Formal Representation and Automated Transformation of Geometric Statements

Xiaoyu Chen

Beihang University, Beijing, China

July 24, 2010
Outline

1. Motivation
2. Geometry Programming Language
3. Geometric Statement Simplification
4. Implementation
5. Conclusion and Future Work
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

<table>
<thead>
<tr>
<th>Cinderella</th>
<th>GeoGebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perpendicular(a;A)</td>
<td>PerpendicularLine[A,a]</td>
</tr>
<tr>
<td>Circumcircle(A;B;C)</td>
<td>Circle[A,B,C]</td>
</tr>
<tr>
<td>AngleBisector(m;n;A)</td>
<td>AngleBisector[m,n]</td>
</tr>
</tbody>
</table>
Geometry problems (drawing or proving) are specified by applying similar (or same) concepts which are implemented differently in these systems.

<table>
<thead>
<tr>
<th>Constructive style</th>
<th>Constraint style</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cinderella</strong></td>
<td><strong>GEOTHER</strong></td>
</tr>
<tr>
<td>Perpendicular(a;A)</td>
<td>midpoint(A,B,C)</td>
</tr>
<tr>
<td>Circumcircle(A;B;C)</td>
<td>parallel(A,B,C,D)</td>
</tr>
<tr>
<td>AngleBisector(m;n;A)</td>
<td></td>
</tr>
<tr>
<td><strong>GeoGebra</strong></td>
<td><strong>GeoProof</strong></td>
</tr>
<tr>
<td>PerpendicularLine[A,a]</td>
<td>is_midpoint C A B</td>
</tr>
<tr>
<td>Circle[A,B,C]</td>
<td>parallel A B C D</td>
</tr>
<tr>
<td>AngleBisector[m,n]</td>
<td></td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Cinderella</th>
<th>GeoGebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perpendicular(a;A)</td>
<td>PerpendicularLine[A,a]</td>
</tr>
<tr>
<td>Circumcircle(A;B;C)</td>
<td>Circle[A,B,C]</td>
</tr>
<tr>
<td>AngleBisector(m;n;A)</td>
<td>AngleBisector[m,n]</td>
</tr>
</tbody>
</table>

**Table: Constraint style**

<table>
<thead>
<tr>
<th>GEOTHER</th>
<th>GeoProof</th>
</tr>
</thead>
<tbody>
<tr>
<td>midpoint(A,B,C)</td>
<td>is_midpoint C A B</td>
</tr>
<tr>
<td>parallel(A,B,C,D)</td>
<td>parallel A B C D</td>
</tr>
</tbody>
</table>

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.
Motivation

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.
Motivation

More — Macro Constructions

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

Idea

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

X. Chen (BUAA)
ADG '10 (Munich)
July 24, 2010
Motivation

Idea

Create a collection of the definitions for customized concepts

Create problem specification

Perform transformation

Simplified specification of the problem

Invoke DGS for drawing the diagrams automatically

Invoke geometry provers for automated proving

Concept Mapping

Concept Mapping
Outline

1. Motivation
2. Geometry Programming Language
3. Geometric Statement Simplification
4. Implementation
5. Conclusion and Future Work
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, squire, etc.
  - **Set concepts**: list, choose, ismember, etc.
  - **Logic concepts**: and, or, not.
Formalization of Geometric Concepts

- **Abstract concepts**: $A :: \text{Point}$, $l :: \text{Line}$, $t :: \text{Triangle}$, etc.
Formalization of Geometric Concepts

- **Abstract concepts**: $A :: \text{Point}$, $l :: \text{Line}$, $t :: \text{Triangle}$, etc.
- **Entity concepts**:
  - **Geometric objects**: intersection($l :: \text{Line}, m :: \text{Line}$), perpendicular_line($A :: \text{Point}, l :: \text{Line}$), circumcenter($\text{triangle}(A :: \text{Point}, B :: \text{Point}, C :: \text{Point})$), etc.
  - **Geometric quantities**: length($\text{segment}(A :: \text{Point}, B :: \text{Point})$), ratio($a :: \text{GeometricQuantity}, b :: \text{GeometricQuantity}$), 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**: $A := \text{point}()$, $P := \text{intersection}(l, m)$, etc.
Constructing Geometric Clauses

Clauses are constructed by using instances (of concepts).

