LT/GEOL 1.0 : A MAPLE package for constrained planar euclidean geometric structures

L.W. Ericson

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 101

Programme 1

LT/GEOL 1.0: A MAPLE PACKAGE FOR CONSTRAINED PLANAR EUCLIDEAN GEOMETRIC STRUCTURES

Lars Warren ERICSON

Décembre 1988

★RT_0101★

LT/GEOL 1.0: Un ensemble de fonctions en MAPLE pour la représentation de contraintes géométriques dans le plan complexe

LT/GEOL 1.0: A MAPLE package for constrained planar Euclidean geometric structures

November 18, 1988

Lars Warren Ericson [1]
INRIA
B.P. 105-78153
Le Chesnay CEDEX, France

Abstract

This paper describes a package in MAPLE for representing geometric structures such as points, lines and circles in the complex plane, with geometric constraints between structure components involving measures such as angle and distance. This package is a component of the LINETOOL planar geometric editor proposed by Ericson and Yap (1988), but is sufficiently general to be used in other symbolic geometry applications.

Resumé

Cet article décrit un ensemble de fonctions en MAPLE pour la représentation de structures géométriques comme des points, des lignes, et des cercles dans le plan complexe, avec des contraintes géométriques entre les composants de la structure, incluant des mesures comme l'angle et la longueur. Cet ensemble de fonctions est une partie du système LINETOOL de calcul formel géométrique planaire proposé par Ericson et Yap (1988), mais il est suffisamment géneral pour être utilisé dans d'autres applications du calcul formel géométrique.

[1]This work was done using the facilities of Projects FORMEL (Gérard Huet) and ALGO (Philippe Flajolet) at INRIA. I am supported by a Bourse Chateaubriand from the Ministry of Foreign Affairs of the French Government, for the period 15 January 1988 to 15 February 1989, hosted by the École Nationale Supérieure des Télécommunications in Paris.

Contents

List of Tables

List of Figures

Acknowledgements

I thank Professor Chee-Keng Yap of the Courant Institute of Mathematical Scienes at New York University for his helpful critical reading of some earlier versions of this document. I also thank Robert Ehrlich of INRIA for his help with the SMSCRIPT interactive POSTSCRIPT previewer, which he has installed and maintained for the groups of Bâtiment 8. Without SMSCRIPT this document would have been much more difficult to produce. A conversation with Virginie Collette helped me make a clearer presentation of the example of the final section. I thank Mireille Regnier for helping me with the French on the title page.

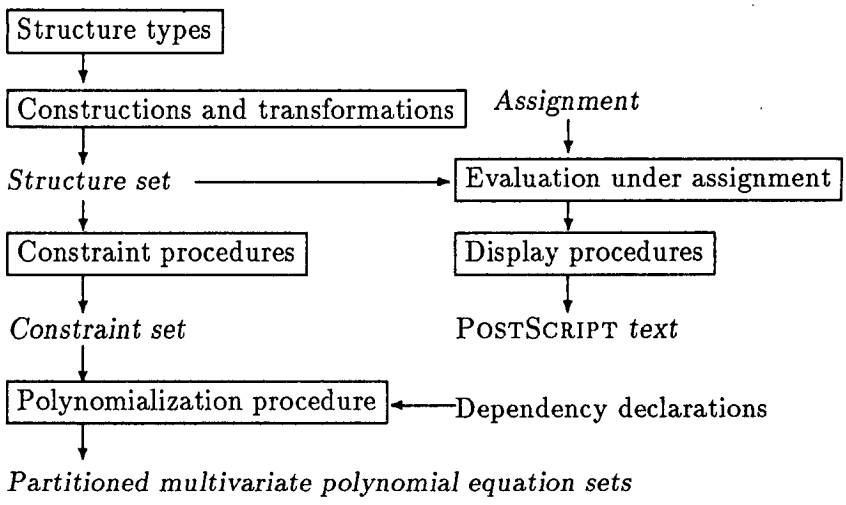

Figure 1: Geometric structure description package

1 Introduction

This paper describes the MAPLE package LT/GEOL version 1.0 for defining planar geometric structures and obtaining their analytic models $\langle E, D \rangle$ as sets E of multivariate polynomial equations together with a dependency relation D on real-valued indeterminates which occur in expressions defining components of geometric structures. The "leaf nodes" of geometric structures are numbers, which are any computable subfield of the reals completed under the inclusion of two elements ∞ and $\perp$. Figure 1 shows the major components of the package.

LT/GEOL is an implementation component of the LINETOOL geometric editor proposed by Ericson and Yap (1988) [11, §3–§5], but is sufficiently general to be used by similar applications. The paper assumes that the reader is an experienced MAPLE programmer [7,6].

1.1 Numbers

The package is parameterized on the existence of a computable subfield of the real numbers which is completed by infinity ∞ and undefined $\perp$ elements, according to the arithmetic rules given by Vuillemin [15, §2.6]. Such a subfield could be the algebraic numbers, for example via Loos' algorithms [12], or via Vuillemin's effective real numbers.

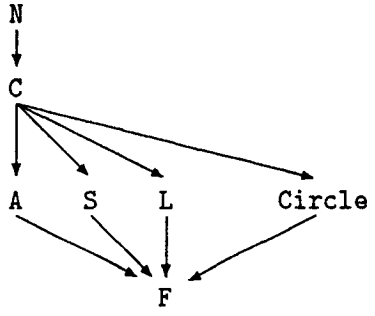

Figure 2: Genealogy of geometric structures in LT/GEOL

1.2 Geometric structures

The package operates on (geometric) *structures* (see §2) such as real numbers `N`, complex numbers `C` (the representation for points in LT/GEOL), angles `A`, lines `L`, line segments `S` and circles `Circle` in the plane, with geometric constraints between structure components involving measures such as angle and distance. The hierarchy of definition for these predefined geometric structure types is shown in Figure 2.

Structures are labelled product types. *Intersection* structures are labelled disjoint union types. The package allows new kinds of structure to be declared. For each kind of structure, by convention one defines a collection of constructions and transformations (a kind of construction), constraints, an *evaluation* method under an assignment of values to variables, and a POSTSCRIPT [2] *graphical display* under an *assignment* of values to variables. A *graphical display* is a text string of valid POSTSCRIPT code. An *assignment* is a set of equations of indeterminates to expressions. Structures may be grouped into figures, which are collections of structures and constraints. In a language such as CAML, these structures would be defined as mutually recursive abstract types [10,16].

1.3 Constructions and constraints

A *construction* is a procedure which yields an instance of a structure. A *constraint* is a procedure which yields a set of *general equations* involving components of the structures which are arguments to the constraint. A *general equation* is an expression $e_1 \phi e_2$ where e_1, e_2 are number-valued multivariate rational functions and ϕ is one of $\{=, \neq, <, >, \leq, \geq\}$. A *transformation* is an action on all objects in a figure in the plane. Thus, one set of transformations is defined on figures, which is recursively defined in terms of transformations on all geometric structures which may occur in a figure.

1.4 Intersections

An intersection is a point in the complex plane. In the case where two objects have no intersection in the complex plane, the components of the intersection point will have undefined $\perp$ or infinite ∞ values, depending on the order of evaluation of the arithmetic expression which defines the intersection.

1.5 Solution of constraints

The package is parameterized on the existence of a procedure for solving systems of multivariate polynomial equations Σ, with the result being a *variety* V, which in this paper should be taken to mean a set of assignments of real numbers to indeterminates such that each assignment in V when substituted into Σ will cause all of the polynomials in Σ to evaluated to 0. In a special case, this procedure could be Gaussian elimination. More generally it could be Gröbner bases [4].

1.6 Multivariate polynomial set normal form

The package includes facilities for obtaining the multivariate polynomial set normal form of a set of constraints. The package includes methods of transforming constraint sets, which are in the form of rational function general equations, entirely into sets of polynomials equated to zero or into sets of rational function inequalities and polynomials equated to zero. These translation facilities compute *normal forms* of constraint sets which are suitable for various theorem-proving and solution algorithms such as Wu's method [8,9] and Gröbner bases [4].

1.7 Hierarchical multidimensional systems

The LT/GEOL package (§11) provides a structure for maintaining an acyclic directed graph of *dependency assertions* relating subsets of the set V of `N`-valued indeterminates occurring in a set Σ of polynomials implicitly equated to zero. This structure also computes a corresponding function on Σ which, for a given node V of the graph, gives the subset of Σ which involves only indeterminates in V and indeterminates which elements of V depend on. This structure was proposed for use in geometric editing by Ericson and Yap (1988) [11].

The *validity* of dependency assertions is not checked by the system: it is up to the user to know what assertions are intended to be valid in the geometrically describable (abstracted physical) system that is under construction. Often such assertions are a matter of experimention: make the assertion, then see if the system is solvable by parts under this assumption.

1.8 Geometric expression evaluation

An evaluation procedure `evalG` is supplied, which takes MAPLE expressions which involve applications of overloaded operation names such as + and `PS`, and takes an assignment of structure types to indeterminates in the expression, and applies the correct operation. `evalG` is described in §12.

2 Structures

This section gives a facility for representing a labelled product type, here called a *structure* (like a C ***struct***), in LT/GEOL.

2.1 Declaring a structure type

A *structure type* g = ⟨a, b, c⟩ is a labelled product type. It may be declared in LT/GEOL with the MAPLE procedure call ***structure***(g,[a,b,c]).

In no case should the user's code at any time directly assign a value to the MAPLE *global names* g, `g Fields`, a, b *or* c *after defining a structure with those names with* ***structure***(g,[a,b,c]). But the same field name (such as b) *may* be used in more than one structure definition.

Example 1 The definition of the structure of *points* Point = $\langle x, y \rangle$ could be ***structure***(Point,[x,y]).

2.2 Making an instance of a structure type

An instance of structure type g, with actual values a, b, c for the fields a, b and c, is then obtained with the MAPLE procedure call g(a, b, c). This instance is known to the MAPLE ***type*** function as having type g.

2.3 Accessing the fields of a structure type

The field a (respectively b, c) may then be access for an instance x of structure type g with the MAPLE neutral operator expression a &of x. If x is a name, then the call is returned unevaluated.

$+$	0	1	∞	$\perp$	$\times$	0	1	∞	$\perp$
0	0	1	∞	$\perp$	0	0	0	$\perp$	$\perp$
1	1	2	∞	$\perp$	1	0	1	∞	$\perp$
∞	∞	∞	$\perp$	$\perp$	∞	$\perp$	∞	∞	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$-$	0	1	∞	$\perp$	$\div$	0	1	∞	$\perp$
0	0	-1	∞	$\perp$	0	$\perp$	0	0	$\perp$
1	1	0	∞	$\perp$	1	∞	1	0	$\perp$
∞	∞	∞	$\perp$	$\perp$	∞	∞	∞	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 1: The ∞-$\perp$ completion of a real subfield

3 Numbers

The *number* structure type `N` is intended to represent any computable subfield of the reals completed by the adjoinment of infinity (∞) and undefined ($\perp$) elements, according to the operation tables of Table 1 [15, §2.6]. In this paper, this is called the ∞-$\perp$ *completion* of a real subfield. Authors who have discussed arithmetic for suitable real subfields include Loos (1983; algebraic numbers [12]), Norman (1983; transcendental numbers [13]) and Vuillemin (1987; effective real numbers [15]).

3.1 Number structure

A *number* is a quantity v encased in an `N` structure, such that v is of a type which is a ∞-$\perp$ completion of a computable subfield of the reals. Table 2 shows the constructions, constraints, graphical display and evaluation operations provided by the LT/GEOL package for the `N` structure. The values `Infinity` and `Bottom` are special possibilities for v.

3.2 Number constructions

The basic number constructions are the arithmetic operations $+, -, \times, \div$ and exponentiation. In this package the default is MAPLE arithmetic. Other arithmetics may be installed by simply redefining the `N` arithmetic operations defined below.

The following numbers may be constructed from *complex numbers* $z = x + yi$ and z_1 in the complex plane (see §4):

Structure	
⟨v⟩	Any subset of the reals
Constructions	
+, −, ×, ÷, ^, Max, Min	N × N → N
−, Inverse, Sqrt	N → N
Re &of z, Im &of z	C → N
x &of α, y &of α	A → N
Modulus	C → N
Distance, Slope	C × C → N
XDistance, YDistance	C × C → N
Sine, Cosine, Tangent	A → N
Distance	C × L → N
Distance	S → N
R &of c	Circle → N
Constraints	
$=, \neq, <, >, \leq, \geq$	N × N → General equation set
Evaluation	
FValue	N × Assignment → float
Graphical Display	
PS	float → string sequence

Table 2: The N structure

- The x and y components, via the MAPLE expressions `Re &of z` and `Im &of z`.
- The *modulus* $|z| = \sqrt{x^2 + y^2}$.
- The *distance* between z and z_1, that is,

$$d_{z,z_1} = |z_1 - z|$$

- The x- and y-distance between z and z_1 (cf. [8, functions X, Y]).
- The *slope* of the line passing through z and z_1.

Some numbers that may be constructed from *angles* (see §5) include:

- The *sine* of an angle.
- The *cosine* of an angle.
- The *tangent* of an angle.

A number that may be constructed from a *line* (see §7) is the distance from a complex number to a line, which is the distance from the complex number to its projection on the line.

A number that may be constructed from line *segments* (see §6) is the length of a segment (cf. [14, §3(d)]).

3.3 Number constraints

The number constraints are equality and the inequalities.

3.4 Number evaluation under an assignment

An `N` may be evaluated under substitution with procedure `FValue` to an `N` or to a MAPLE expression involving `floats` and indeterminates. This is useful when computing graphical displays when one is confident that all indeterminates in the expression defining a number will be assigned numeric values.

3.5 Number graphical display

`PS` gives the POSTSCRIPT string for a number which has been approximated to a floating point value.

4 Complex numbers

4.1 Complex number structure

A complex number $C = \langle \texttt{Re}, \texttt{Im} \rangle$ consists of a real part `Re` and an imaginary part `Im`. Both `Re` and `Im` are `N` structures (see §3). Table 3 shows the constructions, constraints, transformations, evaluation and graphical display operations provided by the LT/GEOL package for the `C` structure.

