An Axiomatic Approach for Solving Geometric Problems Symbolically ‡

Beat Bruderlin
Computer Science Department
University of Utah
Salt Lake City, UT 84112,
bruderlin@cs.utah.edu
Technical Report Number: UUCS-90-023

November, 1990

‡This work was supported in part by NSF grant # DDM-8910229

An Axiomatic Approach for Solving Geometric Problems Symbolically

Beat Brüderlin
Computer Science Department
University of Utah

Abstract

This paper describes a new approach for solving geometric constraint problems and problems in geometry theorem proving. We developed a rewrite-rule mechanism operating on geometric predicates. Termination and completeness of the problem solving algorithm can be obtained through well foundedness and confluence of the set of rewrite rules. To guarantee these properties we adapted the Knuth-Bendix completion algorithm to the specific requirements of the geometric problem. A symbolic, geometric solution has the advantage over the usual algebraic approach that it speaks the language of geometry. Therefore, it has the potential to be used in many practical applications in interactive Computer Aided Design.

1. Introduction

The theory of Euclidean Geometry is the foundation of almost all Computer-Geometry applications. Also it is one of the first mathematical theories that has been axiomatized systematically by D. Hilbert, in the beginning of this century [HIL 71]. Already, there have been some very early approaches for proving geometric theorems with Computer-Algebra methods [COL 75]. The development of new, more efficient algebraic methods (see, for instance, [BUL 82]), led to a renaissance of automated geometry theorem proving (see [WUW 86], [KAM 88]). In this paper a new approach is presented. We use term rewriting and logic programming and directly apply geometric axioms for deducing constructive proofs of elementary geometry theorems, as well as for solving geometric constraint problems. We describe the basic conceptions of the algorithm and its application for interactive modelling of geometric objects.

2. Symbolic geometric constraint solving.

Computer Aided Design- (CAD) systems are used for interactively constructing geometric objects. We assume that these objects should fulfill certain given specifications and that the interactive user can chose from a great number of construction tools offered by the system. Let us further assume that the specification can be expressed by geometric constraints, such as, for instance, distances, angles, etc. Therefore, the problem and its solution are of purely geometric nature. If we express the properties of each construction operation in an axiomatic way, the problem of finding the sequence of construction operations for a specified object may be seen as the problem of finding a proof of the constructibility of the object with the given tools. Therefore, a construction problem is a special case of a geometric theorem proving problem.

Most known approaches to this or similar problems first translate from a geometric language to an algebraic formulation, by introducing coordinates. A solution can be found with numerical or algebraic methods. With the approach presented here the geometric constraints (which are geometric relations between points) are represented by predicates. The geometric construction operations are represented by functional expressions. We can express geometric axioms by equations on first order formulas, which allows us to use a rewrite-rule mechanism, and by implications, which may be inferred by a backtracking mechanism, such as implemented in the logic-programming language Prolog. We can prove uniform termination of the inference mechanism using the rewrite-rules, and with a special formulation of the Knuth-Bendix critical pair criterion we may prove completeness (or unique termination).

The algorithm may be applied for supporting the interactive definition of geometric objects by geometric constraints: 1) It may derive a sequence of construction operations automatically from the constraints. 2) It may be used for detecting contradictions between the constraints or redundancies, and for explicitly showing internal degrees of freedom of not yet sufficiently specified objects. Thus, it gives the interactive user a hint, where to add further constraints for uniquely specifying the object. 3) Geometric relations that are implied by the specification by constraints may be inferred. Thus, a certain class of theorems of elementary geometry may be proved automatically by the algorithm.

Since the approach does not require a translation to algebraic expressions over

coordinates it is easy to explain the intermediate or final results in a geometric language. Therefore, the approach is better suited for an integration with an interactive CAD system than algebraic approaches.

In order to describe the symbolic approach to geometric constraint solving we first need to define a language in which the problems may be specified formally. Then, we express the constraint solving mechanism in this language.

2.1 Expressing constraints by predicates.

Constraints are relations between points which may be expressed by predicates on these points. The predicates may be stored in the database of a Prolog interpreter as so called 'facts'. Here is a list of the geometric predicates on points, as used in our approach:

```
p(P, Pos).
d(P1, P2, D).
s(P1, P2, Alpha).
v(P1, P2, [Alpha, D]).
a(P1, P2, P3, Beta).
tr(P1, P2, P3, [Alpha, Beta]).
```

The semantics of the predicates is defined as follows:

To express that the position of a point *P* is known we use the predicate *p(P, Pos)*. This means that the position of *P* is defined by the value of *Pos* to which a symbolic expression may be assigned. The expressions must be ground terms, i.e. no variables may occur in them. Possible expressions are, e.g. a pair of coordinates *[10, 0.5]*, or a geometric function, such as *intersection(line(p(P2), 90), circle(p(P3), 20))*, meaning that the point is located at the intersection of a line through another point *P2* with a slope of 90 degrees toward the x-axis, and a circle with center point *P3* and a radius of 20.

The predicate *d(P1, P2 Dist)*, expresses that we know the distance between two

points. For the value of *Dist* we may again write a constant number, a parameter, or a function such as *length(vector(P3, P4))*, i.e. the distance of two points with already specified position.

s(P2, P1, Alpha) constrains the slope of the line connecting *P1* and *P2* by the angle *Alpha* to the x-axis (measured counterclockwise).

Fig. 1 shows how an angle Beta is associated with three points by the predicate *a(P1, P2, P3, Beta)*.

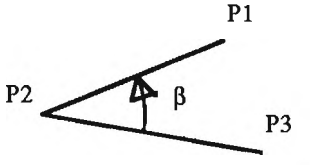


fig.1

The angle *Beta* is measured counterclockwise from the *line (P2, P3)* toward the *line (P2, P1)*. An angle going in the other direction would be written with negative sign: *a(P1, P2, P3, Beta)* ⇔ *a(P3, P2, P1, -Beta)*. If we know the value of two angles of a triangle we express this by the predicate *tr(P1, P2, P3, [Alpha, Beta])*. The predicate *tr* stands for triangle, and implies that the value of the third angle is known implicitly: *Gamma = 180° - (Alpha + Beta)*. Note that only angles, but not distances between points are defined by 'tr'.

Congruence relations. In addition to the above predicates which assign ground terms to angles distances, positions, etc., the following equations are used for expressing congruence relations:

```
d(P1, P2) = d(P3, P4).
```
The distance of points *P1* and *P2* is congruent to the distance between points *P3* and *P4*.

```
a(P1,P2,P3) = a(P4,P5,P6).
```
The two angles are congruent.

`s(P1, P2) = s(P3, P4).`
The *line(P1,P2)* is parallel to *line(P3,P4).*

`v(P1, P2) = v(P3, P4).`
The vector*(P1,P2)* is equal to *vector(P3,P4).*

`tr(P1,P2,P3) = tr(P4,P5,P6).`
The two triangles are similar.

`tr(P1,P2,P3) = str(P4,P5,P6).`
The two triangles are similar up to symmetry.

`p(P1) = p(P2).`
Two points are identical

What can be expressed by the predicates?

The constraints discussed here are restricted to relations between points. Each pair of points implicitly defines a line which may be specified by constraining these two points. To specify the properties of a circle we would have to constrain some characteristic points (e.g. center point, and a point on the periphery).

With *s(P1, P2) = s(P2, P3)* we may express that three points are collinear, and that *P2* is *between P1* and *P3* which is equivalent to Tarski's predicate *b(P1, P2, P3)* [TAR 51].

Two triangles that are joined by two common points build a quadrilateral. By expressing the similarity of joined triangles we may describe the similarity of polygons, or the similarity of mirrored polygons (predicate *str*).

2.2 Rules for compass and ruler construction

To derive the position of points that are not explicitly specified by a corresponding predicate we may apply rules known from constructing with compass and ruler. An example: given the positions of two points *P1* and *P2*, the distance between *P1* and a third point *P3* and the distance between *P2* and *P3*, we may construct *P3* by inter-

secting two circles. We first write the precondition of the rule by a conjunction of predicates:

p(P1, [Pos1]) ∧ p(P2, [Pos2]) ∧ d(P1, P3, R1) ∧ d(P2, P3, R2)

The position of the third point *P3* is found by intersecting two circles with centers *P1* and *P2*. This is expressed symbolically by:

p(P3, intersection(circle(p(P1), R1), circle(p(P2), R2))).

We formalize this construction rule by introducing the following notation for rewrite-rules: Conjunctions of predicates are expressed as lists of predicates '[ ]', and the symbol '->' indicates the direction in which the rewrite-rule is applied. The above rule is represented by the following rewrite-rule:

```
[p(P1, [Pos1]), p(P2, [Pos2]), d(P1, P3, R1), d(P2, P3, R2)]
-> [p(P1, [Pos1]), p(P2, [Pos2]),
    p(P3, intersection(circle(p(P1), R1), circle(p(P2), R2)))].
```

We find similar rules for intersecting circles and lines, or two lines.

```
[p(P1, [Pos1]), p(P2, [Pos2]), d(P1, P3, R), s(P2, P3, Sl)]
-> [p(P1, [Pos1]), p(P2, [Pos2]),
    p(P3, intersection(circle(p(P1), R), line(p(P2), Sl)))].
```

```
[p(P1, [Pos1]), p(P2, [Pos2]), s(P1, P3, Sl1), s(P2, P3, Sl2)]
-> [p(P1, [Pos1]), p(P2, [Pos2]),
    p(P3, intersection(line(p(P1), Sl1), line(p(P2), Sl2)))].
```

All rules are of the form "φ -> Ψ". The *precondition* φ and the *postcondition* Ψ are in Skolem form. I.e., they are conjunctions of literals (predicates on points) with Skolem functions for the point positions, distances, etc. These functions express the geometric operations.

