\wedge$ incident(A,m)]

➢ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_1}$ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_2}$ [intersection(perpendicularline(D,line(E,F)),line(E,F))] $\xrightarrow[substitution]{Def_3,Def_4}$ [var_1::Point **where** incident(D,var_0) $\wedge$ perpendicular(var_0,line(E,F)) $\wedge$ incident(var_1,var_0) $\wedge$ incident(var_1,line(E,F))]

Clause Simplification

Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

•Def_1: line(A::Point,B::Point) $\triangleq$ l::Line
•Def_2: foot(A::Point,l::Line) $\triangleq$ [intersection(perpendicularline(A,l),l)]
•Def_3: perpendicularline(A::Point,l::Line) $\triangleq$ [m::Line **where** incident(A,m) $\wedge$ perpendicular(m,l))]
•Def_4: intersection(l::Line,m::Line) $\triangleq$ [A::Point **where** incident(A,l) $\wedge$ incident(A,m)]

➢ (foot(D,line(E,F)) $\xrightarrow[substitution]{Def_1}$ foot(D,line(E,F)) $\xrightarrow[substitution]{Def_2}$ [intersection(perpendicularline(D,line(E,F)),line(E,F))] $\xrightarrow[substitution]{Def_3,Def_4}$[$var_1$::Point **where** incident(D,var_0) $\wedge$ perpendicular(var_0,line(E,F)) $\wedge$ incident(var_1,var_0) $\wedge$ incident(var_1,line(E,F))]

We adopt eager (inner most) strategy to deal with the nesting cases.

Statement Simplification

Statement Simplification denotes the process of transforming problem specifications by using the related definitions.

Statement Simplification

Statement Simplification denotes the process of transforming problem specifications by using the related definitions.

Example (constraint style)

Problem(Simson,Theorem,assume(A:=point(),B:=point(),C:=point(),
D:=point(),incident(D,circumcircle(triangle(A,B,C)))),
show(collinear(foot(D,line(A,B)),foot(D,line(A,C)),foot(D,line(B,C)))))

$$\xrightarrow[\textit{simplification}]{\textit{definitions}}$$

Problem(Simson,Theorem,assume(declare(var_0::Point,var_1::Point,var_2::Line,
var_3::Point,var_4::Line,var_5::Point,var_6::Line,var_7::Point),
A:=point(),B:=point(),C:=point(),D:=point(),equal(distance(var_0,D),distance
(var_0,var_1)),equal(distance(var_0,var_1),distance(var_0,A)),equal(distance
(var_0,A),distance(var_0,B)),equal(distance(var_0,A),distance(var_0,C),...),
show(incident(var_3,line(var_5,var_7))))

How to select/match definition for simplification?

We use type matching to select the “correct” definitions for simplifying instances.

How to select/match definition for simplification?

We use type matching to select the “correct” definitions for simplifying instances.

Table: Context table for the current statement

label	geobject
A	point()
B	point()
C	point()
D	point()
l	perpendicularline(A,line(C,D))

How to select/match definition for simplification?

We use type matching to select the “correct” definitions for simplifying instances.

Table: Context table for the current statement

label	geobject
A	point()
B	point()
C	point()
D	point()
l	perpendicularline(A,line(C,D))

Type for instance

Type(foot(D,line(A,B))) = foot(point(),line(point(),point()))
Type(intersection(l,line(C,D))) =
intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))

How to select/match definition for simplification?

We use type matching to select the “correct” definitions for simplifying instances.

Table: Context table for the current statement

label	geobject
A	point()
B	point()
C	point()
D	point()
l	perpendicularline(A,line(C,D))

Type for instance

Type(foot(D,line(A,B))) = foot(point(),line(point(),point()))
Type(intersection(l,line(C,D))) =
intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))

Type for concept

Type(foot(A::Point,l::Line)) = foot(Point,Line)
Type(intersection(m::Line,l::Line)) = intersection(Line,Line)

How to select/match definition for simplification?

We use type matching to select the “correct” definitions for simplifying instances.

Table: Context table for the current statement

label	geobject
A	point()
B	point()
C	point()
D	point()
l	perpendicularline(A,line(C,D))

Type for instance

Type(foot(D,line(A,B))) = foot(point(),line(point(),point()))
Type(intersection(l,line(C,D))) =
intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))

Type for concept

Type(foot(A::Point,l::Line)) = foot(Point,Line)
Type(intersection(m::Line,l::Line)) = intersection(Line,Line)

Generally, type for instance is not equal to type for concept. How to match them?

Type Order

Geometry definitions indicate the order of types. We define type upgrade to match types.

Type Order

Geometry definitions indicate the order of types. We define type upgrade to match types.

Example

- point() $\prec$ Point
- line(Point,Point) $\prec$ Line
- perpendicularline(Point,Line) $\prec$ Line

Type Order

Geometry definitions indicate the order of types. We define type upgrade to match types.