- **Reference clauses**: $A:=\text{point()}$, $P:=\text{intersection}(l,m)$, etc.
- **Boolean clauses**: $\text{perpendicular}(l,m)$, $\text{incident}(A,l)$, etc.
Constructing Geometric Clauses

Clauses are constructed by using instances (of concepts).

- **Reference clauses**: $A := \text{point}(), \ P := \text{intersection}(l, m)$, etc.
- **Boolean clauses**: perpendicular($l, m$), incident($A, l$), etc.
- **Compound clauses**:
  - **Nesting**: collinear($\text{foot}(D, \text{line}(A, B)), \text{foot}(D, \text{line}(A, C)), \text{foot}(D, \text{line}(B, C))$);
  - **Give**: give(triangle($A, B, C$));
  - **Configuration**: configuration($E := \text{intersection}(\text{line}(A, B), \text{line}(C, D)), \ F := \text{intersection}(\text{line}(A, C), \text{line}(B, D))$);
  - **Declare**: declare($A :: \text{Point}, B :: \text{Point}, l :: \text{Line}$);
  - **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)

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))))
Formalization of Geometry Problems

Format

Problem(Name, Problem type, Hypothesis, Objective)

For example,

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

1. Motivation
2. Geometry Programming Language
3. Geometric Statement Simplification
4. Implementation
5. Conclusion and Future Work
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 denotes the process of transforming the involved instances by applying the corresponding definitions.

**Example (constraint style)**

- $Def_1$: `line(A::Point, B::Point) ≜ l::Line`
- $Def_2$: `foot(A::Point, l::Line) ≜ [intersection(perpendicularline(A, l), l)]`
Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

- $\text{Def}_1$: line($A::\text{Point}, B::\text{Point}$) $\triangleq l::\text{Line}$
- $\text{Def}_2$: foot($A::\text{Point}, l::\text{Line}$) $\triangleq [\text{intersection}(\text{perpendicularline}(A,l),l)]$
- $\text{Def}_3$: perpendicularline($A::\text{Point}, l::\text{Line}$) $\triangleq [m::\text{Line} \text{ where } \text{incident}(A,m) \land \text{perpendicular}(m,l)]$
Clause Simplification denotes the process of transforming the involved instances by applying the corresponding definitions.

Example (constraint style)

- **Def**$_1$: \(\text{line}(A::\text{Point},B::\text{Point}) \triangleq l::\text{Line}\)
- **Def**$_2$: \(\text{foot}(A::\text{Point},l::\text{Line}) \triangleq [\text{intersection}(\text{perpendicularline}(A,l),l)]\)
- **Def**$_3$: \(\text{perpendicularline}(A::\text{Point},l::\text{Line}) \triangleq [m::\text{Line }\text{where }\text{incident}(A,m) \land \text{perpendicular}(m,l)]\)
- **Def**$_4$: \(\text{intersection}(l::\text{Line},m::\text{Line}) \triangleq [A::\text{Point }\text{where }\text{incident}(A,l) \land \text{incident}(A,m)]\)
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($A,m$) $\land$ perpendicular($m,l$)]
- **Def$_4$**: intersection($l$::Line,$m$::Line) $\triangleq$ [$A$::Point where incident($A,l$) $\land$ incident($A,m$)]