How are the rewrite-rules applied? All constraints defined by the user's specification are expressed by predicates which are stored as 'facts' in the Prolog database. An inference mechanism tries to apply all the rules. A rule can only be applied if the precondition holds. If the corresponding predicates of the precondition can be matched with facts in the database, the rule fires, and a *transaction* on the constraint database is performed. Those facts on the left side of the rule, not occurring on the right side are replaced in the database by the ones only occurring on the right side of the rule. Inserting and retracting facts in the database is realized with the commands *retract* and *assert*, which are built-in predicates in Prolog. A transaction must be carried out as an atomic operation, i.e. either it is carried out completely, or not at all, and thus preserves consistency of the database. The algorithm terminates when no rule applies.

The inference mechanism, together with rules like the one described above determine an algorithm for automatically constructing a geometric object from a geometric specification. The result can be regarded as a proof for the correctness of the specification by constraints. Every step of the proof corresponds to a geometric operation. If we write down the sequence of operations during the proof, we get an imperative definition of the specified object. The specification of a geometric object is complete and consistent, when for each point P_i, there exists exactly one predicate *p(Pi, POSi)*.

The following 2-D example shows the effect of the construction algorithm. The initial content of the database is (see fig. 2):

p(P1, [100, 100]).
p(P2, [200, 100]).
d(P1, P3, 75).
d(P2, P3, 80).
d(P3, P4), 60).

s(P2, P4, 90).

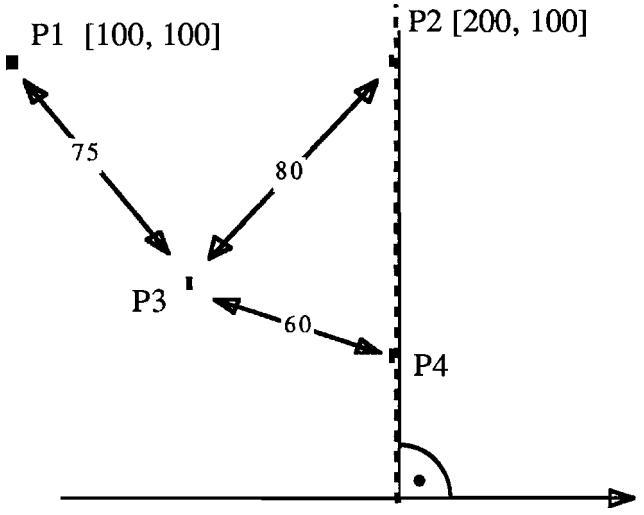


fig. 2 Four constrained points.

After running the construction algorithm, the database contains:

p(P1, [100,100]).
p(P2, [200,100]).
p(P3, intersection(circle(p(P1), 75), circle(p(P2), 80))).
p(P4, intersection(circle(p(P3), 60), line(p(P2), -90))).

The result of the algorithm is a symbolic prescription for constructing the points with compass and ruler. The new facts are asserted in a sequence determined by the algorithm. The expressions specifying point positions therefore may refer to points defined earlier in the construction process (in our example, the symbolic expression for the position of point *P4* refers to points *P2* and *P3*). A solution would have been found even if we expressed the distances and angles by parameters, instead of numbers. In our example the construction parameters are already instantiated by numeric values. Therefore, the geometric expressions may also be evaluated numerically. For displaying the result on a graphical screen we have to provide some procedures for computing the geometric operations on coordinates. In our software system the

numerical evaluation is carried out by procedures written in the procedural programming language Modula-2. These procedures are compiled and linked to the Prolog interpreter as so called 'built-in' predicates. The technique of linking the two languages is discussed in [BRU 85]. When evaluating the symbolic expressions the Prolog program goes sequentially through the facts and finds numeric values for the symbolic expressions by means of the built-in predicates. The computed shape may then be displayed on a graphical screen by some built-in graphic predicates.

Geometric operations do not necessarily have only one unique solution. E.g. intersecting two circles may result in zero, one, or two points. We therefore speak of a non-deterministic operation. Fig. 3 shows the various intersections of circles and lines that correspond to the symbolic solution above.

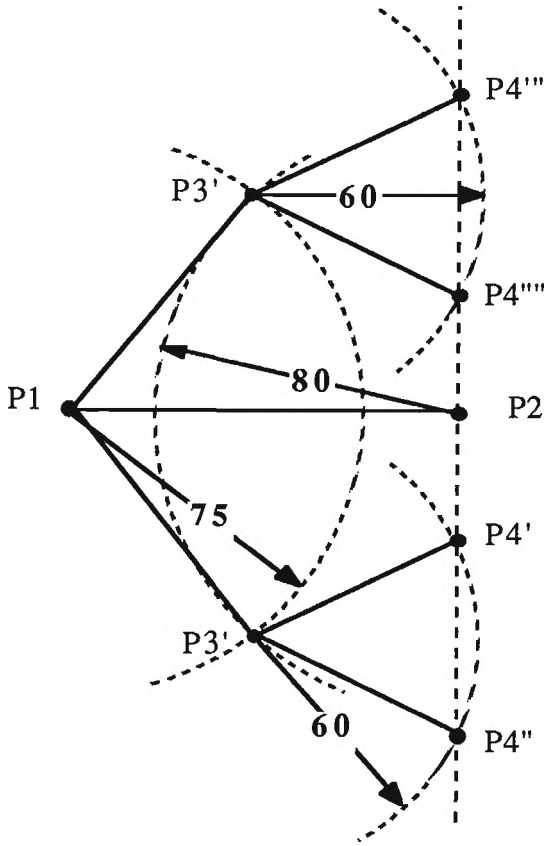


fig. 3 carrying out the geometric constructions

In backtracking, the built-in predicates return all possible values for each operation. The backtracking mechanism of Prolog is used for an exhaustive search of all alternatives. Fig. 4 shows the four possible shapes of a quadrilateral consisting of points P1 to P4.

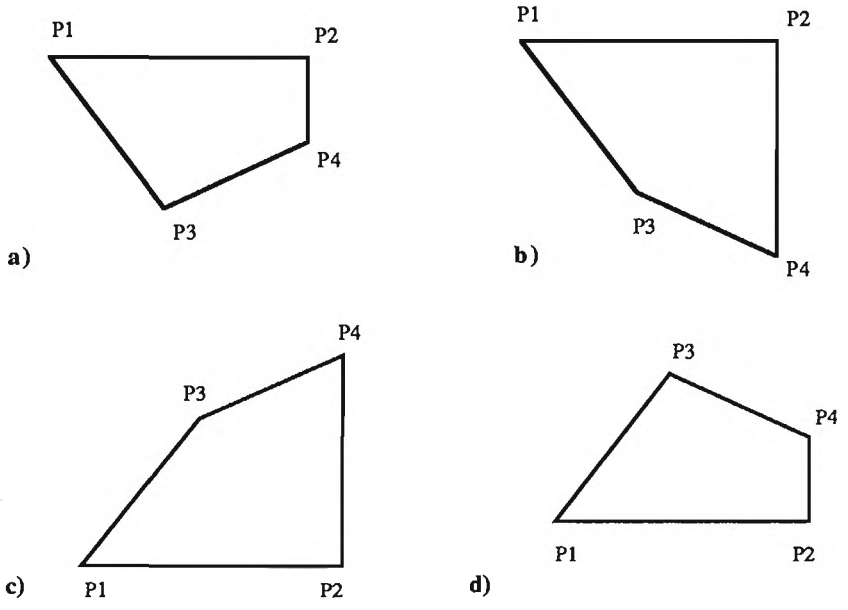


fig. 4 The numerical evaluation of the symbolic result (4 possible solutions).

2.3 Expressing axioms of Euclidean geometry as rewrite-rules

A geometric constraint solver should not only be capable of applying construction operations; it should reason about congruence relations as well. We realized this by incorporating axioms of Euclidean geometry in form of geometric rewrite-rules into the algorithm. An example:

For every pair of triangles, if two sides and the angle between them are congruent, then also all other angles are congruent (fig. 5):

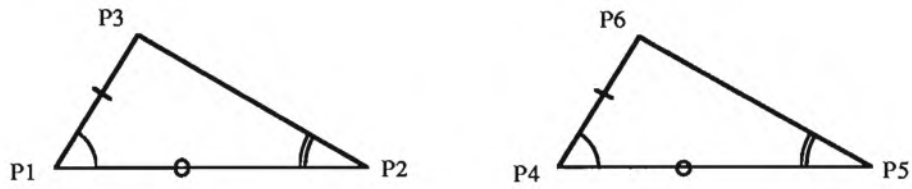


fig. 5 Congruent triangles

```
d(P1,P2) = d(P4,P5) ∧ d(P1,P3) = d(P4,P6)
   ∧ a(P3,P1,P2) = a(P6,P4,P5)
   ⊃ a(P1,P2,P3) = a(P4,P5,P6)
```

We wish to express this axiom as a rewrite-rule which enables us to employ the same inference mechanism as for the construction rules. When carrying out the inference the predicates on the left side of the rule are replaced by those on the right side; therefore it is important that no information is lost by applying the rewrite-rule. First we express the above axiom as an equation on clauses.