4.2 Complex number constructions

The following constructions are defined on complex numbers:

- The arithmetic operations $+, -, \times, \div, \hat{}$.
- Scalar $\times, \div$.
- The *midpoint* of two other complex numbers.

A complex number may be constructed from an *angle* (see §5) by coercion: the argument is the angle, and the modulus is taken to be 1.

Some complex numbers that may be constructed from *lines* (see §7) are:

- Any complex number on the line with representing complex number pairs (u, w) is of the form $un + w$ [14, §3(p)].
- The intersection complex number of two lines with defining complex numbers u, w and u_1, w_1. The `Intersect` operation may result in a complex number, a line, or nothing. The intersection on the complex number representation of lines is defined as follows [14, §3(e)]:

$$((w_1\bar{u}_1 - \bar{w}_1 u_1)u - (w\bar{u} - \bar{w}u)u_1)(u\bar{u}_1 - \bar{u}u_1)^{-1}$$

- The *projection* P_ℓ of a complex number P onto a line ℓ, which is the intersection of a perpendicular to the line through the complex number, and the line itself.
- The *reflection* R_ℓ of a complex number P across a line ℓ, which is equal to $2P_\ell - P$.
- The *slope* of the line passing through two complex numbers, as a number (undefined if the two numbers are both purely imaginary).

Structure	
⟨Re, Im⟩	N × N
Constructions	
−, Inverse, Sqrt, Conjugate	C → C
+, −, ×, ÷	C × C → C
×	N × C → C
÷	C × N → C
C	A → C
MidPoint	C × C → C
p1 &of *s*, p2 &of *s*	S → C
u &of *l*, w &of *l*	L → C
Projection, Reflection	C × L → C
On	L × N → C
Intersection	L × L → C
z &of *c*	Circle → C
Intersection	L × Circle → C × C
Intersection	Circle × Circle → C × C
On	Circle × N → C
Transformations	
Rotate	C × A → C
Translate	C × C → C
Scale	C × N → C
Constraints	
=, ≠, <, >, ≤, ≥	C × C → General equation set
Evaluation	
FValue	C × Assignment → C
Graphical Display	
PS	C × string set → string sequence

Table 3: The C structure

Example 2 Parallel lines do not intersect. Thus, an attempt at intersecting two parallel lines will result in a point with ∞ or $\perp$ components.

```
> L1 := evalG(L(C(-1,0), C(1,0)));
                    L1 := L(C(N(-2), N(0)), C(N(1), N(0)))

> L2 := evalG(L(C(-1,1), C(1,1)));
                    L2 := L(C(N(-2), N(0)), C(N(1), N(1)))

> evalG(Intersection(L1, L2));
                          C(N(Bottom), N(Bottom))
```

From a line (u, w) and a circle (R, z), two complex numbers A, B may be constructed which represent the two intersection points [14, §3(n)]. A and B are the two values of

$$z + (2\bar{u})^{-1}((w-z)\bar{u} - u(\overline{w-z}) \pm (((w-z)\bar{u} - u(\overline{w-z}))^2 + 4R^2u\bar{u})^{1/2})$$

From a circle and a circle, two complex numbers A, B may be constructed which represent the two intersection points. These are the points of intersection of the *chord* of intersection of the circles (see §7), with one of the circles.

A *complex number on a circle* (see §8) is of the form [14, §3(q)], for n any number, $Rn + z$.

4.3 Complex number transformations

A complex number may be rotated by an *angle*, scaled by a *number* or translated by another complex number. The *rotation* of a complex number $r(\cos\theta + i\sin\theta)$ by an angle α is the complex number $r(\cos(\theta+\alpha) + i\sin(\theta+\alpha))$. The *scaling* of a complex number $x + yi$ by r is the complex number $xr + yri$. The *translation* of a complex number u by a complex number w is just $u + w$.

4.4 Complex number constraints

On complex numbers we have the following constraints:

- That two complex numbers are *equal* or unequal.
- That three complex numbers are *colinear* line (cf. [8, §2.2, Command L]).
- That four complex numbers are *cocircular* (cf. [8, §2.2, Command C]).

4.5 Complex number evaluation under assignment

The floating point value of a complex number under an assignment is the complex number formed by the value of its real and imaginary number parts under the assignment.

4.6 Complex number graphical display

The POSTSCRIPT *graphical display string* for a complex number which has been evaluated to floating point is obtained with the `PS` procedure. `PS` takes a `DisplayStyle` parameter and an optional third argument, a label. `DisplayStyle` is a set which may be empty or include any consistent combination of the keywords:

```
AsRectangularPoint AsComplex AsPolarPoint
Label              Visible   Invisible
Hollow             Solid
```

The effect on the display of complex number $z = x + yi = r(\cos\theta + i\sin\theta$ depends on the presence of a given keyword:

- `Visible`: the point will be displayed.
- `Invisible`: the dot is not displayed but a label may be.
- `Solid`: The point will be marked by a solid dot (•).
- `Hollow`: The point will be marked by a hollow dot (o).
- `Label`: The label which is the third argument to `PS` will be printed next to the dot.
- `AsComplex`: the legend $x + yi$ will be printed next to the dot, preceded by an = if there is a label.
- `AsPolar`: the legend $(r, \angle\theta)$ will be printed next to the dot, preceded by an = if there is a label.
- `AsRectangular`: the legend (x, y) will be printed, preceded by an = if there is a label.
- `AsPS`: The real and imaginary parts are displayed separated by a blank.

Figure 3: Some ways of displaying points

Example 3 The following LT/GEOL code displays points in various ways.[2] P is given a hollow dot and polar coordinates. Q is given a solid dot and is displayed as a complex number. R is invisible but labelled. S is solid and given a label and rectangular point coordinates. The code for this is as follows:

```
read 'geol.m';

F1 := evalG(F({C(10, 50)  = {Hollow, AsPolar},
               C(10, 10)  = {Solid, AsC},
               C(270, 50) = ({Invisible}, 'Point R '),
               C(270, 10) = ({Solid, AsRectangular}, 'S ')}));

PSFile('excdisp.ps', 'gsave 100 0 translate', F1, grestore);

done
```

This creates the graphical display of Figure 3.

[2]This should be executed in a MAPLE started up with the quiet option (`maple -q`) to prevent garbage collection messages from messing up the POSTSCRIPT code.

Structure	
⟨x, y⟩	N × N
Constructions	
Right, Straight	A
+, −	A × A → A
Argument	C → A
Angle	C × C → A
Angle	C × C × C → A
Angle	S → A
Angle	L → A
DoubleAngle .	L × L → A
Constraints	
=	A × A → General equation set
Graphical display	
PS	A → string sequence
Evaluation	
FValue	A × Assignment → A

Table 4: The A structure

5 Angles

5.1 Angle structure

An angle is isomorphic to the equivalence class of all complex numbers, excluding 0, with the same argument. Only one representative of this equivalence class is necessary to represent the angle. Thus an angle is representable by a complex number-like structure under (for purposes of equivalence) modular arithmetic. Table 4 shows the constructions, constraints, and evaluation operations provided by the LT/GEOL package for the A structure.

5.2 Angle and angle-related constructions

Angles may be constructed from the following angle constants and constructions:

- The constant *right angle* (cf. [14, §3(k)]).
- The constant *straight angle* (cf. [14, §3(l)]).

- The *sum of two angles*, obtained as the complex product (cf. [14, §3(g)]).
- The *difference of two angles*, obtained as the complex quotient (cf. [14, §3(h)]).

An angle that may be obtained from a nonzero *complex number* $z = x+yi$ is it's *argument* $\arg z = \theta$ that is, the θ such that for some number r, $x + yi = r(\cos\theta + i\sin\theta)$.

Angles may be constructed from complex numbers as follows:

- The angle between two complex numbers A, B and the origin, $\angle B0A$, which is the counterclockwise motion from $\overline{0A}$ to $\overline{0B}$.
- The angle between three complex numbers A, B, C, $\angle CBA$, which is the counterclockwise motion from $\overline{AB}$ to $\overline{BC}$.

Let s be a line *segment* whose endpoints are $p_1 = x_1 + y_1 i$ and $p_2 = x_2 + y_2 i$. Define the *angle of* s *with respect to the origin* as follows. Assume that $x_1 \leq x_2$ (otherwise, switch the endpoints). If $y_2 \geq y_1$ then let $p_0 = x_2 + y_1 i$, and then the angle is $\angle p_0 p_1 p_2$. Otherwise let $p_0 = x_1 + y_2 i$, and then the angle is $\pi - \angle p_0 p_2 p_1$.[3]

An angle of a line *segment* with respect to the origin, is the angle $\angle p_0 p_1 p_2$, where $p_0 = \max(x_1, x_2) + \min(y_1, y_2)$. The angle of a line with defining complex numbers (u, w) is the angle of the line segment with endpoints $u + w, w$.

One may also construct the *double angle* (twice the value of the angle) between two *lines* (cf. [14, §3(f)]). Lines are discussed in §7.

5.3 Angle constraints

Equality is defined on angles.

Two angles are equal if the arguments of their complex number representations are equal. This is so if, when scaled down to the unit circle, they are equal complex numbers, i.e., if u and v are complex numbers representing angles α and β, then $\alpha = \beta$ when

$$\frac{u}{\sqrt{u\bar{u}}} = \frac{v}{\sqrt{v\bar{v}}}$$

[3] To be usable in a constraint computation, the above construction has to be reduced to a single expression which reduces to rational function expressions without embedded conditional operations.

that is, when

$$\frac{u^2}{u\bar{u}} = \frac{v^2}{v\bar{v}}$$

which is when

$$u\bar{v} = \bar{u}v$$

5.4 Angle evaluation under assignment

An angle may be converted to a PostScript angle floating point approximation with procedure `FValue` under an assignment.

5.5 Angle graphical display

PS gives the graphical display string sequence for an angle which has been previously evaluated to a floating point approximation. It takes a parameter `DisplayStyle`, one of:

- `Radians`: The radian value is displayed followed by the word `radians`.
- `Degrees`: The degree value is displayed followed by the word `degrees`.
- `AsPS`: The radian value alone is displayed.

Example 4 Here are the degree values of some important angles:

```
> [[1,0], [1,1], [0,1], [-1,1], [-1,0], [-1,-1], [0,-1], [1,-1]];
    [[1, 0], [1, 1], [0, 1], [-1, 1], [-1, 0], [-1, -1], [0, -1], [1, -1]]

> map(proc(x) evalG(Argument(C(op(x)))) end, ");
                        1          1
[A(N(1), N(0)), A(N(------), N(------)), A(N(0), N(1)),
                       1/2        1/2
                      2          2

          1          1                                  1           1
  A(N(- ------), N(------)), A(N(-1), N(0)), A(N(- ------), N(- ------)),
         1/2        1/2                                1/2         1/2
        2          2                                  2           2

                          1          1
  A(N(0), N(-1)), A(N(------), N(- ------))]
                         1/2        1/2
                        2          2

> map(proc(a) 'PS A'(a, Degrees) end, ");
```

```
[0 degrees, 45.00000002 degrees, 90.00000000 degrees, 135.0000000 degrees,
   180.0000000 degrees, 225.0000000 degrees, 270.0000000 degrees,
   314.9999999 degrees]
```

Structure	
⟨p1, p2⟩	C × C
Constructions	
S	C × C → S
Transformations	
Rotate	S × A → S
Translate	S × C → S
Scale	S × N → S
Constraints	
=, ≠, Parallel	S × S → General equation set
Perpendicular	S × S → General equation set
Evaluation	
FValue	S × Assignment → S
Graphical Display	
PS	S → string sequence

Table 5: The S structure

6 Segments

6.1 Segment structure

A line *segment* consists of a pair of endpoint complex numbers p1 and p2. Table 5 shows the constructions, constraints, transformations, evaluation and graphical display operations provided by the LT/GEOL package for the S structure.

6.2 Segment transformations

A segment may be rotated by an *angle*, translated by a *complex* number and scaled by a real *number*. A segment ℓ is rotated (respectively, translated, scaled) by forming the segment ℓ' with endpoints being the rotation (respectively translation, scaling) of the two endpoints of ℓ.

6.3 Segment constraints

Two segments may be constrained to be *parallel* or *pependicular*. This is obtained by extended the segments to lines (see §7) and obtaining the corresponding relations.

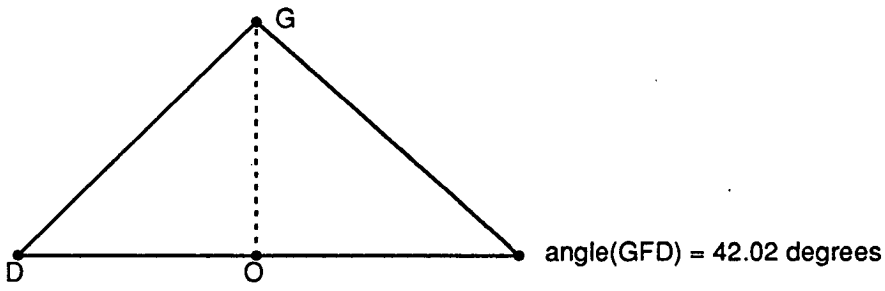

Figure 4: A triangle and its altitude

6.4 Segment evaluation under assignment

A segment's components may be approximated to floating point values with the procedure `FValue` under an assignment.

6.5 Segment graphical display

A POSTSCRIPT *graphical display* string is obtained for a line segment which has been evaluated to floating point with the `PS` procedure, which takes as parameters:

- `DisplayStyle`: One of the strings:

  `dashdot shortdash longdash solid dotted`

- `Label`: If non-null, a label to be displayed next to the segment.