$\Rightarrow$ foot($D$,line($E,F$)) $\xrightarrow{Def_1}$ 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$_1$**: \( \text{line}(A::\text{Point},B::\text{Point}) \triangleq l::\text{Line} \)
- **Def$_2$**: \( \text{foot}(A::\text{Point},l::\text{Line}) \triangleq [\text{intersection}(\text{perpendicularline}(A,l),l)] \)
- **Def$_3$**: \( \text{perpendicularline}(A::\text{Point},l::\text{Line}) \triangleq [m::\text{Line} \text{ where } \text{incident}(A,m) \land \text{perpendicular}(m,l)] \)
- **Def$_4$**: \( \text{intersection}(l::\text{Line},m::\text{Line}) \triangleq [A::\text{Point} \text{ where } \text{incident}(A,l) \land \text{incident}(A,m)] \)

\[ \text{foot}(D,\text{line}(E,F)) \xrightarrow{\text{substitution by Def}_1} \text{foot}(D,\text{line}(E,F)) \xrightarrow{\text{substitution by Def}_2} \]

\[ [\text{intersection}(\text{perpendicularline}(D,\text{line}(E,F)),\text{line}(E,F))] \]
Clause Simplification

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

Example (constraint style)

- **Def**₁: \( \text{line}(A::\text{Point},B::\text{Point}) \triangleq l::\text{Line} \)
- **Def**₂: \( \text{foot}(A::\text{Point},l::\text{Line}) \triangleq [\text{intersection}(\text{perpendicularline}(A,l),l)] \)
- **Def**₃: \( \text{perpendicularline}(A::\text{Point},l::\text{Line}) \triangleq [m::\text{Line where} \ \text{incident}(A,m) \land \text{perpendicular}(m,l)] \)
- **Def**₄: \( \text{intersection}(l::\text{Line},m::\text{Line}) \triangleq [A::\text{Point where} \ \text{incident}(A,l) \land \text{incident}(A,m)] \)

\[ \text{foot}(D,\text{line}(E,F)) \xrightarrow{\text{Def}_1} \text{foot}(D,\text{line}(E,F)) \xrightarrow{\text{Def}_2} \]
\[ [\text{intersection}(\text{perpendicularline}(D,\text{line}(E,F)),\text{line}(E,F))] \xrightarrow{\text{Def}_3,\text{Def}_4} [\text{var}_1::\text{Point where} \]
\[ \text{incident}(D,\text{var}_0) \land \text{perpendicular}(\text{var}_0,\text{line}(E,F)) \land \text{incident}(\text{var}_1,\text{var}_0) \land \text{incident}(\text{var}_1,\text{line}(E,F))] \]
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($A::Point,l::Line$) $\triangleq [m::Line \text{ where } incident(A,m) \land perpendicular(m,l)]$
- $Def_4$: intersection($l::Line,m::Line$) $\triangleq [A::Point \text{ where } incident(A,l) \land incident(A,m)]$

$\Rightarrow (foot(D,line(E,F))) \xrightarrow{Def_1 \text{ substitution}} foot(D,line(E,F)) \xrightarrow{Def_2 \text{ substitution}} [\text{intersection(perpendicularline}(D,line(E,F)),line(E,F))] \xrightarrow{Def_3,Def_4 \text{ substitution}} [\text{var}_1::Point \text{ where } incident(D,\text{var}_0) \land perpendicular(\text{var}_0,line(E,F)) \land incident(\text{var}_1,\text{var}_0) \land incident(\text{var}_1,line(E,F))]$

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

\[
\frac{\text{definitions}}{\text{simplification}}
\]

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.

<table>
<thead>
<tr>
<th>Context table for the current statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>label</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Type for instance**

\[
\text{Type}(\text{foot(}D, \text{line(}A, B)\)) = \text{foot(}\text{point}(), \text{line(}\text{point}(), \text{point}())\)
\]

\[
\text{Type}(\text{intersection(}l, \text{line(}C, D)\)) = \text{intersection(}\text{perpendicular line(}\text{point}(), \text{line(}\text{point}(), \text{point}())\), \text{line(}\text{point}(), \text{point}())\)
\]

**Type for concept**

\[
\text{Type}(\text{foot(}A::\text{Point, l::Line}) = \text{foot(}\text{Point, Line})
\]

\[
\text{Type}(\text{intersection(}m::\text{Line, l::Line}) = \text{intersection(}\text{Line, Line})
\]