[d(P1,P2) = d(P4,P5), d(P1,P3) = d(P4,P6), a(P3,P1,P2) = a(P6,P4,P5)] =
[a(P1,P2,P3) = a(P4,P5,P6), a(P3,P1,P2) = a(P6,P4,P5), d(P1,P2) = d(P4,P5)]

By imposing a direction of application the equation may be expressed as a rewrite-rule with the desired property.

[d(P1,P2) = d(P4,P5), d(P1,P3) = d(P4,P6), a(P3,P1,P2) = a(P6,P4,P5)]
   ->
[a(P1,P2,P3) = a(P4,P5,P6), a(P3,P1,P2) = a(P6,P4,P5), d(P1,P2) = d(P4,P5)]

3. Theoretical properties of the constraint solver (completeness and termination).

A theoretic framework has been developed for proving the algorithmic completeness and termination of the mechanism that applies the rewrite-rules. We

adapted the 'critical-pair algorithm' by Knuth and Bendix [KNB 70] for the structure of the geometric problem. The algorithm introduces new rewrite-rules as lemmas of the theory. Using this method, we can prove that the set of rules is complete, and that the inference mechanism applying these rules always terminates. Thus, we can show that the algorithm is a decision algorithm for theorems provable by the axioms, and that it solves the constraint problems solvable by the construction rules [BRU 87].

In this chapter we discuss some properties of the rewrite-rules introduced in the previous chapter. Important question are: first, whether the algorithm applying the rules stops in any case (the *uniform termination* property of the algorithm), and second, whether the sequence in which the rules are applied matters for the result (the *unique termination* property). We give a short introduction to theory of term rewriting methods and the Knuth-Bendix critical pair algorithm. We have implemented a Knuth-Bendix algorithm in PROLOG accepting the rewrite rules of the previous chapter. The algorithm uses unification modulo associativity, commutativity, and a set of equations. The Knuth-Bendix algorithm is applied as a meta algorithm for finding new rules, and to make the set of rules algorithmically complete.

3.1 Termination properties of the algorithm

Uniform termination. The idea behind the geometric rewrite-rules was to replace some facts in the database by "simpler" ones. To prove that the inference mechanism applying the rules to a given database terminates, we have to find orderings such that the right hand side of a rule is always smaller than the left hand side, and furthermore to prove that the orderings have lower bounds so that no infinite decreasing sequence is possible.

Definition 3.1: (*well-foundedness*) Given a set of terms T, a partial ordering '$\succ$' on T is *well-founded* (or Noetherian) if there is no infinite descending chain of terms $t_1 \succ t_2 \succ \ldots$

Definition 3.2: (*order-function* τ) For our geometric predicates and for the set of conjunctions C we introduce a function $\tau: C \rightarrow \mathbb{N}$ which is defined as follows:

for *literals*:
$\tau(d(P1, P2, \_)) = 6$
$\tau(a(P1, P2, P3, \_)) = 5$
$\tau(s(P1, P2, \_)) = 4$
$\tau(tr(P1, P2, P3, \_)) = 3$
$\tau(v(P1, P2, \_)) = 2$
$\tau(p(P1, \_)) = 1$
$\tau(false) = 0$

$\tau(d(P1, P2) = d(P3, P4)) = 6$
$\tau(a(P1, P2, P3) = a(P4, P5, P6)) = 5$
$\tau(s(P1, P2) = s(P3, P4)) = 4$
$\tau(tr(P1, P2, P3) = tr(P4, P5, P6)) = 3$
$\tau(tr(P1, P2, P3) = str(P4, P5, P6)) = 3$
$\tau(v(P1, P2) = v(P3, P4)) = 2$
$\tau(p(P1) = p(P2)) = 1$

For conjunctions X, Y: $\tau(X \wedge Y) = \tau(X) + \tau(Y)$, for the empty conjunction: $\tau() = 0$

Lemma 3.1: If for all rules $l \rightarrow r$: $\tau(l) > \tau(r)$ then τ on C is well founded.

Proof of lemma 3.1: For each conjunction X to which a rewrite-rule can be applied τ has some finite value $\tau(X) = n \in \mathbb{N}$. Each rule applied to X reduces $\tau(X)$ at least by one (by assumption), 0 is a lower bound for τ, therefore X is reduced in at most n steps to an irreducible conjunction X_0. ♦

Unique termination. With the following example we want to show what happens when different rules may be applied to the same facts in a database (see fig. 3.1). For the points *P1* to *P6* some relations for distance-congruence, angle-congruence, and parallelism are expressed by the facts in the database. When examining the rewrite-rules described in chapter 3 we can see that rules r_1, and r_2 can both be applied to the initial database. Depending on which of the two rules is applied the database will be in a different state (fig. 3.2, or fig. 3.3). Proceeding from there we may continue applying rules. On the left side, after applying rule d_1 twice, we end with a database state as shown in fig. 3.4. On the right side we may apply rule r_2 twice and then d_1,

and end with the database shown in fig. 3.6.

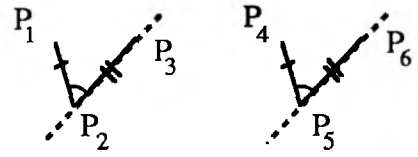


fig. 3.1

```
d(P1, P2) = d(P4, P5),   s(P2, P3) = s(P5, P6),
d(P2, P3) = d(P5, P6),   a(P1, P2, P3) = a(P4, P5, P6).
```

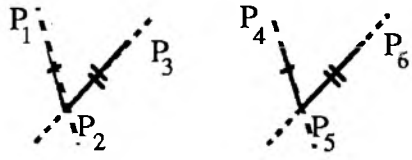


fig. 3.2

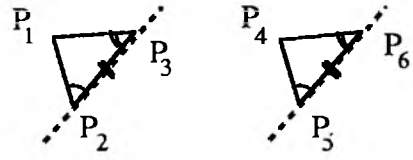


fig. 3.3

```
d(P2, P3) = d(P5, P6),
d(P1, P2) = d(P4, P5),
s(P1, P2) = s(P4, P5),
s(P2, P3) = s(P5, P6).
```

2 x d_1

```
s(P2, P3) = s(P5, P6),
a(P1, P2, P3) = a(P4, P5, P6),
a(P2, P3, P1) = a(P2, P3, P1),
d(P2, P3) = d(P5, P6).
```

2 x r_2

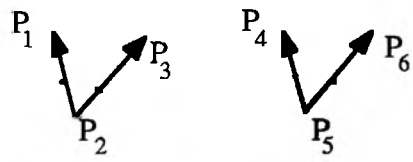


fig. 3.4

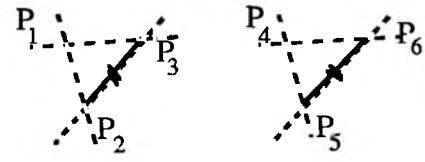


fig. 3.5

```
v(P2, P3) = v(P5, P6),
v(P1, P2) = v(P4, P5).
```

```
d(P2, P3) = d(P5, P6),
s(P2, P3) = s(P5, P6),
s(P1, P2) = s(P4, P5),
s(P1, P3) = s(P4, P6).
```

d_1

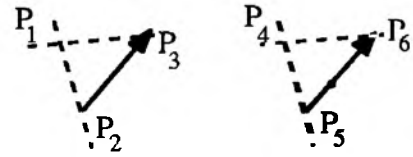


fig. 3.6

```
v(P2, P3) = v(P5, P6),
s(P1, P2) = s(P4, P5),
s(P1, P3) = s(P4, P6).
```

In both cases we can't find rules that apply. The facts in the databases are what we call irreducible. Since both states do have the same offspring and the inference mechanism maintains the consistency of the database the two final states are equivalent but obviously not identical. The sensitivity to the sequence of application is an undesired property of our rules. The actual situation may be cured by adding a new rewrite-rule that either rewrites the facts in fig. 3.4 to those of fig. 3.6 or the other way round. With the order function τ defined above we may find out which way round. The total value for the facts of the database in fig. 3.4 is $\tau = 4$, and the value of the facts in fig. 3.6 is $\tau = 10$, to maintain well-foundedness the new reduction rule must be:

```
[v(P2, P3) = v(P5, P6), s(P1, P2) = s(P4, P5), s(P1, P3) = s(P4, P6)]

->  [v(P2, P3) = v(P5, P6), v(P1, P2) = v(P4, P5)]
```

In order to find out whether we need more rules it seems that we need to inspect an infinite number of such examples. We may also ask ourselves: "Does there exist a finite set of reduction rules that suffice for obtaining the desired property, namely to bring any database to normal form, independently of the order in which the rules are applied, or do we have to add more and more rules?"
The answers to these question can be given by a theory applied for solving word problems in algebra, described by Knuth and Bendix in 1965 [KNB 70]. This theory is discussed formally in the next section.

3.2 Using the Knuth-Bendix completion algorithm to obtain a canonical set of rewrite rules.

In this section the theoretical foundation of a proof of the unique termination- (or Church-Rosser-) property of an algorithm that is based on reduction rules is developed. An algorithm invented by Knuth and Bendix for completing a set of reduction rules that does not have the Church-Rosser property is described. We first need to give some definitions.