Example 5 Figure 4 displays a triangle $\triangle DFG$ in `solid` lines with an altitude OG in a `shortdash` pattern. The angle $\alpha = \angle DFG = \angle OFG$ is also displayed. This display was produced with the following LT/GEOL code:

```
read 'geol.m':

G := C(100, 100):      FF := C(200, 10):
D := C(10, 10):        O  := C(100, 10):
alpha := evalG(Angle(G, FF, D)):

Digits := 4:

F1 :=
```

```
evalG
   (F({S(FF,G) = solid, S(D,G) = solid, S(D,FF) = solid, S(O,G) = shortdash,
       G = ({Visible, Label}, 'G '),
       FF = ({Visible, Label},
             cat(' angle(GFD) = ', 'PS A'(alpha, Degrees))),
       D = {Visible}, D - C(5,10) = ({Invisible, Label}, 'D '),
       O = {Visible}, O - C(5,10) = ({Invisible, Label}, 'O ')})):

'PS File'('segtest.ps', '', F1, ''):
```

done

Structure	
⟨u, w⟩	C × C
Constructions	
LThrough	C × C → L
LThrough	S → L
Parallel, Perpendicular	L × C → L
EquidistantFrom	C × C → L
Chord	Circle × Circle → L
TangentTo	Circle × C → L
Transformations	
Rotate	L × A → L
Translate	L × C → L
Scale	L × N → L
Constraints	
On	C × L → General equation set
Parallel, Perpendicular	L × L → General equation set
Evaluation	
FValue	L × Assignment → L
Graphical Display	
PS	L × string × string × C × C → string sequence

Table 6: The L structure

7 Lines

7.1 Line structure

Following Schwartz [14, §3], a *line* is isomorphic to a pair of complex numbers (u, w) such that there exist complex numbers p_1, p_2 which lie on the line and

$$\begin{aligned} (u, w) &= (p_1 - p_2, p_2) \\ u &\neq 0 \end{aligned}$$

Table 6 shows the constructions, constraints, transformations, evaluation and graphical display operations provided by the LT/GEOL package for the L structure.

7.2 Line constructions

A line may be constructed from *complex numbers* (see §4):

- From a pair of complex numbers p, q through which it passes, in which case the defining complex numbers are $p - q, q$ (cf. [14, §3(i)]). The constructor is LThrough.
- Equidistant to two complex numbers z_1 and z_2 (cf. [14, §3(r)]), in which case the defining complex numbers are $i(z_2 - z_1), \frac{1}{2}(z_1 + z_2)$.

A line may be constructed from a line *segment* (see §6), by forming the line through the segment's defining complex numbers.

A line may be constructed from other lines as follows:

- *Parallel* to another line with defining complex numbers u, w, passing through a complex number z, in which case the defining complex numbers are u, z (cf. [14, §3(i)]).
- *Perpendicular* to a line with defining complex numbers (u, w), passing through a complex number z (cf. [14, §3(j)]), in which case the defining complex numbers are (iu, z).

There are several lines which may be constructed from *circles* (see §8):

- The line tangent to a *circle* (R, z) (see 8) at a given complex number w. The defining complex numbers for this line are $i(z - w), w$ (cf. [14, §3(m)]).
- The line which is the *chord of intersection* of two circles (R_1, z_1) and (R_2, z_2) is the line with the pair of representing complex numbers [14, §3(o)]:

$$\begin{aligned} u &= i(z_2 - z_1) \\ w &= z_1 + (2(\overline{z}_2 - z_1))^{-1}(R_1^2 - R_2^2 + (z_2 - z_1)(\overline{z}_2 - z_1)) \end{aligned}$$

Example 6 The following LT/GEOL code constructs a line L1 which is tangent to a circle P1 with radius 50 and center $(100, 0)$, through the point $(150, 0)$.

```
read 'geol.m':

C1 := Circle(50, C(100, 0)):
```

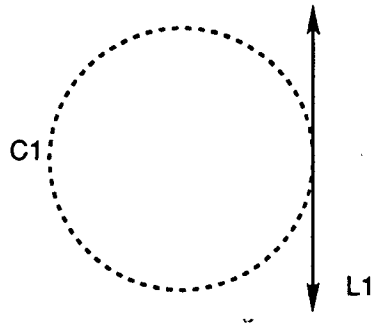

Figure 5: A line tangent to a circle

```
L1 := TangentTo(C1, C(150,0)):

TL := evalG(C(0,50)):
BR := evalG(C(300,-50)):
Digits := 4:

PSFile('line1test.ps', 'gsave 51 60 translate',
  F({C1 = ('C1 ', shortdash), L1 = ('L1 ', solid, TL, BR)}), grestore):
```

done

This results in the following POSTSCRIPT display of Figure 5.

7.3 Line transformations

A line may be rotated by an *angle*, translated by a *complex* number and scaled by a real *number*. Scaling is a "no-op" which is present for uniformity in the design of the system.

A line ℓ is rotated by forming the line ℓ' passing through the rotation of two points through ℓ.

A line ℓ is translated by forming the line ℓ' passing through the translation of two points through ℓ.

7.4 Line constraints

The complex number z lies *on* the line with representing complex number pairs (u, w) if $u\overline{(z-w)} = \bar{u}(z-w)$.

Two lines with representing complex number pairs (u, w) and (u', w') are

- *Parallel* if $u\bar{u}' = u'\bar{u}$ (cf. [14, §3(a)]).

- *Perpendicular* if $u\bar{u}' = -u'\bar{u}$ (cf. [14, §3(b)]).

7.5 Line evaluation under assignment

A line is evaluated to a floating point approximation by evaluating its defining complex numbers.

7.6 Line graphical display

A POSTSCRIPT *graphical display* string is obtained for a line which has been evaluated to floating point with the PS procedure, which takes as parameters:

- `Label`: If non-null, a label to be displayed next to the line.
- `DisplayStyle`: One of the strings:
  `dashdot shortdash longdash solid dotted`
- `TL, BR`: Lines are infinitely long, but are displayed finitely as a line segment with arrows at either end. Parameters `TL` ("top left') and `BR` ("bottom right") define a "window" on the plane to which the line will be clipped for display purposes. If the line is outside of this window, the null display string will be returned.

Example 7 The following LT/GEOL code defines two intersecting lines, `L1` and `L2`, and their point of intersection P.

```
read 'geol.m';

L1 := evalG(LThrough(C(-10,100), C(50,0)));
L2 := evalG(LThrough(C(0,20), C(200,200)));
P  := evalG(Intersection(L1, L2));

TL := evalG(C(-50,70));
BR := evalG(C(80,-50));
Digits := 3:

F1 := evalG(F({L1 = ('L1 ', solid, TL, BR), L2 = ('L2 ', solid, TL, BR),
               P = ({Label, AsRectangular}, 'P ')}));

PSFile('linetest.ps', 'gsave 150 50 translate', F1, grestore);

done
```

This results in the following POSTSCRIPT display of Figure 6.

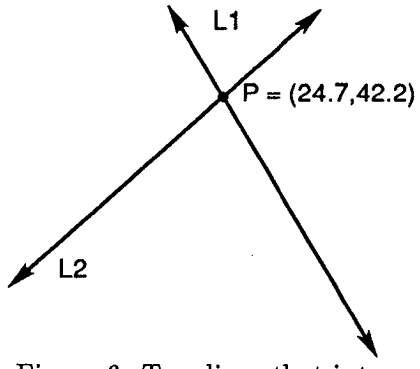

Figure 6: Two lines that intersect

Structure	
⟨R, z⟩	N × Circle
Transformations	
Rotate	Circle × A → Circle
Translate	Circle × C → Circle
Scale	Circle × N → Circle
Constraints	
On	C × Circle → General equation set
Evaluation	
FValue	Circle × Assignment → Circle
Graphical Display	
PS	Circle × string × string → string sequence

Table 7: The Circle structure

8 Circles

8.1 Circle structure

A *circle* is defined by a radius number R and a center point z. Table 7 shows the transformations, constraints, evaluation and graphical display operations provided by the LT/GEOL package for the Circle structure.

8.2 Circle transformations

A circle may be rotated, translated and scaled. A circle is translated (rotated) by translating (rotating) its center point. A circle is scaled by scaling

its radius.

8.3 Circle constraints

A point v lies on a circle (R, z) if $|v - z| = R$.

Example 8 The following LT/GEOL code starts with three points F, G, O at $(-50, 50)$, $(50, 50)$ and $(0, 0)$. The circle with radius x and center $y + zi$ passing through these points is computed by solving for the union Σ of the constraints that each of F, G, O lies on it. For the solution procedure we call on MAPLE's built-in `solve` routine. This yields two solutions, one assigning a negative radius value to x. The solution with a positive assignment to x is chosen. The circle is displayed in a `dotted` pattern. The code for this is:

```
read 'geol.m';

FF := C(-50,50);
G := C(50,50);
O := C(0,0);
C1 := Circle(x, C(y,z));
Digits := 4;

Sigma := evalG(On(FF, C1) union On(G, C1) union On(O, C1));

sigma := [solve(Sigma, {x,y,z})];
sigma := op(1, map(proc(s) if subs(s,x) > 0 then s fi end, sigma));

c1 := subs(sigma, C1);

F1 := evalG(F({FF = ({Visible, Label}, 'F '),
            G  = ({Visible, Label}, 'G '),
            O  = {Visible},
            O + C(-1,8) = ({Invisible, Label}, 'O '),
            c1 = ('C1 ', solid)}));

PSFile('circtest3.ps', 'gsave 150 0 translate', F1, grestore);

done
```

The trace of this procedure yields:

```
F := C(N(-50), N(50))

G := C(N(50), N(50))
```

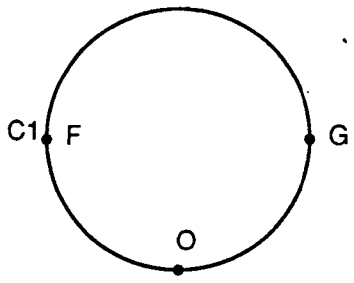

Figure 7: A circle through three points

```
O := C(N(0), N(0))

C1 := Circle(N(x), C(N(y), N(z)))

Sigma :=

                    2            2 1/2
  {(5000 + 100 y + y  - 100 z + z )    = x,

                    2            2 1/2        2    2 1/2
    (5000 - 100 y + y  - 100 z + z )    = x, (y  + z )    = x}

      sigma := [{z = 50, y = 0, x = 50}, {z = 50, y = 0, x = -50}]

                  sigma := {z = 50, y = 0, x = 50}

                 c1 := Circle(N(50), C(N(0), N(50)))
```

The procedure results in the PostScript display of Figure 7.

8.4 Circle evaluation under assignment

A circle is evaluated to floating point under an assignment by the procedure `FValue` by evaluating its radius and center to floating point under the assignment.

8.5 Circle graphical display .

A *graphical display* POSTSCRIPT string is produced for a circle which has been evaluated to floating point with the PS procedure, which takes the following parameters:

- Label: If non-null, place the given label next to the circle.
- DisplayStyle: One of the strings:

  dashdot shortdash longdash solid dotted

Example 9 The following LT/GEOL code draws a circle with center at (100,100) and radius 85. It draws a line through this circle which passes through the points (0,10) and (100,50). The two intersection points are computed, and are displayed with complex number labels. The circle is displayed in a dashdot pattern:

```
read 'geol.m';

C1 := Circle(85, C(100,100)):
L1 := evalG(LThrough(C(0,10), C(100,50))):
P  := evalG(Intersection(L1, C1)):

TL := evalG(C(0,110)):
BR := evalG(C(200,0)):
Digits := 4:

F1 := F({L1   = ('L1 ', solid, TL, BR),
         P[1] = {Visible, AsComplex},
         P[2] = {Visible, AsComplex},
         C1   = ('C1 ', dashdot)}):

PSFile('circtest1.ps', 'gsave 100 0 translate', F1, grestore);

done
```

This results in the POSTSCRIPT display of Figure 8.

Example 10 The following LT/GEOL code draws a circle with center at (50,0) and radius 50, and a circle with center at (100,−10) and radius 60. It computes the chord of intersection, and from that the two intersection points. The chord and the intersection points are drawn. The circles are displayed in a solid pattern, with the chord in a dashdot pattern.

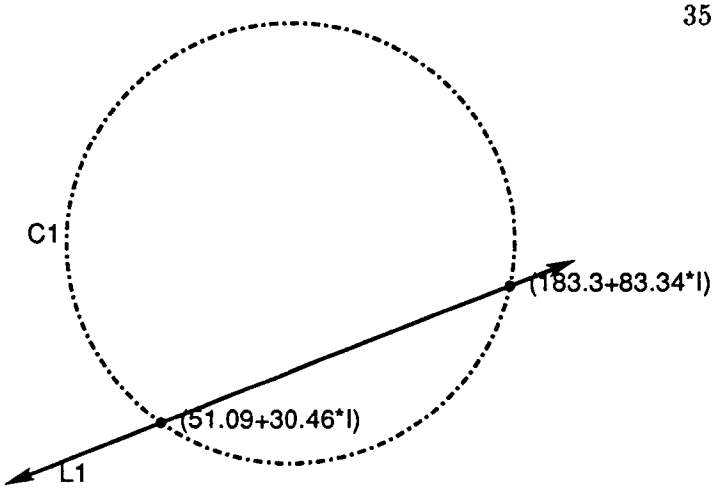

Figure 8: A circle and a line that intersects it

```
read 'geol.m';

C1 := Circle(50, C(50,0)):
C2 := Circle(60, C(100,-10)):
L1 := Chord(C1, C2):
P  := evalG(Intersection(L1, C1)):
TL := evalG(C(0,90)):
BR := evalG(C(300,-70)):
Digits := 4:

F1 := evalG(F({P[1] = {Visible}, P[2] = {Visible},
               C1 = ('C1 ', solid), C2 = ('C2 ', solid),
               L1 = ('L1 ', dashdot, TL, BR)})):

PSFile('circtest2.ps', 'gsave 100 70 translate', F1, grestore):

done
```

This results in the PostScript display of Figure 9.