Example

- point() $\prec$ Point
- line(Point,Point) $\prec$ Line
- perpendicularline(Point,Line) $\prec$ Line

intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))
$\prec$ intersection(perpendicularline(Point,Line),Line) $\prec$ intersection(Line,Line)

Type Order

Geometry definitions indicate the order of types. We define type upgrade to match types.

Example

- point() $\prec$ Point
- line(Point,Point) $\prec$ Line
- perpendicularline(Point,Line) $\prec$ Line

intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))
$\prec$ intersection(perpendicularline(Point,Line),Line) $\prec$ intersection(Line,Line)

Type Matching Rule

Let I and C be an instance and a concept, if $\mathtt{Type}(I) \preceq \mathtt{Type}(C)$, then the definition of C can be used to simplify instance I.

How to Perform Simplification?

Instances are not alone but associated with extra information. Normal form is needed to normalize the specification of instance during the process of simplification.

How to Perform Simplification?

Instances are not alone but associated with extra information. Normal form is needed to normalize the specification of instance during the process of simplification.

Normal Form

[I **where** constraint **context** configuration **with** nondegeneracyCondition]

How to Perform Simplification?

Instances are not alone but associated with extra information. Normal form is needed to normalize the specification of instance during the process of simplification.

Normal Form

[I **where** constraint **context** configuration **with** nondegeneracyCondition]

The simplified instances will be normalized into this form at each step of simplification process.

Analysis

Geometry Statement Simplification is a series of operations of transforming all the instances involved in the geometric statement until no instances can be simplified further.

Analysis

Geometry Statement Simplification is a series of operations of transforming all the instances involved in the geometric statement until no instances can be simplified further.

Termination

The process terminates only if there is no loop in the type structure determined by the definitions.

Analysis

Geometry Statement Simplification is a series of operations of transforming all the instances involved in the geometric statement until no instances can be simplified further.

Termination

The process terminates only if there is no loop in the type structure determined by the definitions.

Usability

The simplified problem specifications can be interfaced with Geometry software systems.

More Demo

- Reuse definitions and problem specifications.

  ▸ Pappus,completeQuandrilateral,197,198

More Demo

- Reuse definitions and problem specifications.
  - Pappus,completeQuandrilateral,197,198
- Dealing with multiple returns.
  - example90, 180

Outline

Tools and Open Source Packages

- XML based;

Tools and Open Source Packages

- XML based;
- Java;

Tools and Open Source Packages

- XML based;
- Java;
- JDIC package: JDesktop Integration Components;
- XSLT: SAXON;

Tools and Open Source Packages

- XML based;
- Java;
- JDIC package: JDesktop Integration Components;
- XSLT: SAXON;
- GeoGebra;
- GEOTHER.

Outline

Conclusion

We have presented a geometry programming language for specifying geometric concepts, definitions, and problems.
The specifications are

- encoded easily and naturally;

Conclusion

We have presented a geometry programming language for specifying geometric concepts, definitions, and problems.
The specifications are

- encoded easily and naturally;
- used in both constraint and constructive cases;

Conclusion

We have presented a geometry programming language for specifying geometric concepts, definitions, and problems.
The specifications are

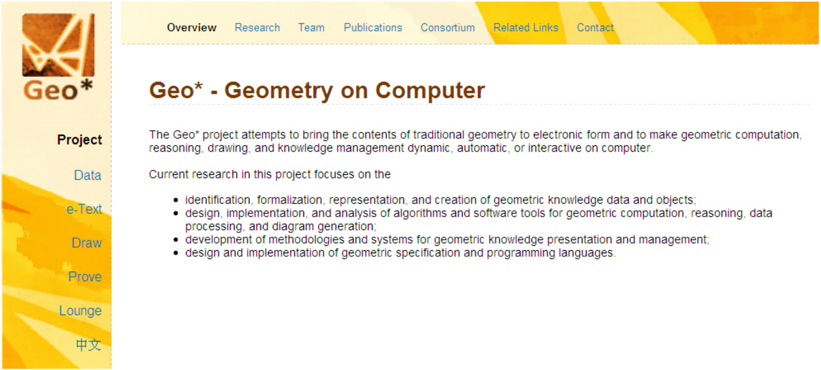
- encoded easily and naturally;
- used in both constraint and constructive cases;
- transformed into ones that can be interfaced with available geometry software systems.

Future Work

The geometry programming language is still at a preliminary stage. The following problems should be considered further.

- prove the correctness of transformation;
- transform the specifications in this language into natural language and the other way round;
- transform the specifications in this language into algebraic counterparts and interface with CAS.

Geo* - Geometry on Computer



Welcome to visit our project home at http://geo.cc4cm.org/

Thanks!