Definition 3.3: *(term, set of term T, sub-terms, occurrence, @, Oc, replacement, $t[u \leftarrow s]$)*

1) A variable is a term: $x_1, x_2, \ldots \in T$
2) A constant is a term: $c_1, c_2, \ldots \in T$
3) If $t_1, t_2, \ldots, t_m \in T$ and f_i^m is a function symbol for an m-ary function or operator then $f_i^m(t_1, t_2, \ldots, x_m) \in T$.

We say that the term $t = t(t_1, t_2, t_3(t_{3,1}, t_{3,2}))$ consist of terms $t, t_1, t_2, t_3, t_{3,1}, t_{3,2}$ (the *sub-terms* of t). The structure of t may be represented as a tree (fig. 3.7)

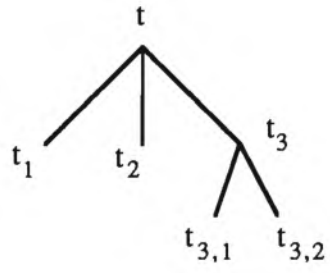


fig. 3.7 The tree structure of a term

With $t_{3,1} = t@(3,1)$ we express that $t_{3,1}$ occurs at the place $(3,1)$ in t. We call $(3,1)$ the *occurrence* of $t_{3,1}$ in t. The occurrence of t itself in t is (0). With $Oc(t)$ we denote the set of the occurrences of all sub-terms in t. For the above example $Oc(t) = \{(0), (1), (2), (3), (3,1), (3,2)\}$. For terms s, t and an occurrence $u \in Oc(t)$, $w = t[u \leftarrow s]$ is the term derived from t by replacing the sub-term occurring at u in t by s.

Definition 3.4: *(substitution, instance, unifier, most general common instance, mgu)*
A *substitution* σ is a finite set of replacements of variables x_i by terms t_i : $\sigma = \{x_1/t_1, x_2/t_2, \ldots, x_k/t_k\}$. Applying a substitution σ to a term t which is expressed by σt means to replace simultaneously all occurrences of x_i in t by t_i for all replacements $x_i/t_i \in \sigma$. The term $s = \sigma t$ is called an *instance* of t. A term t is called *more general* than a term s if s is an instance of t, i.e. if $\exists\ \sigma$: $\sigma t = s$ but t is not an instance of s. Two terms t_1 and t_2 can be unified if there is a substitution σ which applied to the two terms makes them identical $(\sigma t_1 = \sigma t_2)$. The term $w = \sigma t_1 = \sigma t_2$ is called a *common instance* of t_1 and t_2 and σ is called their *unifier*. A unifier does not necessarily exist for any two terms, and in general there is more than only one possible common instance for two terms. If the common instance w is more general than all other common instances of t_1 and t_2, w is called the *most general common instance*, and the corresponding substitution σ is called the *most general unifier (mgu)* of t_1 and t_2. It can be shown that an *mgu* is unique up to renaming of variables. An early description of an algorithm for finding the *mgu* of two terms may be found in [ROB 65]. More efficient unification algorithms have been

developed since then which are integrated in the interpreters of the programming language PROLOG.

Definition 3.5: *(Equational theory, '$=_E$').* An equational theory is a theory represented by a set of equations E on terms: $E = \{t_i' = t_i''\}$. An important question that may be asked in such a theory is: Given two terms s and t, are s and t congruent modulo the set of equations E? We say that $s =_E t$ if we find a finite chain of equations $t_{i_1}' = t_{i_1}''$, $t_{i_2}' = t_{i_2}''$, ... $t_{i_n}' = t_{i_n}''$ where $t_{i_k}' = t_{i_k}''$ are instances of equations of E: with either $(t_{i_k}' = t_{i_k}'') = (\sigma_i t_i' = \sigma_i t_i'')$ or $(t_{i_k}' = t_{i_k}'') = (\sigma_i t_i'' = \sigma_i t_i')$, $s = t_{i_1}''$, $t = t_{i_n}'$, and $t_{i_k}'' = t_{i_{k+1}}'$ for $k = 1$ *to* $n-1$. The congruence relation '$=_E$' is defined as the *reflexive-symmetric-transitive closure* of the equations in E. The question whether such a chain of equations exists for two terms and a given set of equations E is called a *word problem* and is undecidable in general.

Definition 3.6: *(reduction rules, order function τ, $\rightarrow$).* For the equations $a = b \in E$ one can often define in a natural way a "*reduction operation*" '$\rightarrow$' or "*reduction rule*" $a \rightarrow b$. With R we denote the set of all reduction rules derived from E by orientating the equations, such that the left side of the rule is always bigger than the right side, with respect to an order function τ (see for example the previous section). To explain how a reduction $a \rightarrow b \in R$ is applied to a term s we give the following definition.

Definition 3.7: *(reduction, completeness, uniform termination, unique termination).*
$s \rightarrow_R t$ (s reduces to t by R) if and only if there is a rule $a \rightarrow b \in R$, a substitution σ and an occurrence $u \in Oc(s)$ such that $s@u = \sigma a$, and $t = s[u \leftarrow s(b)]$.

'$\rightarrow_R$' is called complete if it has the following two properties:

- The iteration of reduction process always terminates after a finite number of steps at an irreducible object (*uniform termination property*).

- Different reduction processes starting from the same object always terminate at the same irreducible object (*unique termination property*).

In order to prove these properties for a given set of reduction rules R we need to introduce the following definitions.

Definition 3.8: *($\rightarrow_R^*$, $x\downarrow_R y$, Noetherian, irreducible term).*

1) $x \rightarrow_R^* y$ means that there is a finite sequence of reduction rules $a \rightarrow_R b \in R$ that can be applied consecutively to the term x which is then reduced to y by means of these rules.

2) By $x\downarrow_R y$ we mean that x and y have a common successor, i.e. $\exists\, z: x \rightarrow_R^* x \; {}_R^*\!\leftarrow y$.

3) '$\rightarrow_R$' is called *Noetherian* if there is no infinite chain of the form $x_1 \rightarrow_R x_2 \rightarrow_R x_3 \rightarrow_R \cdots$.

4) x is called an *irreducible term*, if there is no rule in R that further reduces x.

From now on we assume that '$\rightarrow_R$' is Noetherian. Then the following lemmas (3.2 - 3.4) hold:

Lemma 3.2: S_R is a canonical simplifier for '$=_E$' if and only if '$\rightarrow_R$' has the *Church-Rosser* property, i.e. $\forall\, x, y \in T: x =_E y \Rightarrow x\downarrow_R y$. In other words: If we can prove that S_R is a canonical simplifier, then congruence modulo E becomes decidable. For any two terms $x =_E y$ holds, if and only if $S_R(x) = S_R(y)$. (the function $S_R(x)$ is defined algorithmically by the inference mechanism applying the rules of R on x until an irreducible term is obtained).

Lemma 3.3: Reduction of the Church-Rosser property to *confluence.*
'$\rightarrow_R$' has the Church-Rosser property if and only if '$\rightarrow_R$' is confluent (i.e. $\forall\, x, y, z \in T, x \; {}^*\!\leftarrow z \rightarrow^* y \Rightarrow x\downarrow_R y$.

Lemma 3.4: Reduction of confluence to *local confluence* (Newman 1942). '$\rightarrow_R$' is confluent if and only if '$\rightarrow$' is locally confluent, i.e. $\forall\, x, y, z \in T,\ x \leftarrow z \rightarrow y \Rightarrow x{\downarrow_R}y$.

Proof of lemmas 3.2 - 3.4: The proofs of the lemmas with an almost identical notation can be found in [BUC 82]. Therefore they are given without proof here.

The criterion of local confluence seems intuitively simpler than proving canonicality through the original definition but is still not effective since an infinite number of terms x, y, z need be considered in general. With the following theorem 3.5 by Knuth and Bendix we obtain a criterion for proving the Church Rosser property of a set of rules by a finite number of tests.

Definition 3.9: *(critical pair).* The terms p and q form a *critical pair* <p,q> in R if there are rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ in R, and an occurrence u in $Oc(l_1)$ such that $l_1@u$ is not a variable, $l_1@u$ and l_2 are unifiable, $p = \sigma_1(l_1)\ [u \leftarrow \sigma_2(r_2)]$, and $q = \sigma_1 r_1$ where σ_1 and σ_2 are substitutions such that $\sigma_1(l_1@u) = \sigma_2 l_2$ is a *most general common instance* of $l_1@u$ and l_2 with the property that no variable of $\sigma_1(l_1@u)$ occurs in l_2.

Theorem 3.5: Reduction of local confluence to *confluence of critical pairs* (Knuth - Bendix, 1967). '$\rightarrow_R$' is locally confluent if and only if for all critical pairs p, q of R: $p{\downarrow_R}q$.

Proof of theorem 3.5: Again the proof of the theorem can be found in [BUC 82].