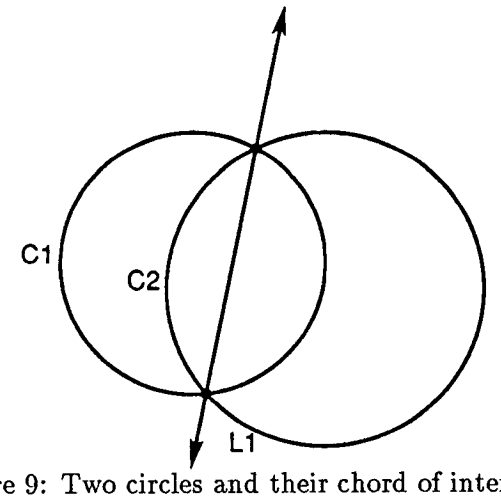

Figure 9: Two circles and their chord of intersection

9 Figures

9.1 Figure structure

A *figure* is a set of geometric structures. Figures are a way of organizing geometric constructions and also a way of simplifying access to the underlying code for various types of geometric structure. Table 8 shows the transformations, evaluation and graphical display operations provided by the LT/GEOL package for the F structure.

9.2 Figure transformations

A figure may be transformed by rotatation, translation and scaling. This is accomplished by carrying out the transformation on each of the structures in the structure set.

Example 11 A triangle like that if Figure 4 is constructed, with a circle below it. These are rotated π radians. The result is shown in Figure 10. The LT/GEOL code for this is:

```
read 'geol.m';

D := C(0, 0): O := C(100, 0): FF := C(200, 0): G := C(100, 100):
C1 := Circle(50, C(100, -50)):
alpha := Angle(G, FF, D):
d := S(FF,G): f := S(D,G): g := S(D,FF): o := S(O,G):
```

Structure	
⟨cònstructions⟩	Structure set
Transformations	
Rotate	F × A → F
Translate	F × C → F
Scale	F × N → F
Evaluation	
FValue	F × Assignment → F
Graphical Display	
PS	F → string sequence
PSFile	string × string × F × string → ()
PSOpenFile	string → ()
PSWrite	(string + F)* → ()
PSCloseFile	() → ()

Table 8: The F structure

```
F1 := evalG(F({D = {Visible}, O = {Visible}, d = solid, f = solid, g = solid,
               o = shortdash, G = {Visible}, C1 = (``, longdash),
               FF = {Visible}})):

PSOpenFile(`figtest.ps`):
PSWrite(`gsave 200 100 translate 0.8 0.8 scale`, F1,
        evalG(Rotate(F1, Straight)), grestore):
PSCloseFile():
```

done

9.3 Figure evaluation under assignment

A the components of a figure may be evaluated to a floating point approximation. This is accomplished by carrying out the evaluation on each of the structures in the structure set.

9.4 Figure graphical display

A figure which has been evaluated to floating point is displayed by displaying each of the components in the display set, together with their arguments,

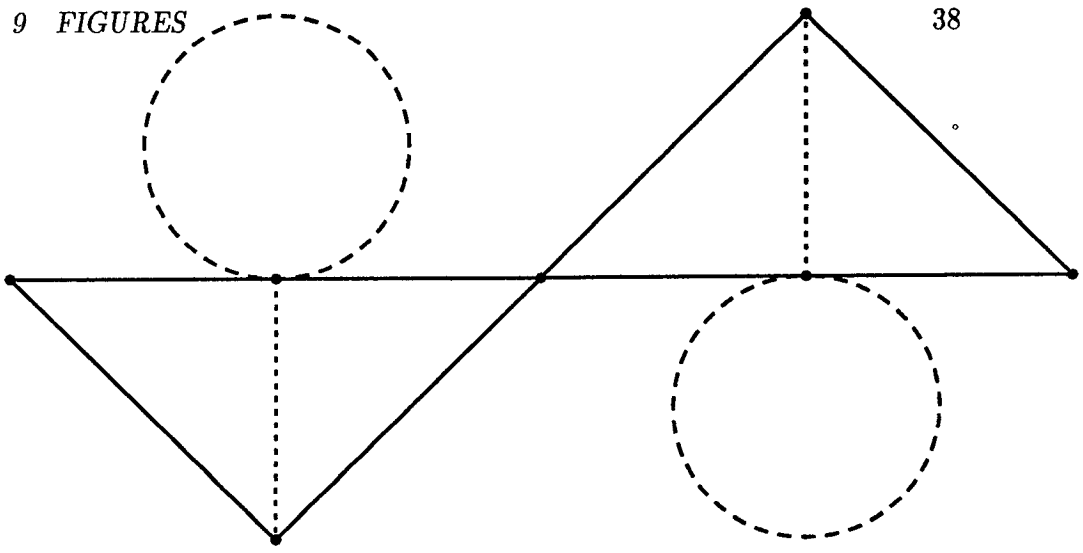

Figure 10: A triangle and a circle, rotated

with PS.

The procedure `PSFile(filename, PSpre, fig, PSpost)` will output a figure `fig` in POSTSCRIPT format to a file with name `filename`, prepending POSTSCRIPT code `PSPre` and appending POSTSCRIPT code `PSPost`.

The procedure `PSOpenFile(filename)` opens output to `filename` and selects a Helvetica font.

The procedure `PSWrite` with an arbitrary number of arguments writes the argument to the open file. If the argument is a string, it is written. If it is a figure, the POSTSCRIPT form is computed and then written.

The procedure `PSCloseFile()` closes the output file and opens the terminal.

10 Multivariate polynomial set normal form

Common applications, such as the computation of Gröbner bases [4] or Wu's geometric theorem proving method [8], assume as input a set of polynomials implicitly equated to zero, where the polynomials are in the ring $Q(\bar{u})[\bar{x}]$, where $\bar{u}$ are *universal indeterminates*, that is, parameters or free or independent variables, and the $\bar{x}$ are *existential indeterminates*, that is, dependent or bound variables. Call this the *multivariate polynomial equation set normal form.*

But the constraint sets resulting from the constraint and construction procedures defined in prior sections are sets of *general equations* (see §1.3) on *rational functions*, that is, expressions built up from polynomials and the division operation.

One can imagine two strategies for solving a system of polynomial equations:

- *Translate all:* Translate the non-equations into equations, translate all the equations polynomials implicitly equated to zero, then solve them.
- *Translate equalities:* Separate out the non-equations, translate the equations into polynomials implicitly equated to zero, solve for the equations, and then discard solutions which do not satisfy the non-equations.

Hence LT/GEOL includes the following procedures:

- `TranslateAll(`*constraints*`)`, which returns a *table* with the following fields:
  - `Polynomials`, a set of polynomials implicitly equated to zero.
  - `Vars`, a set of the indeterminates originally in the *constraints* set.
  - `NewVars`, a set of new indeterminates introduced in translating non-equations into equalities. (The method of translation is discussed in §A.9.2.)
- `TranslateEqns(`*constraints*`)`, which returns a *table* with the following fields:
  - `Nonequations`, a list of the non-equations in *constraints.*
  - `Polynomials`, a list of the polynomials implicitly equated to zero which are the translation of the equations in *constraints.*

- `Vars`, a set of the indeterminates in the equations prior to translation to `Polynomials`.
- `NewVars`, a set of the indeterminates added to the equations in the process of creating the `Polynomials`.
- `NVars`, a set of the indeterminates in `Nonequations`.

- `Filter`(Σ, I), where σ is a set of assignments and I is a general equation set, returns the set of those assignments in Σ which satisfy I. "Satisfy" here means that the general equation set under substitution reduces to the set containing the single element 0.

11 Hierarchical multidimensional systems

11.1 Dependency assertions and hierarchical systems

Let Σ be a set of polynomials and V the set of names of real-valued indeterminates occurring in Σ. Let U and W be disjoint sequences of elements of V. *U depends on W* iff there exists a function f such that, for any assignment of real values to elements of W, under the assignment $U = f(W)$, Σ is solvable. Under these conditions, W is *independent with respect to U*. If U depends on no other set in V, then the *elements* of U are *mutually dependent*, and the *set U* is *independent*.

Let G be a directed acyclic graph in which:

- The nodes N_i are pairs $\langle U_i, E_i \rangle$ such that $U_i \subseteq V$ and $E_i \subseteq \Sigma$.
- If $N_i, N_j, i \neq j$ are nodes of G, then $U_i \cap U_j = \emptyset$ and $E_i \cap E_j = \emptyset$.
- $V = \cup U_i$ and $\Sigma = \cup E_i$.

Such a graph may be used to represent a sequence of *dependency assertions* of three types:

- *Initialize:* Σ is a system of equations with mutually independent indeterminates V.
- *Independent:* $U \subseteq V$ is independent in Σ.
- *Depends on:* U depends on W in Σ, where $U \cap W = \emptyset$ and $U, W \subseteq V$.

In such a *dependency graph* G, if $N = \langle U, E \rangle$ is a node, then the indeterminates in E are included in the union of U and the indeterminates of nodes which are reachable from N, and each $f \in E$ has at least one indeterminate in U. Other appropriate terms for G are *hierarchical multidimensional system* or simply *hierarchical system.*

11.2 The HMS structure for manipulating dependencies

The LT/GEOL package provides the HMS structure for maintaining dependency graphs formed initially from polynomial sets Σ and subsequently from dependency assertions on dependency graphs. The assertions operate on structures or structure sets; the indeterminates involved are computed from the structures or structure sets via the Indets operation. Table 9 displays

Structure	
⟨graph⟩: name set → name set set × polynomial set	
Constructions	
Initialize	polynomial set → HMS
IndependentWith	structure set × HMS → HMS
IndependentSep	structure set × HMS → HMS
Dependent	structure set × structure set × HMS → HMS
Solution	
Trim	HMS → HMS-like structure
Leaves	HMS → (name set × polynomial set) set
Substitute	assignment × HMS → HMS
Solve	HMS → assignment sequence

Table 9: The HMS structure

the HMS structure. The operations in the HMS structure are described in the following paragraphs.

Initialize(Σ) assumes that all the variables in Σ are all mutually dependent. It computes the set of real-valued indeterminates V of Σ, and returns the HMS with the single leaf node $\langle V, \Sigma \rangle$.

IndependentWith(V, H), where H is an HMS and V is a set of names occurring in equations in H, declares those names to be mutually dependent indeterminates which are not dependent on any other indeterminates. A new HMS is constructed from H and V as follows:

1. The elements of V are deleted from the keys of the nodes $\overline{N}$ of H in which they occur.

2. A new leaf node $N' = \langle V, E \rangle$ is adjoined, where E are those equations occurring in $\overline{N}$ which contain only elements of V. The elements of E are deleted from the equation sets of $\overline{N}$.

3. An arc is adjoined from each node of $\overline{N}$ to N'.

IndependentSep(V, H), where H is an HMS and V is a set of names occurring in equations in H, declares each name to be independent of all other indeterminates. It simply calls IndependentWith iteratively for each element of V.

Dependent(U, W, H), where U, W are disjoint subsets of the indeterminates of the HMS H, declares U depends on W in the equations of H. A new

HMS is constructed from H and U, W as follows:

1. G is set to a copy of H, and let $\overline{N}_U$ and $\overline{N}_W$ be empty sets of nodes.
2. For each $g = \langle V_g, E_g \rangle \in G$,
   (a) If U intersects V_g, then
      i. Let G be $G \setminus \{g\}$.
      ii. If W intersects V_g, then
         A. Split g into two nodes

$$g_U = \langle V_g \setminus W, E_{g,U} \rangle$$
$$g_W = \langle W, E_{g,W} \rangle$$

            where the $E_{g,W}$ are those equations in E whose indeterminates are contained in W, and the $E_{g,U}$ are the remaining equations in E.
         B. Let g_W have all the arcs of g, and let there be a single arc from g_U to g_W.
         C. Put g_U in $\bar{N}_U$ and g_W in $\bar{N}_W$.
      iii. Otherwise, put g in $\bar{N}_U$.
   (b) Otherwise, if W intersects V_g then put g in $\bar{N}_W$ and let G be $G \setminus \{g\}$.
3. The following two nodes are formed:

$$N_U = \langle \cup V_{N_{U,i}}, \cup E_{N_{U,i}} \rangle$$
$$N_W = \langle \cup V_{N_{W,i}}, \cup E_{N_{W,i}} \rangle$$

4. If $\cup V_{N_{U,i}}$ does not intersect $\cup V_{N_{W,i}}$, then:
   (a) The two nodes N_U, N_W are added to G.
   (b) All arcs from nodes remaining in G to nodes $\overline{N}_U$ are redirected to N_U.
   (c) N_U has an arc to N_W and arcs to every node remaining in G which some node of $\overline{N}_U$ has an arc to.
   (d) All arcs from nodes remaining in G to nodes $\overline{N}_W$ are redirected to N_W.

(e) N_W has an arc to every node which some node of $\overline{N}_W$ has an arc to.

Otherwise:

(a) The node $N_{UW} = \langle V_{N_U} \cup V_{N_W}, E_{N_U} \cup E_{N_W} \rangle$ is added to G.

(b) All arcs from nodes remaining in G to nodes $\overline{N}_U$ are redirected to N_{UW}.

(c) N_{UW} has an arc to every node remaining in G which some node of $\overline{N}_U$ has an arc to.

(d) All arcs from nodes remaining in G to nodes in $\overline{N}_W$ are redirected to N_{UW}.

(e) N_{UW} has an arc to every node remaining in G to which a node of $\overline{N}_W$ has an arc.

5. G is returned.

`Trim`(H) returns a version H' of H in which all of the leaves of H, and arcs leading to those leaves, have been deleted. If H consists solely of leaves, then it returns `NULL`. The indeterminates of the deleted leaves are not added to the indeterminate sets of the leaves of H', so the result has the structure but not all the properties of an `HMS`. That is, `Trim` is intended for computing with an `HMS` to which no further dependency assertions will be applied.

`Leaves`(H) returns a set of lists [`Key`, `Eqns`] where `Key` is a name set (the node label) and `Eqns` is a polynomial set. This set comprises the leaf nodes of H, that is, those nodes with no outgoing arcs.

`Substitute`(σ, H) returns the `HMS` in which every equation set E in a node is replaced by `subst`(σ, E).