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

<table>
<thead>
<tr>
<th>label</th>
<th>geobject</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>point()</td>
</tr>
<tr>
<td>B</td>
<td>point()</td>
</tr>
<tr>
<td>C</td>
<td>point()</td>
</tr>
<tr>
<td>D</td>
<td>point()</td>
</tr>
<tr>
<td>l</td>
<td>perpendicularline(A,line(C,D))</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>label</th>
<th>geobject</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>point()</td>
</tr>
<tr>
<td>B</td>
<td>point()</td>
</tr>
<tr>
<td>C</td>
<td>point()</td>
</tr>
<tr>
<td>D</td>
<td>point()</td>
</tr>
<tr>
<td>l</td>
<td>perpendicularline(A,line(C,D))</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>label</th>
<th>geobject</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>point()</td>
</tr>
<tr>
<td>B</td>
<td>point()</td>
</tr>
<tr>
<td>C</td>
<td>point()</td>
</tr>
<tr>
<td>D</td>
<td>point()</td>
</tr>
<tr>
<td>l</td>
<td>perpendicularline(A,line(C,D))</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>label</th>
<th>geobject</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>point()</td>
</tr>
<tr>
<td>B</td>
<td>point()</td>
</tr>
<tr>
<td>C</td>
<td>point()</td>
</tr>
<tr>
<td>D</td>
<td>point()</td>
</tr>
<tr>
<td>l</td>
<td>perpendicularline(A, line(C, D))</td>
</tr>
</tbody>
</table>

**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() ≺ Point
- line(Point,Point) ≺ Line
- perpendicularline(Point,Line) ≺ Line
Type Order

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

Example

- point() ≺ Point
- line(Point,Point) ≺ Line
- perpendicularline(Point,Line) ≺ Line

intersection(perpendicularline(point(),line(point(),point())),line(point(),point())) ≺ intersection(perpendicularline(Point,Line),Line) ≺ intersection(Line,Line)
Type Order

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

Example
• \texttt{point()} \prec \texttt{Point}
• \texttt{line(Point,Point)} \prec \texttt{Line}
• \texttt{perpendicularline(Point,Line)} \prec \texttt{Line}

\texttt{intersection(perpendicularline(point(),line(point(),point())),line(point(),point()))} \prec \texttt{intersection(perpendicularline(Point,Line),Line)} \prec \texttt{intersection(Line,Line)}

Type Matching Rule
Let $I$ and $C$ be an instance and a concept, if $\text{Type}(I) \leq \text{Type}(C)$, 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**

\[ I \ where \ constraint \ \textbf{context} \ \text{configuration} \ \textbf{with} \ \text{nondegeneracyCondition} \]
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.
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.
Reuse definitions and problem specifications.

- Pappus, completeQuadrilateral, 197, 198
More Demo

- **Reuse** definitions and problem specifications.  
  - Pappus, completeQuadrilateral, 197, 198

- Dealing with multiple returns.  
  - example90, 180
Outline

1. Motivation
2. Geometry Programming Language
3. Geometric Statement Simplification
4. Implementation
5. Conclusion and Future Work
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

1. Motivation
2. Geometry Programming Language
3. Geometric Statement Simplification
4. Implementation
5. Conclusion and Future Work
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.
The Geo* project attempts to bring the contents of traditional geometry to electronic form and to make geometric computation, reasoning, drawing, and knowledge management dynamic, automatic, or interactive on computer.

Current research in this project focuses on the:

- Identification, formalization, representation, and creation of geometric knowledge data and objects;
- Design, implementation, and analysis of algorithms and software tools for geometric computation, reasoning, data processing, and diagram generation;
- Development of methodologies and systems for geometric knowledge presentation and management;
- Design and implementation of geometric specification and programming languages.

Welcome to visit our project home at [http://geo.cc4cm.org/](http://geo.cc4cm.org/)
Thanks!