It should be clear from the definition that for a finite set of rules only a finite number of critical pairs can exist, and therefore the Church Rosser property may be proved in finite time. The so called "critical pair algorithm" tests for all pairs of rules in R if there is a superposition of the left side of one rule and a sub-term of the left side of another rule. When we know that '$\rightarrow_R$' is not confluent we need some method for transforming the non canonical system into a canonical one without changing the theory. An algorithm called the *Knuth-Bendix critical pair completion algorithm* converts critical

pairs into new rules. The algorithm is outlined roughly as follows:

```
1)  Find a critical pair <p,q> in R where S_R(p) ≠ S_R(q);
    If no such pair was found report "R is confluent"; go to 4
2)  if τ(S_R(p)) < τ(S_R(q)) then
          R := R ∪ (S_R(q) → S_R(p))
    if τ(S_R(p)) > τ(S_R(q)) then
          R := R ∪ (S_R(p) → S_R(q))
    else report "aborted" got to 4
3)  use the new rules to simplify (or delete) the rest of R
    go to 1
4)  stop
```

In the following section we want to apply the theory with some modifications to the geometric reduction rules.

3.3 A critical pair algorithm for geometric rewrite-rules

In chapter 2 we showed how axioms of elementary geometry may be expressed by equations between conjunctions. The theory obtained is an equational theory for elementary geometry. The equations are applied as rewrite rules which have the character of the reduction rules defined in the previous section. With some adaptions for the special properties of conjunctions of predicates we may use the theoretical results, and the Knuth-Bendix completion algorithm, described in section 3.2 to complete the set of geometric reduction rules, or to prove that it is complete.

Definition 3.10 *(literal)*: A predicate, e.g. *a(P1, P2, P3, Alpha)*, *d(P1, P2, Dist)*, etc., or a congruence relation like e.g. *d(P1, P2) = d(P3, P4)*, etc. are called literals. The arguments of the predicates are terms (point variables or geometric operations).

The rewrite rules are of the form $c_1 \rightarrow c_2$, where c_1 and c_2 are conjunctions of literals. In order to prove the completeness of our set of rules we also need to take into consideration the *associativity* law: $(p \wedge q) \wedge w = p \wedge (q \wedge w)$ for conjunctions p, q, w,

and the *commutativity* law $p \wedge q = q \wedge p$. We cannot find an order function such that $p \wedge q$ reduces in a natural way to $q \wedge p$. The two conjunctions are intrinsically *incomparable*. A method that may be applied is to integrate the associativity and commutativity law into the unification algorithm. An algorithm for the so called *unification modulo AC* for conjunctions is described in [HSI 83] under the name of *BN-unification*. We use a similar definition for describing what we call AC-critical pairs of conjunctions for which we can prove that they lead us to a criterion for local confluence for the special type of rewrite rules used in our program.

Definition 3.11: *(separated).* Two conjunctions are *separated* if they do not share common literals.
Some examples: $p \wedge q \wedge r$ and $r \wedge s \wedge q$ are not separated since they share the literals q and r, neither are $p(a) \wedge q(x)$ and $p(a) \wedge r(y,z)$, however the conjunctions $p(a) \wedge q(x)$ and $p(x) \wedge r(y,z)$ are separated although they have the common predicate p (but $p(a)$ and $p(x)$ are not identical).

Let us consider two conjunctions s and t of the form: $s = s_1 \wedge s_2$, and $t = t_1 \wedge t_2$ with sub-conjunctions s_1, s_2, t_1, t_2. We define the conjunctions $s' := u \wedge s_1$, and $t' := v \wedge t_1$ where u and v stand for variable conjunctions.

Definition 3.12: *(unification modulo AC).* The conjunctions s and t are *unifiable modulo AC* if there are substitutions for the arguments of the literals σ', σ'' such that $\sigma' s_2 = \sigma'' t_2 = w$, w is a most general common instance of s_2 and t_2, s_1 and t_1 are separated, and $s' = t'$ i.e. $u \leftarrow t_1 \wedge w$, $v \leftarrow s_1 \wedge w$. Since in general there is a finite number of ways for splitting s ant t into sub-conjunctions there is a finite number of ways for unifying the two terms, and there is in general more than one, but a finite number of *most general unifiers*. By *BNU(s,t)* we mean a complete set of most general unifiers of conjunctions s and t. When the two terms s and t are separated the unifier will be a trivial one, namely $u \leftarrow t$, and $v \leftarrow s$. Since such a unification is useless, as we shall see later, we will treat the conjunctions as non unifiable.

Definition 3.13: (*Reduction of a fact-database*). The database of facts D to which the rewrite rules are applied may be interpreted as a conjunction of literals (the facts): $D = p_1 \wedge p_2 \wedge ... \wedge p_k$. Since conjunction is closed under *associativity* and *commutativity*, all permutations of of literals in D are equivalent ($D' = p_{\pi(1)} \wedge p_{\pi(2)} \wedge ... \wedge p_{\pi(k)} = D$ for some permutation π of the indices). We may therefore regard the expression as an unstructured set $D = \{p_1, p_2, ..., p_k\}$ *or* $D = \bigcup_j \{p_j\}$. We say the database D_i *reduces* to the database D_{i+1} by R ($D_i \rightarrow_R D_{i+1}$) if and only if there is a rule $l \rightarrow r \in R$ and a substitution σ such that for a subset $D' \subseteq D_i$, $D' = \sigma l$, and $D_{i+1} = (D_i - \sigma l) \cup \sigma r$. (With a difference $A - B$ we mean the set difference that is only defined if the set to be subtracted $B \subseteq A$).

Definition 3.14: (*AC-critical pair*). $\langle p,q \rangle$ is called an *AC-critical pair* if $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ is a pair of rules of R with unifiable left sides, l_1 is of the form $l_1' \wedge l_1''$, l_2 is of the form $l_2' \wedge l_2''$, if $\exists\ \sigma_1, \sigma_2$: $\sigma_1 l_1'' = \sigma_2 l_2''$ then $p = \sigma_1 l_1' \wedge \sigma_2 r_2$, $q = \sigma_2 l_2' \wedge \sigma_1 r_1$.

Theorem 3.6: '$\rightarrow_R$' is locally confluent if and only if for all AC-critical pairs p, q of R: $p \downarrow_R q$.

Proof of theorem 3.6:

"$\Leftarrow$"

Let us assume that for arbitrary databases S_1, S_2 and D_{init} where $S_1 \leftarrow D_{init} \rightarrow S_2$, we have to show that $S_1 \downarrow S_2$, i.e. $S_1 \rightarrow^* D_f^* \leftarrow S_2$ for some D_f. We also assume that there are rules $l_1 \rightarrow r_1$, $l_2 \rightarrow r_2 \in R$ and substitutions σ_1, σ_2 with $\sigma_1 l_1 \subseteq D_{init}$ and $\sigma_2 l_2 \subseteq D_{init}$ such that $S_1 = (D_{init} - \sigma_1 l_1) \cup \sigma_1 r_1$, and $S_2 = (D_{init} - \sigma_2 l_2) \cup \sigma_2 r_2$.

There are essentially two cases that need be considered:

In the first case $\sigma_1 l_1 \cap \sigma_2 l_2 = \emptyset$:
$S_1 \cap \sigma_2 l_2 = ((D_{init} - \sigma_1 l_1) \cup \sigma_1 r_1) \cap \sigma_2 l_2 = D_{init} \cap \sigma_2 l_2 = \sigma_2 l_2$, and
$S_2 \cap \sigma_1 l_1 = ((D_{init} - \sigma_2 l_2) \cup \sigma_2 r_2) \cap \sigma_1 l_1 = D_{init} \cap \sigma_1 l_1 = \sigma_1 l_1$, therefore
$(((D_{init} - \sigma_1 l_1) \cup \sigma_1 r_1) - \sigma_2 l_2) \cup \sigma_2 r_2 = D_f$ is defined, and

$(((D_{init} - \sigma_2 l_2) \cup \sigma_2 r_2) - \sigma_1 l_1) \cup \sigma_1 r_1 = D_f$ is defined as well, therefore $S_1 \downarrow S_2$.

In the second case $\sigma_1 l_1 \cap \sigma_2 l_2 \neq \emptyset$:
we call $w = \sigma_1 l_1 \cap \sigma_2 l_2$, therefore $\sigma_1 l_1' = \sigma_1 l_1 \setminus w$, and $\sigma_2 l_2' = \sigma_2 l_2 \setminus w$.
$S_1 = ((D_{init} - w) - \sigma_1 l_1') \cup \sigma_1 r_1$, and $S_2 = ((D_{init} - w) - \sigma_2 l_2') \cup \sigma_2 r_2$. S_1 and S_2 have a common successor if $\sigma_2 l_2' \cup \sigma_1 r_1$ and $\sigma_1 l_1' \cup \sigma_2 r_2$ have a common successor. It is easy to see that this condition for confluence corresponds to the confluence condition of instances of AC critical pairs. If critical pairs are confluent then all their instances are confluent, therefore $S_1 \downarrow S_2$.