11.3 The hierarchical system solution algorithm

If the dependency assertions of H are valid, then Σ may be solved, where the solution is a function of the indeterminates of the leaf nodes of H which gives an assignment to the remaining indeterminates in the system, by the following algorithm `Solve`(H, p), where:

- H is an `HMS`.
- $p(E, V)$ is a procedure which takes a set of polynomials E and a set of names V and returns a sequence of assignments, that is, a MAPLE

sequence of sets of equations of names to real-valued expressions. (An example of such a procedure is the MAPLE `solve` procedure.)

`Solve`(H,p) works as follows:

1. Let $\overline{N}$ = `Leaves`$(H) = N_1, N_2, \ldots N_k$.
2. For each $N_i = \langle V_i, E_i \rangle$ let $\overline{\sigma}_i = p(E_i, V_i)$ be a sequence of assignments.
3. Let
$$\Phi = \{\sigma_{1,j_1} \circ \sigma_{2,j_2} \circ \cdots \circ \sigma_{k,j_k}\}$$
be the set of the compositions of the set of all possible combinations of solutions of the $\overline{N}$. Note that since the V_i are disjoint, composition of assignments is associative, hence the compositions of all permutations of such a combination of assignments are identical. Under these conditions, in MAPLE, composition of assignments is just set union.
4. Let H = `Trim`(H).
5. If H is `NULL`, then return Φ.
6. Consider $R = (H_1, \phi_1), (H_2, \phi_2), \ldots (H_k, \phi_k)$ where H_i is the `HMS` resulting from substituting a composite leaf solution ϕ_i from Φ into H and recursively solving.
7. Form the sequence $H'_{i,j}$ which is the union of ϕ_i with each solution j of H_i.
8. Return the set of the $H'_{i,j}$ for all i, j.

11.4 An example of solving a hierarchical system

The problem is construct a widget which consists of a lever and a flexible triangular structure. The lever consists of a line segment through points P, O, Q, such that O is fixed, and P and Q are equidistant from O. R and S are two fixed points. R is the midpoint of the line segment $\overline{TU}$, Q is the midpoint of the line segment $\overline{TV}$, and S is the midpoint of the line segment $\overline{UV}$. The points and line segments involved are constructed as follows:

```
# Define the points
O := C(0,0): P := C(a,b): Q := C(c,d):
```

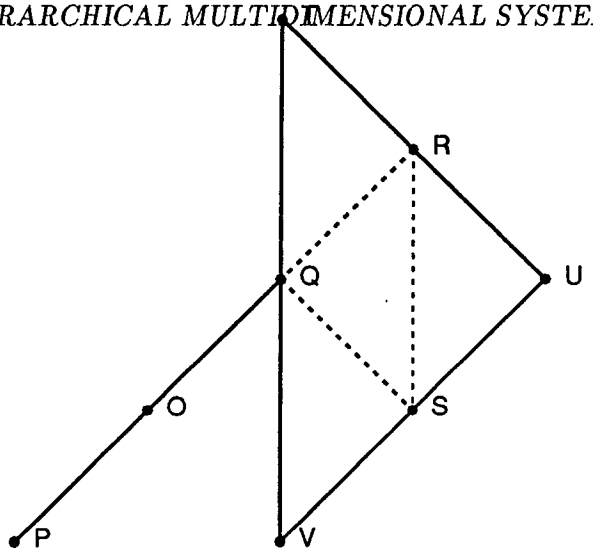

Figure 11: A particular solution for a widget

```
R := C(e,f): SS := C(g,h): T := C(i,j):
U := C(k,l): V := C(m,n):
```

Then their constraints are declared and translated into a set of polynomials, separating out for the moment the inequality constraints:

 # Define the constraints

```
G := evalG((Q = 2*O - P) union (R = MidPoint(T,U)) union
           (SS = MidPoint(U,V)) union (Q = MidPoint(T,V))):

Widget := TranslateEqns(G):
```

A hierarchical system is created with the polynomials obtained from the constraints. R, S, T, U, V are declared dependent on O, P, Q. Q is declared dependent on O, P. R and S are declared independent. The resulting hierarchical system is depicted in Figure 12). The LT/GEOL dependency declarations are as follows:

 # Create the hierarchical system

```
G1 := Initialize     (Widget[Polynomials]):
G2 := Dependent      ({R,SS,T,U,V}, {O,P,Q}, G1):
G3 := Dependent      (Q, {O, P}, G2):
G4 := IndependentSep ({R, SS}, G3):
```

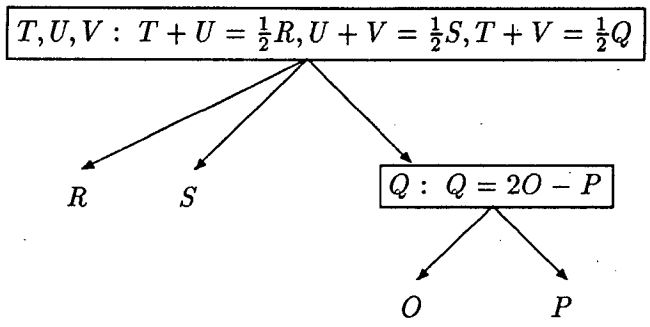

Figure 12: A dependency graph for a widget

To solve the system for P, we must first supply some constant values for R and S. Then we solve for Z in terms of P. Then we solve for T, U, V. Using `Solve`, which calls MAPLE `solve` and then deletes equations of the form $x = x$ from the resulting assignment, `G4` is solved for the indeterminates in its leaves with the command:

```
# Solve the hierarchical system

Sigma := 'Solve HMS'(G4, Solve):
```

This results in a single solution:

```
[{c = -a, d = -b, m = - 1/4 e + 1/4 g - 1/4 a, l = 1/4 b + 1/4 h + 1/4 f,
  k = 1/4 e + 1/4 g + 1/4 a, n = - 1/4 b + 1/4 h - 1/4 f,
  j = - 1/4 b - 1/4 h + 1/4 f, i = 1/4 e - 1/4 g - 1/4 a}]
```

Supplying some values for the independent variables, the following code computes the display of Figure 11:

```
# Create a figure from the solved system and display a particular solution

F1 := F({S(P, Q) = solid, S(T, V) = solid,
         S(V, U) = solid, S(T, U) = solid,
         S(R,Q) = shortdash, S(Q,SS) = shortdash,
         S(R,SS) = shortdash,
         O = ({Visible}, 'O '),
         P = ({Visible}, 'P '),
         Q = ({Visible}, 'Q '),
         R = ({Visible}, 'R '),
         SS = ({Visible}, 'S '),
         T = ({Visible}, 'T '),
```

```
        U = ({Visible}, 'U '),
        V = ({Visible}, 'V ')}):

sigmaIndeps := evalG((P  = C(-50,-50)) union
                     (R  = C(100,100)) union
                     (SS = C(100,0))):

sigma := subs(sigmaIndeps, Sigma[1]):
F2 := subs(sigma union sigmaIndeps, F1):
PSFile('deptest.ps', 'gsave 100 50 translate', F2, grestore):
```

12 Geometric expression evaluation

This section describe `evalG`(e, σ), which evaluates expression e under an assignment of structure types to indeterminates σ. e is any MAPLE expression, which is interpreted as follows:

- A numeric constant k becomes `N`(k).
- If $v =$ `N` is in set σ, then occurrences of indeterminate v in e become `N`(v).
- Any indeterminate which does not have a structure type assignment in σ is assumed to be an `N`.
- `evalG`$(\{s_1, s_2, \ldots s_k\}, \sigma)$ is $\{$`evalG`$(s_1, \sigma), \ldots$ `evalG`$(s_k, \sigma)\}$.
- `evalG` of the expression `N`(s) returns `N`(s).
- `evalG` of the expression `F`(s) where s is a set with elements of the form $a = b$, returns `F`(s') where the elements of s' are of the form `evalG`$(a, \sigma) = b$.
- `evalG`$(f(s), \sigma)$, where f is a structure type name other than `F` or `N`, results in $f(op(map($`evalG`$, s, \sigma)))$.
- If a procedure name in a procedure call is bound to a procedure, it is called on the recursively evaluated arguments. Otherwise, interpretations of function names and infix operators depend on the type of the recursively evaluated arguments. Thus, if in `a*b`, `a` is an `N` and `b` is a `C`, then the result from `evalG` will be the evaluation of `'* N C'(a,b)`.

σ is optional and may be ommitted. If it is, then every indeterminate in an non-function position is assumed to be an `N` structure.

Example 12 Here are some geometric expression evaluations illustrating the rules discussed above:

```
> evalG(1);
                                N(1)
> evalG(a);
                                N(a)
> evalG(1/a);
```

```
                                        N(1/a)
> evalG(C(1,0));
                                    C(N(1), N(0))
> evalG(C(a,1));
                                    C(N(a), N(1))
> evalG(L(a,b), {a = C, b = C});

L(C(N((v &of (Re &of a)) - (v &of (Re &of b))),
   N((v &of (Im &of a)) - (v &of (Im &of b)))),
   b)

> evalG(a*C(0,1) + C(1,0)/b);
                                   C(N(1/b), N(a))
> evalG(C(a,b) = C(d,e));
                                    {a = d, b = e}
```

A Implementation

A.1 Structures

A *structure* g = ⟨a, b, c⟩ is represented by the following convention:

- All software which uses structures of *type* g is presumed to "know" that g has components a, b and c.
- No user software which uses the LT/GEOL package will bind a value to the global names g, a, b or c or `'g Fields'`
- The global name `'g Fields'` is assigned to the table with field-value pairs {(a, 1), (b, 2), (c, 3)}.
- The structure-accessing operation `&of` is defined as follows:

```
'&of' :=
  proc (field, x)
  local structtype, structfields, fieldid;
  option remember;
    if not type(x,function)
    then 'field &of x'
    else structtype := op(0,x);
         structfields := ''.structtype.' Fields';
         if not type(structfields, table)
         then 'field &of x'
         else fieldid := structfields[field];
              if type(fieldid, function)
              then ERROR(''.field.' is not a field of '.structtype)
              else op(fieldid, x)
              fi
         fi
    fi
  end:
```

- An *instance* of type g with values a, b, c for structure components a, b and c will be represented as the MAPLE procedure call g(a, b, c).
- g is defined as a type to MAPLE by the global name binding:

```
'type/g' := proc (x) op(0,x) = g end;
```

The above declarations are performed by the ***structure*** procedure, defined as follows:

```
structure :=
  proc (name, fields)
  local i;
    ``.`type/`.name := subs(N = name, op(TypeCheck));
    ``.name.` Fields` := table({('fields[i]' = i) $ i = 1..nops(fields)})
  end;

TypeCheck :=
 proc (x) if type(x, function) then evalb(op(0,x) = N) else false fi end;
```

A.2 Numbers

A.2.1 Number structure

```
structure(N, [v]);
```

A.2.2 Number constructions

```
'+ N N' :=
  proc(a, b)
    if v &of a = Infinity
    then  if (v &of b) &in {Infinity, Bottom} then N(Bottom)
          else N(Infinity) fi
    elif v &of a = Bottom then N(Bottom)
    elif v &of b = Infinity then N(Infinity)
    elif v &of b = Bottom then N(Bottom)
    else N(v &of a + v &of b) fi
  end:

'- N' :=
  proc(a)
    if (v &of a) &in {Infinity, Bottom} then a else N(- v &of a) fi
  end:

'- N N' := proc(a, b) '+ N N'(a, '- N'(b)) end:

'^ N N' :=
  proc (a, b)
    if v &of b = -1 then 'Inverse N'(a)
    else N((v &of a)^(v &of b)) fi  # Not complete
  end:

'* N N' :=
  proc(a, b)
    if v &of a = 0
    then if (v &of b) &in {Infinity, Bottom}
         then N(Bottom) else N(0) fi
    elif v &of a = Infinity
    then if v &of b = 0 or v &of b = Bottom then N(Bottom)
         else N(Infinity) fi
    elif v &of a = Bottom then a
    elif v &of b = Infinity then b
    elif v &of b = Bottom then b
    else N(v &of a * v &of b) fi
  end:
```

```
'Inverse N' :=
  proc (a)
    if v &of a = Infinity then N(0)
    elif v &of a = Bottom then a
    elif v &of a = 0 then N(Infinity)
    else N(1/v &of a) fi
  end:

'/ N N' := proc(a, b) '* N N'(a, 'Inverse N'(b)) end:

'Modulus C' :=
  proc (c)
  local re, im;
    re := Re &of c; im := Im &of c;
    'Sqrt N'('+ N N'('* N N'(re, re), '* N N'(im, im)))
  end:

'Sqrt N' :=
  proc (n)
    v &of n;
    if " &in {Infinity, Bottom} then n else N(sqrt(")) fi
  end:

'Distance C C'  := proc (p1, p2) 'Modulus C'('- C C'(p1, p2)) end:
'XDistance C C' := proc (p1, p2) '- N N'(Re &of p2, Re &of p1) end:
'YDistance C C' := proc (p1, p2) '- N N'(Im &of p2, Im &of p1) end:

'Slope C C' :=
  proc (p1, p2)
    '/ N N'('YDistance C C'(p1, p2), 'XDistance C C'(p1, p2));
  end:

'Sine A'    := proc (A) C(op(A)); '/ N N'(Im &of ", 'Modulus C'(")) end:
'Cosine A'  := proc (A) C(op(A)); '/ N N'(Re &of ", 'Modulus C'(")) end:
'Tangent A' := proc (A) '/ N N'(y &of A, x &of A) end:

'Distance C L' :=
  proc (p, line) 'Distance C C'(p, 'Projection C L'(p, line)) end:

'Distance S' := proc (s) 'Distance C C'(p1 &of s, p2 &of s) end:
```

A.2.3 Number constraints

```
'= N N'  := proc(a, b) {v &of a = v &of b} end;
'< N N'  := proc(a, b) {v &of a < v &of b} end;
'> N N'  := proc(a, b) {v &of a > v &of b} end;
```

```
'<> N N' := proc(a, b) {v &of a <> v &of b} end;
'<= N N' := proc(a, b) {v &of a <= v &of b} end;
'>= N N' := proc(a, b) {v &of a >= v &of b} end;
```

A.2.4 Number evaluation

```
'FValue N' := proc (n, sigma) evalf(simplify(subs(sigma, v &of n))) end;
```

A.2.5 Number graphical display

```
'PS N' :=
  proc (n)
  local mant, exp, s, f;
    f := n;
    if type(f,integer) then RETURN(convert(f, name))
    elif type(f,N) then RETURN('PS N'(op(f)))
    elif not type(f, float) then f := convert(f, float) fi;
    mant := convert(op(1,f), name); exp := -op(2,f);
    s := length(mant);
    if s = exp then '.'.mant
    else cat(substring(mant, 1..(s-exp)),'.',substring(mant,s-exp+1..s)) fi
  end;
```

A.3 Complex numbers

A.3.1 Complex number structure

```
structure(C, [Re,Im]);
```

A.3.2 Complex number constructions

```
'C N N' := C;

'- C' := proc (p) C('- N'(Re &of p), '- N'(Im &of p)) end:
'Conjugate C' := proc (p) C(Re &of p, '- N'(Im &of p)) end:

'+ C C' :=
  proc (p1, p2)
    C('+ N N'(Re &of p1, Re &of p2), '+ N N'(Im &of p1, Im &of p2))
  end:

'- C C' :=
  proc (p1, p2)
    C('- N N'(Re &of p1, Re &of p2), '- N N'(Im &of p1, Im &of p2))
  end:

'* N C' := proc (n, p) C('* N N'(n, Re &of p), '* N N'(n, Im &of p)) end:
'* C N' := proc (p, n) C('* N N'(n, Re &of p), '* N N'(n, Im &of p)) end:

'/ C N' :=
  proc (p, n) C('/ N N'(Re &of p,n), '/ N N'(Im &of p,n)) end:

'* C C' :=
  proc (p1, p2)
  local a, b, c, d;
    a := Re &of p1; b := Im &of p1;
    c := Re &of p2; d := Im &of p2;
    C('- N N'('* N N'(a,c), '* N N'(b,d)),
            '+ N N'('* N N'(a,d), '* N N'(b,c)))
  end:

'Sqrt C' :=
  proc (c)
    evalc(sqrt(v &of (Re &of c) + (v &of (Im &of c))*I));
    C(N(evalc(Re(%))), N(evalc(Im(%))))
  end:

'Inverse C' :=
  proc (c)
```

```
    local a, b, r;
      a := Re &of c; b := Im &of c;
      r := '+ N N'('* N N'(a,a), '* N N'(b,b));
      C('/ N N'(a, r), '/ N N'('- N'(b), r))
  end:

'/ C C' := proc (p1, p2) evalG(p1 * Inverse(p2)) end:

'MidPoint C C' := proc(p1, p2) evalG((1/2)*(p1 + p2)) end:

'C A' := proc (a) C(op(a)) end:

'Intersection L L' :=
  proc (l1, l2)
  local U, W, U1, W1, cU1,cW1, cU, cW;
    U := u &of l1; W := w &of l1;
    U1 := u &of l2; W1 := w &of l2;
    cU := 'Conjugate C'(U); cW := 'Conjugate C'(W);
    cU1 := 'Conjugate C'(U1); cW1 := 'Conjugate C'(W1);
    'Inverse C'('- C C'('* C C'(U, cU1), '* C C'(cU, U1)));
    '* C C'('- C C'('* C C'('- C C'('* C C'(W1, cU1), '* C C'(cW1, U1)), U),
                    '* C C'('- C C'('* C C'(W, cU), '* C C'(cW, U)), U1)), ")
  end:

'Intersection L Circle' :=
  proc (l, cc)
  local U, W, r, Z, 'w - z', cWZ, cU, pm, inner;
    U := u &of l; W := w &of l;
    r := R &of cc; Z := z &of cc;
    'w - z' := '- C C'(W, Z);
    cWZ := 'Conjugate C'('w - z');
    cU := 'Conjugate C'(U);
    '* C C'('w - z', cU);
    '* C C'(U, cWZ);
    inner := '- C C'("", ");
    '* C C'(", ");
    '* N C'('* N N'(N(4), '* N N'(r, r)), '* C C'(U, cU));
    '+ C C'("", ");
    pm := 'Sqrt C'(");
    '* N C'(N(2), cU);
    'Inverse C'(");
    '* C C'(", '+ C C'(inner, pm));
    '* C C'("", '- C C'(inner, pm));
    ['+ C C'(Z, "), '+ C C'(Z, "")]
  end:

'Intersection Circle Circle' :=
```

```
  proc (c1, c2)
    'Intersection L Circle'('Chord Circle Circle'(c1, c2), c1)
  end:

'On L N' := proc(l, n) '+ C C'('* N C'(n, u &of l), w &of l) end:

'Projection C L' :=
  proc(p, l) 'Intersection L L' (LPerpendicularTo(l, p), l) end:

'Reflection C L' :=
  proc(p, l) '- C C'('* N C'(N(2),'Projection C L'(p,l)), p) end:

'On Circle N' := proc (c, n) '+ C C'('* N C'(n, R &of c), z &of c) end:
```

A.3.3 Complex number transformations

```
'Rotate C' :=
  proc (c, a)
  local r, theta, phi;
    r := 'Modulus C'(c);
    if v &of r = 0 then c
    else theta := 'Argument C'(c);
         phi := '+ A A'(a, theta);
         '* N C'(r, C(op(phi))) fi
  end;

'Scale C' := proc (c, n) '* N C' (n, c) end;

'Translate C' := '+ C C';
```

A.3.4 Complex number constraints

```
'= C C' :=
  proc(pa, pb)
    '= N N'(Re &of pa, Re &of pb) union '= N N'(Im &of pa, Im &of pb)
  end:

'<> C C' :=
  proc(pa, pb)
    '<> N N'(Re &of pa, Re &of pb) union '<> N N'(Im &of pa, Im &of pb)
  end:

'CoCircular C C C C' :=
  proc(p1, p2, p3, p4)
    '= N N'('Tangent A'('A C C C'(p1, p2, p3)),
```

```
            'Tangent A'('A C C C'(p1, p4, p3)))
  end:

'CoLinear C C C' :=
  proc(p1, p2, p3)
    '= N N'('Slope C C'(p1, p2), 'Slope C C'(p2, p3))
  end:
```

A.3.5 Complex number evaluation

```
'FValue C' :=
  proc (c, sigma)
    C('FValue N'(Re &of c, sigma), 'FValue N'(Im &of c, sigma))
  end;
```

A.3.6 Complex graphical display

```
'PS C' :=
  proc (c, DisplayStyle)
  local label, Label, dot;
    if nargs = 3 then Label := args[3] else Label := NULL fi;
    if AsComplex &in DisplayStyle
    then label := 'As Complex'(c)
    elif AsRectangular &in DisplayStyle
    then label :=  'As Rectangular'(c)
    elif AsPolar &in DisplayStyle
    then label :=  'As Polar'(c)
    elif AsPS &in DisplayStyle
    then RETURN('As PS'(c))
    else label := NULL fi;
    dot := Dot (2, c, DisplayStyle);
    if Label <> NULL
    then if label <> NULL then label := cat(Label, '= ', label)
         else label := Label fi
    elif label = NULL then label := '' fi;
    if Invisible &in DisplayStyle
    then gsave, 'PS String'(label, c), grestore
    else gsave, dot, 'PS String'(label, '+ C C'(c, C(N(7), N(-2)))),
         grestore fi
  end;

'As Complex' :=
  proc (c)
    if Im &of c = 0 then 'PS N'(Re &of c)
    elif Re &of c = 0
```

```
    then 'PS N'(Im &of c), '*I'
    else 'PS N'(Re &of c), '+', 'PS N'(Im &of c), '*I' fi;
    cat('(', '', ')')
  end;

'As Rectangular' :=
  proc (c) cat('(','PS N'(Re &of c),',','PS N'(Im &of c),')') end;

'As Polar' :=
  proc (c)
  local x, y, r, theta;
    cat('PS N'('Modulus C'(c)), ' at ', 'PS A'('Argument C'(c), Radians))
  end;

'As PS' := proc (c) cat('PS N'(Re &of c),' ','PS N'(Im &of c)) end;

'PS String' :=
  proc(text, P)
  local start;
    start := cat('newpath ', 'PS N'(Re &of P), ' ',
                 'PS N'(Im &of P), ' moveto',' ');
    if type(text, indexed)
    then start, cat('(',op(0,text), '[', op(1,text), '])'), show
    else start, cat('(.text.)'), show fi
  end;

Dot :=
  proc(radius, P, DisplayStyle)
  local px, py;
    px := 'PS N'(Re &of P);
    py := 'PS N'(Im &of P);
    if Invisible &in DisplayStyle then ''
    elif Hollow &in DisplayStyle
    then 'gsave newpath', px, py, 'translate 0 0', radius,
         '0 360 arc stroke grestore'
    else 'gsave newpath', px, py, 'translate 0 0',
         radius, '0 360 arc fill grestore' fi
  end;

'&in' := member;
```

A.4 Angles

A.4.1 Angle structure

```
structure(A, [x,y]);
```

A.4.2 Angle constructions

```
Right := evalG(A(0,1)):
Straight := evalG(A(-1,0)):

'+ A A' := proc (a1, a2) A(op('* C C'(C(op(a1)),C(op(a2))))) end:
'- A A' := proc (a1, a2) A(op('/ C C'(C(op(a1)),C(op(a2))))) end:

'Argument C' :=
  proc (c)
    'Modulus C'(c);
    A('/ N N'(Re &of c, "), '/ N N'(Im &of c, "))
  end:

# Angle from p1 rotating counterclockwise to p2

'Angle C C' :=
  proc (p1, p2) '- A A'('Argument C'(p2), 'Argument C'(p1)) end:

# Angle from p1 rotating counterclockwise to p3 with p2 as center

'Angle C C C' :=
  proc (p1, p2, p3)
    '- A A'('Argument C'('- C C'(p3, p2)), 'Argument C'('- C C'(p1, p2)))
  end:

'DoubleAngle L L' :=
  proc (l1, l2)
    evalG(Argument((u &of l1 * Conjugate(u &of l2))/
                   (Conjugate(u &of l1) * u &of l2)))
  end:

'Angle S' :=
  proc (seg)
  local P0, P1, P2, x1, y1, x2, y2;
    P1 := p1 &of seg; P2 := p2 &of seg;
    x1 := Re &of P1; x2 := Re &of P2;
    y1 := Im &of P1; y2 := Im &of P2;
    if v &of x1 > v &of x2 then RETURN('Angle S'(S(P2, P1)))
    elif v &of y1 >= v &of y2
```

```
    then 'Angle C C C'(C(x2,y1), P1, P2)
    else 'Angle C C C'(C(x2,y1), P1, P2) fi
  end:

'Angle L' :=
  proc (l)
    u &of l;
    if v &of (Re &of ") = 0 then Right
    elif v &of (Im &of ") = 0 then A(N(1), N(0))
    else evalG(Angle(S(u &of l + w &of l, w &of l))) fi
  end:
```

A.4.3 Angle constraints

```
'= A A' :=
   proc (a1, a2)
   local A, B;
      A := C(op(a1));
      B := C(op(a2));
      '= C C' ('* C C'(A, 'Conjugate C'(B)), '* C C'(B, 'Conjugate C'(A)))
   end;
```

A.4.4 Angle evaluation

```
'FValue A' :=
  proc (a, s) A('FValue N'(x &of a, s), 'FValue N'(y &of a, s)) end;
```

A.4.5 Angle graphical display

```
'PS A' :=
  proc (a, DisplayStyle)
  local r, t;
    r := 'Modulus C'(C(op(a)));
    v &of (x &of a); evalf(");
    if evalf(v &of (x &of a)) >= 0                    # x >= 0
    then t := evalf(arcsin(v &of '/ N N'(y &of a, r)))
    elif evalf(v &of (y &of a)) < 0                   # x < 0, y < 0
    then t := evalf(arctan(v &of '/ N N'(y &of a, x &of a)) + Pi)
    else t := evalf(arccos(v &of '/ N N'(x &of a, r))) fi;
    if t < 0 then t := evalf(t + 2*Pi) fi;
    if DisplayStyle = Radians
    then 'PS N'(t), radians
    elif DisplayStyle = AsPS
    then 'PS N'(evalf(t*360/(2*Pi))) ◦
```

A.5 Segments

A.5.1 Segment structure

```
structure(S, [p1, p2]);
```

A.5.2 Segment transformations

```
'Rotate S' :=
  proc (s, a) S('Rotate C'(p1 &of s, a), 'Rotate C'(p2 &of s, a)) end;

'Translate S' :=
  proc (s, c) S('Translate C'(p1 &of s, c),'Translate C'(p2 &of s,c)) end;

'Scale S' :=
  proc (s, n) S('Scale C'(p1 &of s, n), 'Scale C'(p2 &of s,n)) end;
```

A.5.3 Segment constraints

```
'= S S' :=
    proc(pa, pb)
      '= C C'('p1 S'(pa), 'p1 S'(pb)) union
      '= C C'('p2 S'(pa), 'p2 S'(pb))
    end;

'<> S S' :=
    proc(pa, pb)
      '<> C C'('p1 S'(pa), 'p1 S'(pb)) union
      '<> C C'('p2 S'(pa), 'p2 S'(pb))
    end;

'Parallel S S' :=
  proc (s1, s2) 'Parallel L L'('L S'(s1), 'L S'(s2)) end;

'Perpendicular S S' :=
  proc (s1, s2) 'Perpendicular L L'('L S'(s1), 'L S'(s2)) end;
```

A.5.4 Segment evaluation

```
'FValue S' :=
  proc (seg, s)
    S('FValue C'(p1 &of seg, s), 'FValue C'(p2 &of seg, s))
  end;
```