"$\Rightarrow$"

We assume that D_{init} is a most general common instance of the left sides l_1 and l_2 of two rules in R. Therefore $S_1 = \sigma_1 r_1 \cup \sigma_2 l_2'$, and $S_2 = \sigma_2 r_2 \cup \sigma_1 l_1'$. From the confluence of S_1 and S_2 follows the confluence of the corresponding critical pair. ♦

We have implemented an *AC critical pair algorithm* in PROLOG for testing the confluence of the geometric rewrite-rules. An important part of the algorithm is "unification modulo AC" which is not supported by PROLOG, but may be implemented easily with the unification already given by the language. The rewrite-rules for the reflexivity-, symmetry- and transitivity-laws of the congruence relations and for the addition properties of vectors, angles and triangles are not reduction rules in the sense of our order function τ. The confluence of these rewrite-rule will be shown in section 3.4. They have been integrated in the unification of our Knuth-Bendix algorithm. Since the equivalence classes are finite and not pair-wise unifiable, there exist no critical pairs between them. For many predicates changing the order of the arguments yields an equivalent predicate. For example, the predicate *d(P1, P2, Dist)* is equivalent to *d(P2, P1, Dist)*. Such equivalences also needed to be integrated in the unification algorithm. They are independent from the other laws, and do not affect the above proofs. Our critical pair algorithm finds all critical pairs for the given set of rules. The critical pairs lead to new rules which are used to reduce or delete old rules, before they are added to the set of rules. This process is done iteratively as is described in section 3.2. The algorithm served us for testing the theory. Many earlier versions of the rewrite rules lead to contradictions not necessarily seen by looking at individual rules but arising as a consequence of their combination. It also became clear that for some rules in their original formulation it was impossible to obtain confluence with a

finite set of reduction rules. The critical pair algorithm was an important tool for guiding us to the right way of thinking about the problem, and finally, for proving the algorithmic completeness and the independence of the rules.

We now want to apply the critical pair algorithm to geometric rewrite-rules such as found in chapter 2. For the following discussion we leave out the terms that represent the construction operations, and treat the axioms as relations between points. We start with a set of 5 rules. The rules listed below are those rules using predicates for assigning constant values to point tuples (The procedure for the rules dealing with congruence relations is very similar).

```
1   [s(A,B, _), d(A,B, _)] -> [v(A,B, _)].
2   [p(A, _), v(A,B, _)] -> [p(A, _), p(B, _)].
3   [s(A,B, _), a(B,A,C, _)] -> [s(A,B), s(A,C, _).
4   [a(A,B,C, _), d(A,B, _), d(B,C, _)]
    -> [tr(A,B,C, _), d(B,C, _)].
5   [a(A,B,C, _), a(B,C,A, _)] -> [tr(A,B,C, _)] .
```

The Knuth-Bendix completion algorithm generates 7 more rules:

```
6   [a(A,B,C, _), v(B,C, _)] -> [v(B,C, _), s(A,B, _)]
```

was found from a critical pair of rules 1 and 3

```
7   [v(A,B, _), tr(A,B,C, _)] -> [v(A,B, _), v(B,C, _)]
```

was found from a critical pair of rules 1 and 4

```
8   [s(A,B, _), s(B,C, _), s(C,A, _)]
    -> [s(A,B, _), tr(A,B,C, _)]
```

was found from a critical pair of rules 3 and 5

```
9   [v(A,B, _), s(A,C, _), s(B,C, _)] -> [v(A,B, _), v(A,C, _)]
```

was found from a critical pair of rules 1 and 8

```
10  [p(A, _), p(B, _), a(C,A,B, _)]
                -> [p(A, _), p(B, _), s(A,C, _)]
```

was found from a critical pair of rules 2 and 6

```
11  [tr(A,B,C, _), p(A, _), p(B, _)]
    -> [p(A, _), p(B, _), p(C, _)]
```

was found from a critical pair of rules 2 and 7

```
12  [p(A, _), p(B, _), s(A,C, _), s(B,C, _)]
    -> [p(A, _), p(B, _), p(A, _)]
```

was found from a critical pair of rules 2 and 9

Finally, we obtain a set of 12 rules that are locally confluent, i.e. for all critical pairs p, q the relation $p\downarrow_R q$ holds. For obtaining the above result we had to also introduce the so called "inconsistency rules", that are used to detect when the database of facts is inconsistent. As an example, the rule *[d(A,B, _), d(A,B, _)] -> [false]* indicates that representing the distance between points A and B by two facts in the database is either redundant or contradictory. Similarly, for slopes we define *[s(A,B, _), s(A,B, _)] -> [false]*. These two rules build critical pairs with rule (1) of above, which introduce the following two new inconsistency rules: *[v(A, B, _), d(A, B, _)] -> [false]* , and *[v(A, B, _), s(A, B, _)] -> [false]*. These rules may be interpreted as follows: The knowledge of the relative vector between two points already implies the knowledge of the slope and the distance between these points, therefore explicitly stating these constraints in combination is redundant or contradicting. More such inconsistency rules that find contradicting conjunctions of predicates by a reduction to *'false'* have been created from critical pairs by the algorithm (they are not listed here). Many critical pairs that have been found do not result in a new rule, but disappear later on, as soon as additional rules are introduced. For example, the critical pair *<[v(A, B, _), d(A, B, _)] , [v(A, B, _), s(A, B, _)]>* is reduced to a trivial critical pair *<false, false>*, as soon as the two inconsistency rules of above are introduced.

Theorem 3.7. The algorithm based on the final set of rules R obtained from EQ by means of the order function τ, and the Knuth-Bendix completion algorithm, as described, is a decision algorithm for the theory Γ_{CR} (A subset of elementary Euclidean geometry, for compass and ruler constructions).

Proof of theorem 3.7: The proof follows from the confluence of the rules and the Church-Rosser property stated by lemma 3.2 (see diagram, fig. 3.8)..

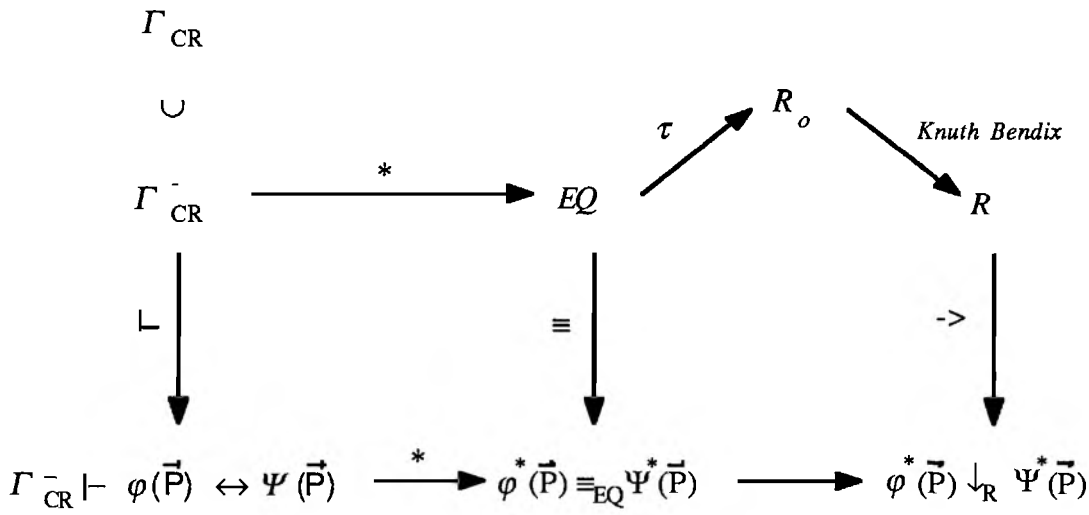


fig. 3.8

Forward vs. backward reasoning. The information associated with the predicates retracted from the database by application of rewrite-rules is still implied by the new predicates. Instead of adding the inverse rule (which would cause termination problems) we define some implications. Such implications can be represented in Prolog by so called "Prolog-rules". The notation of a Prolog rule is as follows: "A :- B." means that A is implied by B, where A is a predicate (the head of the rule), and B is a conjunction of predicates (the body of the rule). The Prolog backtracking mechanism is used to derive the truth of predicates either directly from facts '[ ]', or indirectly by applying Prolog-rules. Together with the rewrite-rule d_3 we define the following

implications.

```
a(P1,P2,P3) = a(P4,P5,P6) :-  [tr(P1,P2,P3) = tr(P4,P5,P6)]
a(P2,P3,P1) = a(P5,P6,P4) :-  [tr(P1,P2,P3) = tr(P4,P5,P6)]
a(P3,P1,P2) = a(P6,P4,P5) :-  [tr(P1,P2,P3) = tr(P4,P5,P6)]
```

Another axiom of Euclidean geometry used in our system is expressed by the rewrite-rule r_2.

```
r2:
[a(P1, P2, P3) = a(P4, P5, P6), s(P2, P3) = s(P5, P6)]
          -> [s(P2, P3) = s(P5, P6), s(P1, P2) = s(P4, P5)]
```

The corresponding implication is:

```
a(P1, P2, P3) = a(P4, P5, P6):-
          [s(P1, P2) = s(P4, P5)], [s(P2, P3) = s(P5, P6)].
```

Here the importance of the order of the arguments becomes clear. If we reversed the order of points in one of the predicates the rule would be in contradiction to its geometric meaning.

It is a general principle of our program to apply the *rewrite-rules* are in one direction (for replacing the predicates in the database with new predicates, and thus bringing them to a certain normal form), and the *implications* in the other direction (for deriving predicates implied by other predicates). The method of applying rewrite-rules is also called *forward reasoning*, whereas the built-in reasoning mechanism of Prolog is called *backward reasoning*. Both reasoning mechanisms are combined in this program. The implications used in combination with the so called construction rules introduced at the beginning of this section may be interpreted as measuring operations. The following implication may be used for measuring the distance between two points:

```
distance(P1, P2, D):-

    p(P1, [X1,Y1]),
```

```
p(P2, [X2,Y2]),

D is sqrt((X2 - X1) *  (X2 - X1) + (Y2 - Y1) * (Y2 - Y1)).
```

Here forward reasoning is used for construction operations and backward reasoning is used for measuring operations.