A.5.5 Segment graphical display

```
'PS S' :=
  proc(s, DisplayStyle)
  local Label;
    if nargs = 3 then Label := args[3] else Label := '' fi;
    gsave, 'PS LineStyle'(DisplayStyle), newpath, 'MoveTo C'(p1 &of s),
    'LineTo C'(p2 &of s), 'stroke grestore'
  end;

'PS LineStyle' :=
   proc(ls)
     if ls = solid then '[1 0 1 0] 0 setdash'
     elif ls = dotted then '[8 8 8 8] 0 setdash'
     elif ls = dashdot then '[1 2 3 2] 0 setdash'
     elif ls = shortdash then '[2 3 2 3] 0 setdash'
     elif ls = longdash then '[5 4 5 4] 0 setdash'
     else ERROR('Unimplemented line style '.ls); fi;
end;

'LineTo C' := proc (A) 'PS N'(Re &of A), 'PS N'(Im &of A), lineto end;

'MoveTo C' := proc (A) 'PS N'(Re &of A), 'PS N'(Im &of A), moveto end;
```

A.6 Lines

A.6.1 Line structure

```
structure(L, [u, w]);
```

A.6.2 Line constructions

```
'LThrough C C' := proc (p1, p2) L(evalG(p1 - p2), p2) end:

'LThrough S' :=
  proc (s)
  local p1, p2;
    p1 := p1 &of s;
    p2 := p2 &of s;
    evalG(LThrough(p1, p2))
  end:

'Parallel L C' := proc(l, p) evalG(LThrough(u &of l, p)); end:

'Perpendicular L C' :=
  proc(l, p) evalG(LThrough(C(0,1) * u &of l, p)) end:

'EquidistantFrom C C' :=
  proc (A, B) evalG(LThrough(C(0,1) * (B - A), 1/2 * (A + B))) end:

'TangentTo Circle C' := proc (c, w) evalG(L(C(0,1)*(z &of c - w), w)) end:

'Chord Circle Circle' :=
  proc (c1, c2)
  local R1, z1, R2, z2, 'z2-z1', 'Cz2-z1', U, W;
    R1 := R &of c1; z1 := z &of c1;
    R2 := R &of c2; z2 := z &of c2;
    'z2-z1' := evalG(z2 - z1);
    'Cz2-z1' := evalG(Conjugate('z2-z1'));
    U := evalG(C(0,1) * 'z2-z1');
    W := evalG(z1+Inverse(2*'Cz2-z1')*(C(R1^2-R2^2, 0)+'z2-z1'*'Cz2-z1'));
    L(U, W)
  end:
```

A.6.3 Line transformations

```
'Rotate L' :=
  proc (l, a)
    evalG(LThrough(Rotate(u &of l + w &of l, a), Rotate(w &of l, a)))
```

```
  end;

'Translate L' :=
  proc (l, c)
    evalG(LThrough(Translate(u &of l + w &of l, c), Translate(w &of l, c)))
  end;

'Scale L' := proc (l, number) l end;
```

A.6.4 Line constraints

```
'On C L' :=
  proc (z, l)
    '= C C'('* C C'(u &of l, 'Conjugate C'('- C C'(z,w &of l))),
            '* C C'('Conjugate C'(u &of l), '- C C'(z,w &of l)))
  end;

'Parallel L L' :=
  proc (l1, l2)
    '= C C'('* C C'(u &of l1, 'Conjugate C'(u &of l2)),
            '* C C'(w &of l1, 'Conjugate C'(w &of l2)))
  end;

'Perpendicular L L' :=
  proc (l1, l1)
    '= C C'('* C C'(u &of l1, 'Conjugate C'(u &of l2)),
            '* C C'('- C'(w &of l1), 'Conjugate C'(w &of l2)))
  end;
```

A.6.5 Line evaluation

```
'FValue L' :=
  proc (l, s) L('FValue C'(u &of l, s), 'FValue C'(w &of l, s)) end;
```

A.6.6 Line graphical display

Line graphical display for a line L is computed as follows:

1. Corners BL ("bottom left") and TR ("top right") are formed from TL and BR.

2. Lines BLBR, TLBL, TLTR and TRBR are formed to define the window.

3. The four intersections I1, I2, I3, I4 of these lines with L are formed. If an intersection point has ∞ or $\perp$ components, or is outside of the

box formed by TL and BR, then these are discarded. The remaining two intersection points will be the endpoints of the line segment seg, which is then displayed.

4. The angle of seg is computed, and arrows, one at that angle and one at the negative of the angle, are displayed, one at either end of seg.

```
'PS L' :=
  proc (l, Label, DisplayStyle, TL, BR)
  local TR, BL, BLBR, TLBL, TLTR, TRBR, I, sl, slop, seg;
  option remember;
    TR := C(Re &of BR, Im &of TL);
    BL := C(Re &of TL, Im &of BR);
    BLBR := evalG(LThrough(BL, BR));
    I := {};
    I := I union {GoodPoint(evalG(Intersection(l, BLBR)), BL, BR, Re)};
    TLBL := evalG(LThrough(TL, BL));
    I := I union {GoodPoint(evalG(Intersection(l, TLBL)), BL, TL, Im)};
    do if nops(I) = 2 then break fi;
       TLTR := evalG(LThrough(TL, TR));
       I := I union {GoodPoint(evalG(Intersection(l, TLTR)), TL, TR, Re)};
       if nops(I) = 2 then break fi;
       TRBR := evalG(LThrough(TR, BR));
       I := I union {GoodPoint(evalG(Intersection(l, TRBR)), BR, TR, Im)};
       break;
    od;
    if nops(I) < 2 then RETURN(NULL) fi;
    if v &of (Re &of I[1]) <= v &of (Re &of I[2])
    then seg := S(I[1], I[2]) else seg := S(I[2], I[1]) fi;
    sl := evalG(Angle(l));
    slop := evalG(sl + Straight);
    'PS C'(p1 &of seg, {Label}, cat(' ', Label)),
    'PS Arrow'(p1 &of seg, slop), 'PS S'(seg, solid),
    'PS Arrow'(p2 &of seg, sl)
end;

GoodPoint :=
  proc (p, low, hi, access)
  local x;
    x := v &of (access &of p);
    if DefinedPoint(p)
    then if (v &of (access &of low)) <= x
         then if x <= v &of (access &of hi) then p fi fi fi
  end;

DefinedPoint :=
```

```
  proc (p)
    not ((v &of (Re &of p)) &in {Infinity, Bottom} or
         (v &of (Im &of p)) &in {Infinity, Bottom})
  end;

'PS Arrow' :=
  proc(P, A)
  local origin, botend, tip, topend;
    origin := C(N(0),N(0));
    botend := C(N(-2), N(-2));
    topend := C(N(-2), N(2));
    tip := C(N(7),N(0));
    'gsave', 'PS C'(P,{AsPS}), translate,
    'PS A'(A, AsPS), 'rotate newpath 1.2 1.2 scale',
    'MoveTo C'(origin), 'LineTo C'(botend),
    'LineTo C'(tip), 'LineTo C'(topend),
    'LineTo C'(origin), 'fill grestore'
end;
```

A.7 Circles

A.7.1 Circle structure

```
structure(Circle, [R, z]);
```

A.7.2 Circle transformations

```
'Rotate Circle' :=
  proc (c, a) Circle(R &of c, 'Rotate C'(z &of c, a)) end;

'Translate Circle' :=
  proc (c, p) Circle(R &of c, 'Translate C'(z &of c, p)) end;

'Scale Circle' :=
  proc (c, n) Circle('* N N'(n, R &of c), z &of c) end;
```

A.7.3 Circle constraints

```
'On C Circle' := proc(p, c) '= N N'('Distance C C'(p, z &of c), R &of c) end;
```

A.7.4 Circle evaluation

```
'FValue Circle' :=
  proc (c, s)
    Circle('FValue N'(R &of c, s), 'FValue C'(s &of c, s))
  end;
```

A.7.5 Circle graphical display

```
'PS Circle' :=
  proc (C, Label, DisplayStyle)
    z &of C;
    'gsave newpath', 'PS LineStyle'(DisplayStyle),
    'PS N'(Re &of "), 'PS N'(Im &of "), 'translate 0 0',
    'PS N'(R &of C), '0 360 arc stroke',
    'PS N'(R &of C), '15 add neg 0 moveto', cat('(', Label, ')'),
     show, grestore
  end:
```

A.8 Figures

A.8.1 Figure structure

```
structure(F, [constructions]);
```

A.8.2 Generic operations in figures

The package usually does not attempt to give the effect of overloading for procedures in MAPLE. Thus addition for complex numbers is defined as the MAPLE procedure '+ Complex Complex', and the programmer must choose the appropriate + variant once and for all when writing a procedure. Overloading is effectively done in one case, in §9, for the *transformation*, *evaluation* (*floating point approximation*) and *graphical display* of a figure. In these cases, overloading is accomplished dynamically by inspecting the type of the component of a procedure, then forming the name of the appropriate transformation or display procedure, then evaluating the name to obtain and call the procedure. This is relatively slow, and thus is restricted to absolutely necessary cases such as the one just described. But in all cases where the type is forced by the context, the full name of the appropriate variant of the procedure given. Beyond these special "by-hand" cases, the package does not provide a general overloading facility, because the design of a complete type system with overloading is greatly beyond the scope and intent of the package.

```
# Elements of the structure set are equations of structure to
# display arguments

'Op Structure-Set' :=
  proc (Op, structs)
    if nargs > 2 then [args[3..nargs]] else NULL fi;
    map ('Op Structure', structs, Op, '')
  end;

'Op Structure' :=
  proc ('struct = displayargs', Op, xyz)
  local struct, stype, Oper;
    struct := op(1, 'struct = displayargs');
    stype := op(0, struct);
    Oper := cat(Op, ' ', stype);
    Oper(struct, op(xyz)) = op(2, 'struct = displayargs')
  end;
```

```
'Op Struct Args' :=
  proc (Op, struct, args)
  local stype, Oper;
    stype := op(0, struct);
    Oper := cat(Op, ' ', stype);
    Oper(struct, op(args))
  end;
```

A.8.3 Figure transformations

```
'Rotate F' :=
  proc (f, a)
    F('Op Structure-Set'(Rotate, constructions &of f, a))
  end;

'Translate F' :=
  proc (f, c)
    F('Op Structure-Set'(Translate, constructions &of f, c))
  end;

'Scale F' :=
  proc (f, n)
    F('Op Structure-Set'(Scale, constructions &of f, n))
  end;
```

A.8.4 Figure evaluation

```
'FValue Figure' :=
  proc (figure, sigma)
    Figure('Op Structure-Set'(FValue, constructions &of figure, sigma),
           constraints &of figure)
  end;
```

A.8.5 Figure graphical display

```
PSFile :=
  proc (fname, PSpre, fig, PSpost)
  local i, mot;
    PSOpenFile (fname);
    PSWrite(PSpre, fig, PSpost);
    PSCloseFile();
  end;
```

```
PSOpenFile :=
  proc (fname)
    screenwidth := 900; # The only way to stop MAPLE from inserting \ at EOL
    writeto(fname);
    writeln('gsave /Helvetica findfont 10 scalefont setfont');
  end;

PSWrite :=
  proc ()
  local i;
    for i from 1 to nargs
    do if type(args[i], F) then PSWrite('PS F'(evalG(args[i])))
       else writeln(args[i]) fi od
  end;

PSCloseFile :=
  proc ()
    writeln('grestore');
    screenwidth := 79;
    writeto(terminal);
  end;

'PS F' :=
    proc (fig)
      op(map('PS Struct', [op(constructions &of fig)]))
    end;

'PS Struct' :=
  proc ('struct = dispargs')
  local struct, dispargs;
    struct := op(1, 'struct = dispargs');
    dispargs := op(2, 'struct = dispargs');
    'Op Struct Args'(PS, struct, [dispargs])
  end;
```

A.9 Multivariate polynomial set normal form

```
TranslateAll :=
  proc (geqns)
  local step1, step2;
    step1 := 'Noneqns->Eqns'(geqns);
    step2 := 'RFncs->Polys'(step1[RF]);
    table([Polynomials = step2[Polys],
           Vars = Indets(geqns),
           NewVars = step1[NewVars] union step2[NewVars]])
  end:

TranslateEqns :=
  proc (geqns)
  local step1, step2;
    step1 := 'Geqs->Eqs+Noneqs'(geqns);
    step2 := 'RFncs->Polys'('Noneqns->Eqns'(step1[Eq])[RF]);
    table([Nonequations = step1[NonEq],
           Polynomials = step2[Polys],
           Vars = Indets(step1[Eq]),
           NewVars = step2[NewVars],
           NVars = Indets(step1[NonEq])])
  end:
```

A.9.1 The indeterminates in a polynomial expression set

Indets gives the indeterminates in a polynomial expression set.

```
Indets :=
  proc (s)
    map(proc (x) if type(x, name) then x else NULL fi end, indets(s))
  end;
```

A.9.2 Translation of general equations into rational functions

Noneqns->Eqns translates a set of general equations G into a set of rational functions implicitly equated to zero by adding extra indeterminates. It returns a table with fields RF containing the rational functions an NewVars containing the added indeterminates. It works by iteratively applying the procedure Noneqn->Eqn to an initially empty set of new variables and each element of G.