3.4 Representing equivalence classes

Congruence relations, like e.g. *d(P1,P2) = d(P3,P4)*, are of great importance to geometry; they are equivalence relations, satisfying the reflexivity-, symmetry-, and transitivity laws. The present section describes how equivalence relations are treated in our system. To treat the issue in a more general way here we abstract from the geometric meaning, and write an equation $X = Y$, where X and Y stand for predicates. An equation may be syntactically represented as a directed graph (an arrow going from X to Y). The system of equations $A = B, B = C, D = C$, and $D = E$ is represented by the graph shown in fig. 3.9

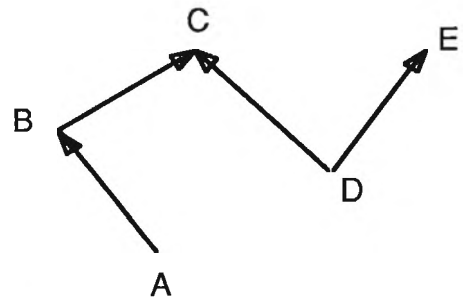


fig. 3.9 Representing equations by a directed graph

We use the graph notation to illustrate the laws of equivalence classes (fig. 3.10)

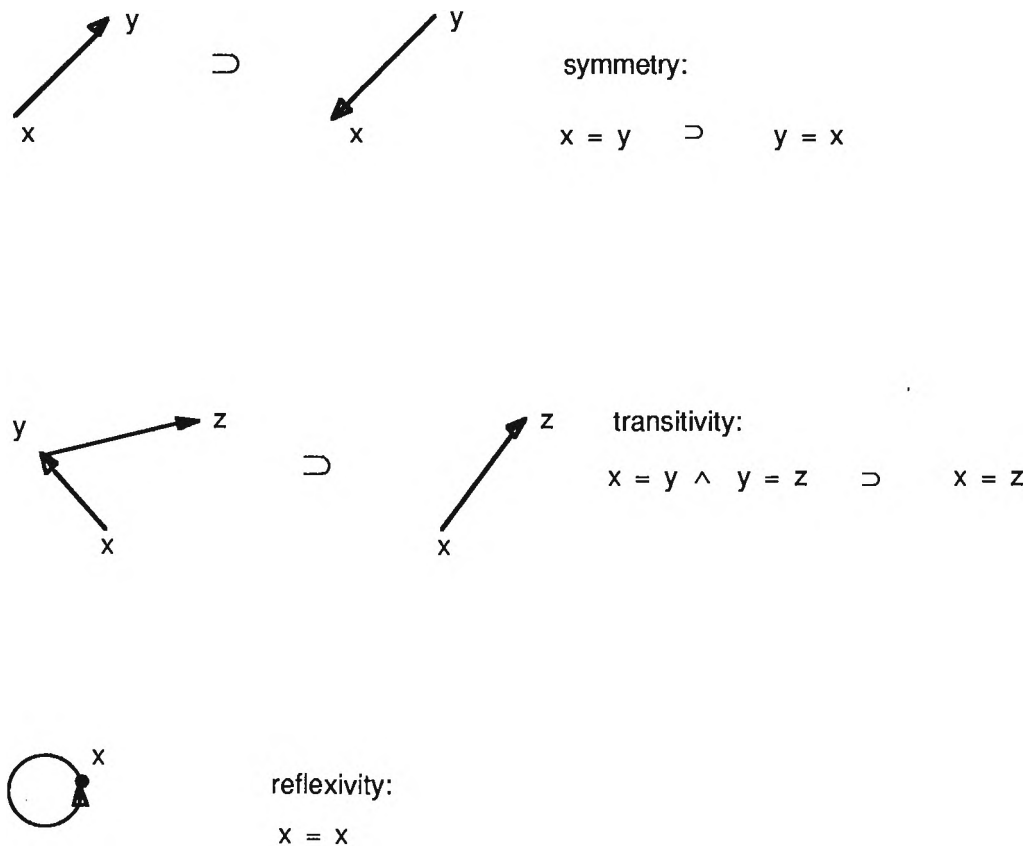


fig. 3.10 The laws of equivalence classes represented with directed graphs

To find out whether two objects U, V (here vertices of a graph) belong to the same equivalence class we would have to apply the above laws repeatedly to the graph, and to infer a directed edge from U to V. The representation of equivalence classes by directed graphs, as shown in fig. 3.9, is not unique and solving such a problem requires a graph search algorithm. A unique and complete representation of the equivalence class in fig. 3.9 is the reflexive-, transitive-, symmetric closure of the graph (see fig. 3.11) which is a complete and undirected graph for the vertices A, ..., E.

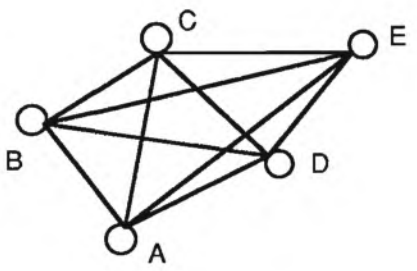
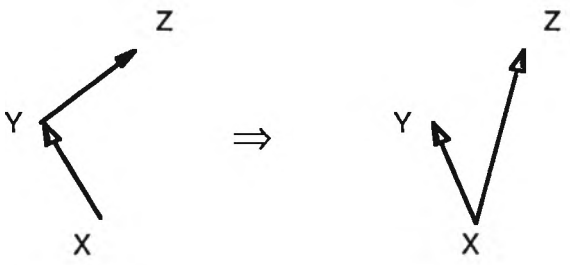
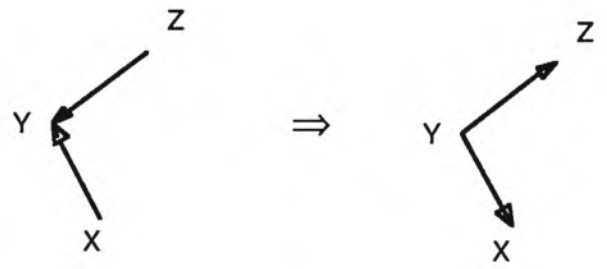


fig. 3.11 transitive closure of the equation system

The information is explicit, but certainly redundant. For n equivalent objects the n^2 edges need to be generated by some preprocessing algorithm. We were looking for a representation that a) is unique, b) does not store information redundantly, and c) fits into our system of rewrite-rules such that we can apply the same inference mechanism as for the other rules. We define two rewrite-rules $e1$ and $e2$ (shown graphically in fig. 3.12).



e1: [X = Y, Y = Z] -> [X = Y, X = Z]



e2: [X = Y, Z = Y] -> [Y = X, Y = Z]

fig. 3.12 rewrite rules for congruence relations

Applying these rewrite-rules to the graph in fig. 3.9 results in a tree of depth 1 (fig. 3.13) where *n-1* vertices are directly connected to one vertex (the root of the tree).

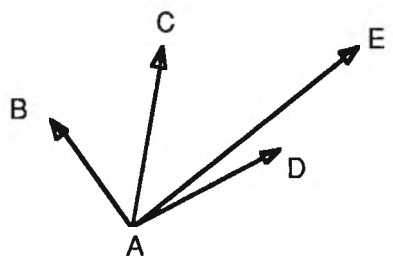


fig. 3.13 An equivalence class represented by a graph in tree form.

Lemma 3.1. An inference mechanism applying the rewrite-rules e_1 and e_2 to a set of m equations represented as a directed graph with $m + 1$ vertices stops after finitely many steps resulting in a tree representation of depth 1 as shown in fig. 3.13. The representation is unique up to the choice of the root vertex.

Proof of Lemma 3.1. For the proof we make the further assumption that rule e_1 is

applied with higher priority, i.e. rule *e2* is only applied when rule *e1* does not apply. An arbitrary graph may be viewed as consisting of several partial trees that point to each other pair-wise.

a) By applying rule *e1* to each of the partial trees the connection of each node is propagated to the root of the partial tree. After a while all partial trees are in normal form (trees of depth *1*, see fig. 3.15), and *e1* does no longer apply.

b) By applying e2 to one connected pair of partial trees the two directed edges pointing to each other reverse their direction. (fig. 3.15). This makes their common vertex the new root, and the two partial trees the sub-trees of the now merged tree. By repeated application of *e1* this new tree is brought to normal form.

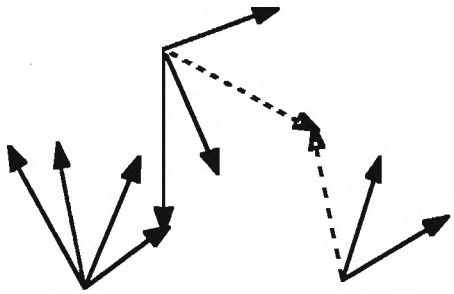


fig. 3.14

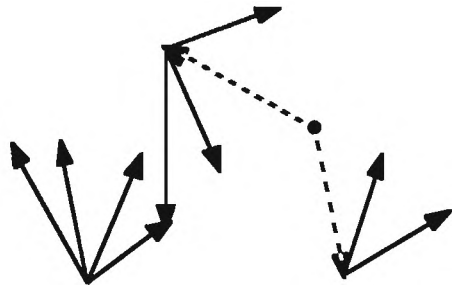


fig. 3.15

Whenever *e1* does not apply *e2* merges two partial trees as described under b). This works until the complete tree is in normal form. The root of the final tree will be the common vertex of the two partial trees merged last. ♦

To decide that two objects, *X* and *Y*, belong to the same equivalence class once the equations are in normal form means to check if either $X = Y$ is stated explicitly, or is implied by the other equations which are in normal form. In PROLOG we express the

implications by the following rules for the predicate '*congruent*':

```
congruent(X, Y):- [X = Y].
congruent(X, Y):- [Y = X].
congruent(X, X):- true.
congruent(X, Y):- [Z = X], [Z = Y].
```

The method for treating equivalence classes described in this section is applied to all congruence relations, such as $d(P1, P2) = d(P3, P4)$, $s(P1, P2) = s(P3, P4)$, $a(P1, P2, P3) = a(P4, P5, P6)$, etc.

Addition properties of vectors, angles and triangles. The addition properties of vectors, angles and triangles can be treated in a similar way as the congruence relations above. Some examples:

- If we know a relative vector going from point A to point B, by $v(A, B, v_1)$, and a vector from B to C by $v(B, C, v_2)$, the knowledge of a vector from A to C is implied, namely $v(A, C, v_1 + v_2)$ *(transitivity)*.

- If we know the vector $v(A, B, v)$, the vector in the opposite direction $v(B, A, -v)$ is also known *(symmetry)*.

- For every point P the vector $v(P, P, 0)$ is known to have zero length *(reflexivity)*.

We need some means for treating the transitivity-, symmetry- and reflexivity-properties of the predicate *'v'*. Like above, we write some rewrite-rules that bring the vectors to normal form:

[v(A,B, v_1), v(B,C, v_2)] -> [v(A,B, v_1), v(A,C, $v_1 + v_2$)]

[v(A,B, v_1), v(C,B, v_2)] -> [v(B,A, $-v_1$), v(B,C, $-v_2$)]

and express the corresponding implications by PROLOG rules:

```
v(A,B, V):-          [v(A,B, V)].
v(B,C, V2 - V1):-    [v(A,B, V1),v(A, C, V2)].
v(B,A, -V):-         [v(A,B, V)].
v(X,X, 0):-          true.
```

Very similar rules have been found for treating the addition properties of angles and triangles (they are not explicitly given here). By means of these rules we may not only express properties for simple triangles but also for polygons specified by constraints,

4 Conclusion

An advantage of the symbolic, geometric approach is the closeness of the language in which it is realized to the language of geometric applications, for instance interactive Computer Aided Design (CAD). The predicates and functions used in the algorithm have a direct geometric interpretation. Therefore it is relatively straight forward to develop a user interface that explains the results in words or graphically. In the implemented prototype system the graphical user interface and the numerical evaluation of the symbolical expressions are realized with the language interface of the Prolog interpreter with the procedural programming language Modula-2 (see [MUL 85] and [WIR 83]). An application of the constraint solver for 3-dimensional geometric modelling is described in [SOB 91].

The rewrite-rules arise from equations of formulas. Therefore applying these rules cannot invent new points that don't occur in the description of the problem, although this is sometimes necessary for finding a proof or a construction. Theoretically it is

possible to have the computer invent such points, but then the program would be intractable for real-life problems. The intended application of the system is to support the interactive user in finding a solution when constructing geometric objects. Also the system may be used for interactively proving elementary geometry theorems. The strength of the system is that it helps detecting inconsistencies in the definition (in an interactive session such inconsistencies are quite often) and gives hints where to add constraints.

References

[BUL 82] B. Buchberger and R. Loos. Algebraic Simplification. Computing, Suppl. 4, 11, (1982) pp. 11 - 43

[BRU 86] B. Brüderlin. Constructing Three-Dimensional Geometric Objects Defined By Constraints. 1986 Workshop on Interactive 3D Graphics, Conference Proceedings, Chapel Hill, North Carolina, published by ACM Siggraph, 1986

[BRU 87] B. Brüderlin. Rule-Based Geometric Modelling. Ph.D. thesis, ETH Zürich, Switzerland, Verlag der Fachvereine, vdf-Verlag, Zürich 1987 (ISBN 3 7281 1638)

[CHO 84] Shang-Ching Chou. Proving Elementary Geometry Theorems Using Wu's Algorithm. Contemporary Mathematics, Volume 29, 1984, American Mathematical Society, pp 234 - 287

[CHS 86] Shang-Ching Chou and William F. Schelter. Proving Geometry Theorems with Rewrite Rules. Journal of Automated Reasoning 2 (1986) pp. 253 - 373

[CHO 86] Shang-Ching Chou. A Collection of Geometry Theorems Proved Mechanically. Technical Report 50, July 1986. Institute for Computing Science, University of Texas at Austin

[CLM 81] W.F. Clocksin, C.S. Mellish. Programming in Prolog. Springer Verlag, Berlin, Heidelberg, New York 1981

[COP 86] Helder Coelho, Luiz Moniz Pereira. Automated Reasoning in Geometry Theorem Proving with Prolog. Journal of Automated Reasoning 2 (1986)

[COL 75] G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. Lecture Notes in Computer Science No. 33, pp. 134 - 183, Springer Verlag, 1975.

[ENG 83] E. Engeler. Metamathematik der Elementarmathematik. Springer Verlag, Berlin, Heidelberg, New York 1983

[GEL 59] H. Gelernter. Realization of a Theorem Proving Machine, 1959. Published in "Automation of Reasoning" Vol.1, Springer Verlag, 1983

[GOS 83] James Gosling. Algebraic Constraints. Ph.D. thesis, Carnegie-Mellon University, May 1983

[HIL 71] D. Hilbert. Foundations of Geometry. Open Court Publishing Company, La Salla, Illinois 1971

[HUC 86] Ulrich Huckenbeck. Geometrische Maschinenmodelle (german). PhD. thesis, Universität Würzburg, Germany, 1986

[HSI 83] Jieh Hsiang. Topics in Automated Theorem Proving and Program Generation. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1983

[HOA 69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM. Vol. 12 No. 10, 1969

[KAM 88] D.Kapur, J.L. Mundi. Special Volume on Geometric Reasoning. Artificial Inteligence, Volume 37, Numbers 1-3, December 1988.

[KNB 70] D.E. Knuth, P.B. Bendix. Simple Word Problems in Universal Algebra. Computational Problems in Abstract Algebra. Conference Proceedings, Oxford 1967, J. Leech ed., Pergamon 1970

[KOH 85] H.-P. Ko and M.A. Hussain. ALGE-PROVER. An Algebraic Geometry Theorem Proving Software. Report No. 85CRD139, july 1985. Technical Information Series, General Electric

[LIG 82] R. Light, D. Gossard. Modification of geometric models through variational geometry. CAD vo. 14, No. 4, Butterworth 1982

[MUL 85] C. Muller. Modula -- Prolog, User Manual. Rep. No. 63, July 1985. Inst. für Informatik, ETH Zürich, Switzerland

[NEL 85] G. Nelson. Juno, a constraint-based graphics system. 1985 ACM Siggraph Conference Proceedings

[POP 86] R.J. Popplestone. The Edinburgh Designer System as a Framework for Robotics

[ROB 65] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle, JACM, Vol. 12, No. 1, Jan. 1965, pp. 23-41.

[SCH 88] F. Schmid. A Symbolic Approach to Solving Formulas in Projective Geometry. Ph.D. Thesis, ETH, Switzerland, 1988.

[SCH 75] Peter Schreiber. Theorie der geometrischen Konstruktionen (german). VEB Verlag der Wissenschaften, Berlin, 1975

[SST 83] W. Schwabhäuser, W. Szmielev, A. Tarski. Metamathematische Methoden in der Geometrie (german). Springer Verlag Berlin, Heidelberg, New York 1983

[SHA 78] Michael Ian Shamos. Computational Geometry. Ph.D. thesis, Yale University, New Haven, Connecticut, 1978.

[SUT 63] I. Sutherland. Sketchpad, A Man-Machine Graphical Communication System. Ph.D. thesis, MIT, January 1963

[SOB 91] W. Sohrt, B. Bruderlin, Interactive Geometric Modelling with Constraints. To appear in proceedings of the 17th Annual Conference on Advances in Design and Manufacturing, Austin Texas, Jan 1991.

[TAR 51] A. Tarski. A Decision Method for Elementary Algebra and Geometry. Univ. of Calif. Press, Berkeley, 1951

[WIR 83] N. Wirth. Programming in Modula-2. Texts and Monographs in Computer Science. Springer Verlag Berlin, Heidelberg, New York 1983

[WUW 84a] Wu Wen-tsün. Some Recent Advances in Mechanical Theorem Proving of Geometries. Contemporary Mathematics, Volume 29, 1984, American Mathematical Society, pp. 235 - 241

[WUW 84b] Wu Wen-tsün. On the decision Problem and the mechanization of Theorem-Proving in Elementary Geometry. Contemporary Mathematics, Volume 29, 1984, American Mathematical Society. pp. 213 - 23

[WUW 86] Wu Wen-tsün. Basic Principles of Mechanical Theorem Proving in Elementary Geometries. Journal of Automated Reasoning 2 (1986) pp. 221 - 252