Noneqn->Eqn works by the following method. A general equation e is rewritten to a pair of a rational function r and a set which is either empty

or contains a variable not free in r (a new variable is easily obtained as a local variable in the execution context of the procedure):

- $e_1 = e_2 \Rightarrow (e_1 - e_2, \emptyset)$
- Similarly, $e \leq 0$ if there is a negative distance between e and 0, that is, if there exists a value for an indeterminate z, z not free in e, such that $e + z^2 = 0$. So the rule is

$$e_1 \leq e_2 \Rightarrow (e_1 - e_2 + z^2, \{z\})$$

- $e \neq 0$ if it can be scaled by a constant to 1, that is, if there exists a value for an indeterminate z, z not free in e, such that $ez - 1 = 0$. So the rule is

$$e_1 \neq e_2 \Rightarrow ((e_1 - e_2)z - 1, \{z\})$$

- $e < 0$ if it can be scaled by a positive constant to -1, that is, if there exists a value for an indeterminate z, z not free in e, such that $ez^2 + 1 = 0$. So the rule is

$$e_1 < e_2 \Rightarrow ((e_1 - e_2)z^2 + 1, \{z\})$$

- $e > 0$ and $e \geq 0$ are not considered because these are represented by MAPLE with $<$ and $\leq$.

```
'Noneqns->Eqns' :=
  proc (geqns)
  local V, G, geqn;
    V := {}; G := {};
    for geqn in geqns
    do 'Noneqn->Eqn'(geqn);
        V := V union "[Vars];
        G := G union {""[RF]} od;
    table([RF = G, NewVars = V])
  end:

'Noneqn->Eqn' :=
  proc (geqn)
  local Op, g, z, e, V;
    Op := whattype(geqn);
    V := {};
    e := op(1,geqn) - op(2,geqn);
    if Op = '=' then g := e
```

```
    elif Op = '<=' then g := e + z^2; V := {z}
    elif Op = '<>' then g := e*z - 1; V := {z}
    elif Op = '<' then g := e*z^2 + 1; V := {z}
    else ERROR('Unknown relational operator', Op, ' in', geqn) fi;
    table([RF = simplify(g), Vars = V])
  end:
```

A.9.3 Separating equalities and nonqualities

`Geqs->Eqs+Noneqs` separates a set of general equations into a table with field `NonEq` containing a set of nonequalities and `Eq` containing a set of equalities.

```
'Geqs->Eqs+Noneqs' :=
  proc (geqns)
  local r, eqns, noneqns;
    eqns := {}; noneqns := {};
    for r in geqns
    do if type(r, '=') then eqns := {r} union eqns
       else noneqns := {r} union noneqns fi od;
    table([Eq = eqns, NonEq = noneqns])
  end;
```

A.9.4 Translating rational functions into polynomials

`RFncs->Polys` translates a set of rational functions R implicitly equated to zero into a table containing a set `Polys` of polynomials implicitly equated to zero and a set `NewVars` of new variables that may need to be introduced in the translation process. It works by iteratively applying the procedure `RFnc->Poly` on an initially empty set of new variables and each element of R.

`RFnc->Poly` takes a rational function r implicitly equated to zero. It returns a table of a set of polynomials `Polys` and a set of new variables `NewVars` created in translating r into a set of polynomials. It applies one of the following rewrite rules after applying MAPLE's predefined `simplify` procedure to r, where the pair construction on the right-hand side of a $\Rightarrow$ rule denotes the table structure just described, and where the operation $\cup^T$ applied to two tables with fields `Polys` and `NewVars` returns the table with the same fields which is the union of the fields of the given tables.

- $a/b = 0$ iff $a = 0$ and $b \neq 0$, so the rule is

$$a/b \Rightarrow (\texttt{RFnc->Poly}(a, \{z\}) \cup^T \texttt{RFnc->Poly}(bz - 1, \{z\})$$

- $\sqrt{a} = 0$ iff $a = 0$, so the rule is

$$\sqrt{a} \Rightarrow (\{a\}, \emptyset)$$

- $\sqrt{a} - b = 0$ iff $a - b^2 = 0$, so the rules are

$$\sqrt{a} - b \Rightarrow \texttt{RFnc->Poly}(a - b^2) - b + \sqrt{a} \Rightarrow \texttt{RFnc->Poly}(a - b^2)$$

- Otherwise the rule is

$$(a, V) \Rightarrow (\{a\}, \emptyset)$$

```
'RFncs->Polys' :=
  proc (R)
  local V, G, r;
    V := {}; G := {};
    for r in R
    do 'RFnc->Poly'(simplify(r));
       V := V union "[NewVars];
       G := G union ""[Polys] od;
    table([Polys = G, NewVars = V])
  end:

'RFnc->Poly' :=
  proc (r)
  local z, V, r1, r2, G;
    if type(r, '/')
    then r1 := 'RFnc->Poly'(simplify(Dividend(r)));
         r2 := 'RFnc->Poly'(simplify(Divisor(r)*z-1));
         V := r1[NewVars] union r2[NewVars] union {z};
         G := r1[Polys] union r2[Polys];
    elif type(r, sqrt)
    then V := {}; G := {op(1, r)}
    elif type(r, '+') and type(op(1,r), sqrt)
    then RETURN('RFnc->Poly'(op(1, op(1,r)) - op(2,r)^2))
    elif type(r, '+') and type(op(2,r), sqrt)
    then RETURN('RFnc->Poly'(op(1, op(2,r)) - op(1,r)^2))
    else V := {}; G := {r} fi;
    table([Polys = G, NewVars = V])
  end:

'type//' :=
  proc (f)
    type(f, '*') and type(op(2,f), '^') and op(2, op(2,f)) < 0
  end:
```

```
# The following works only for something of type "/" !

Dividend := proc (f) op(1,f) end:
Divisor := proc (f) op(1, op(2,f)) end:
```

A.9.5 Filtering assignments on general equation sets

```
Filter :=
  proc (Sigma, G)
  local r, s;
    r := {};
    for s in Sigma
    do if map(simplify, subst(s, G)) = {0} then r := r union {s} fi od;
    r
  end;
```

A.10 Dependency declarations and analysis

A.10.1 Structure of an HMS

In an HMS structure, the *dependency graph* is a *table* whose *entries* $N = \langle k, (K, E) \rangle$ are:

- A pair of a *key* k, which is a set of names.
- A set of keys K, which are the descendants of N.
- An *equation set* E, whose indeterminates lie in k and the union of the keys of nodes reachable from N.

```
structure(HMSNode, [Key, Keys, Eqns]):
structure(HMSLabel, [Keys, Eqns]):

'HMS->Node set' :=
  proc (H)
    map (proc (eqn)
           HMSNode(op(1,eqn), Keys &of op(2,eqn), Eqns &of op(2,eqn))
         end,
         convert(H, list, '='));
    {op('')}
  end:

'Node set->HMS' :=
  proc (N)
    table(map(proc (n) Key &of n = HMSLabel(Keys &of n, Eqns &of n) end,N))
  end:

'Redirect arcs to' :=
  proc (g, U)
    HMSNode(Key &of g,
            map(proc (s,U) if s &intersects U then U else s fi end,
                Keys &of g, U),
            Eqns &of g)
  end:

'Arcs not in' :=
  proc (U, nodes)
    'union'(op(map(proc (n, U) map('Non-intersecting arc', Keys &of n, U) end,
                   nodes, U)))
  end:

'Non-intersecting arc' :=
  proc (a, U) if a &intersects U then NULL else a fi end:
```

```
'&intersects' := proc (U, V) nops(U intersect V) > 0 end:
```

A.10.2 Constructions on an HMS

```
Initialize :=
  proc (polys) table([Indets(polys) = HMSLabel({}, polys)]) end:

IndependentWith :=
  proc (V0, H)
  local G, N, E, M, V;
    V := Indets(V0);
    G := 'HMS->Node set'(H);
    N := map(proc (n, V)
               if V &intersects (Key &of n) then n else NULL fi
             end, G, V);
    G := G minus N;
    N := map(proc (n, V)
               HMSNode(Key &of n minus V, Keys &of n, Eqns &of n)
             end, N, V);
    'Filter out equations involving'(V, N);
    N := "[Nodes]; E := ""[Eqns];
    M := HMSNode(V, {}, E);
    N := map(proc(n, V)
               HMSNode(Key &of n, Keys &of n union {V}, Eqns &of n)
             end, N, V) union G union {M};
    'Node set->HMS'(N)
  end:

'Filter out equations involving' :=
  proc (V, N)
  local M, E, n, e, F;
    M := {};
    E := {};
    for n in N do
      F := {};
      for e in Eqns &of n do if V = Indets(e) then F := F union {e} fi od;
      M := M union {HMSNode(Key &of n, Keys &of n, Eqns &of n minus F)};
      E := E union F od;
    table([Nodes = M, Eqns = E])
  end:

IndependentSep :=
  proc (V0, H)
```

```
  local V;
    V := Indets(V0);
    if V = {} then H
    else IndependentSep(V minus {V[1]}, IndependentWith({V[1]}, H)) fi
  end:

Dependent :=
  proc (U0, W0, H)
  local G, NUi, NWi, NU, NW, NUW, UW, U, W, g, gU, gW, Vg, EgU, EgW, e;
    if U0 = {} then RETURN(eval(H)) fi;
    U := Indets(U0); W := Indets(W0);
    G := 'HMS->Node set'(H);
    NUi := {}; NWi := {};
    for g in G
    do Vg := Key &of g;
       if U &intersects Vg
       then G := G minus {g};
            if W &intersects Vg
            then EgU := {}; EgW := {};
                 for e in Eqns &of g
                 do if nops(Indets(e) minus W) = 0
                    then EgW := EgW union {e}
                    else EgU := EgU union {e} fi od;
                 gW := HMSNode(W, Keys &of g, EgW);
                 gU := HMSNode(Vg minus W, {Key &of gW}, EgU);
                 NUi := NUi union {gU};
                 NWi := NWi union {gW};
            else NUi := NUi union {g} fi
       elif W &intersects (Key &of g)
       then G := G minus {g}; NWi := NWi union {g} fi
    od;
    UW := U union W;
    NW := HMSNode('union'(op(map(proc (n) Key &of n end, NWi))),
                  'Arcs not in'(UW, NWi),
                  'union'(op(map(proc (n) Eqns &of n end, NWi))));
    NU := HMSNode('union'(op(map(proc (n) Key &of n end, NUi))),
                  {Key &of NW} union 'Arcs not in'(UW, NUi),
                  'union'(op(map(proc (n) Eqns &of n end, NUi))));
    G := map('Redirect arcs to', G, Key &of NU);
    G := map('Redirect arcs to', G, Key &of NW);
    if not ((Key &of NU) &intersects (Key &of NW))
    then G := G union {NU, NW}
    else NUW := HMSNode(Key &of NU union Key &of NW,
                        Keys &of NU union Keys &of NW,
                        Eqns &of NU union Eqns &of NW);
         G := G union {NUW} fi;
    'Node set->HMS'(G)
```

```
  end:
```

A.10.3 Solution of an HMS

```
Trim :=
  proc (H)
  local nodes, lkeys, inside;
    nodes := 'HMS->Node set'(H);
    map(proc(n) if Keys &of n = {} then (Key &of n = NULL) else NULL fi end,
        nodes);
    map(proc(n, badkeys)
          if Keys &of n = {} then NULL
          else HMSNode(Key &of n, subs(badkeys, Keys &of n), Eqns &of n) fi
        end, nodes, '');
    if nops('') > 0 then 'Node set->HMS'('') else NULL fi
  end:

Leaves :=
  proc (H)
    'HMS->Node set'(H);
    map(proc(n) if Keys &of n = {} then n else NULL fi end, '')
  end:

'Substitute HMS' :=
  proc (sigma, H)
    'HMS->Node set'(H);
    map(proc (n,sigma)
          HMSNode(Key &of n, Keys &of n, subs(sigma, Eqns &of n))
        end, '', sigma);
    'Node set->HMS'('')
  end:

Solve :=
  proc (e, v)
    map(proc (eqn) if op(1,eqn) = op(2,eqn) then NULL else eqn fi end,
        solve(e,v))
  end:

'Solve HMS' :=
  proc (G, p)
  local H, N, Sigma, Phi, HH;
    H := G;
    N := Leaves(H);
    Sigma := map(proc (n,p) [p(Eqns &of n, Key &of n)] end, N, p);
    Phi := 'Compositions of solutions'(Sigma);
    if Phi = NULL then ERROR('Hierarchical system is unsolvable') fi;
```

```
    H := Trim(H);
    if H = NULL then Phi
    elif Phi = {} then 'Solve HMS'(H, p)
    else 'union'(op(map(proc (subsig, p)
                          map('union', 'Solve HMS'(subsig[1],p), subsig[2])
                        end,
                        map(proc (phi, H) ['Substitute HMS'(phi, H), phi] end,
                            Phi, H),
                        p)))
    fi
  end:

'Compositions of solutions' :=
  proc ('set of lists of assignments')
  local first, rest;
    if 'set of lists of assignments' = {} then {}
    elif nops('set of lists of assignments') = 1
    then 'set of lists of assignments'[1]
    else first := 'set of lists of assignments'[1];
         rest := 'set of lists of assignments' minus {first};
        'Compositions of solutions'(
           map(proc ('list of assignments A', 'list of assignments B')
                map (proc (assignment, 'list of assignments')
                       op(map('union', 'list of assignments', assignment))
                     end,
                     'list of assignments A', 'list of assignments B')
               end,
               rest, first))
    fi;
  end:
```

A.11 Geometric expression evaluation

```
evalG :=
  proc (e)
  local sigma, vs, ve, vn, Op;
    if nargs = 2 then sigma := args[2] else sigma := {} fi;
    if type(e, F)
    then F(map(proc (ea, s) evalG(op(1,ea),s) = op(2,ea) end, op(1,e), sigma))
    elif type(e, N) then e
    elif type(e, rational) or type(e,float) then N(e)
    elif type(e, function)
    then OpCall(op(0,e), map(evalG, [op(e)], sigma), sigma)
    elif type(e, set)
    then map(evalG, e, sigma)
    else vs := Indets(sigma); ve := indets(e);
         vn := ve minus vs;
         sigma := map(proc(v) v = N end, vn) union sigma;
         if type(e, name)
         then if subs(sigma, e) = N then N(e) else e fi
         else OpCall(whattype(e), map(evalG, [op(e)], sigma), sigma) fi
    fi
  end:

OpCall :=
  proc (Op, Args, sigma)
    if assigned(``.Op.` Fields`)
    then Op(op(Args))
    else FindOp(Op, Args, sigma); "(op(Args))
    fi
  end:

FindOp :=
  proc (Op, Args, sigma)
    if assigned(Op) then Op
    else map(proc (a, sigma) cat(` `, StructType(sigma, a)) end, Args, sigma);
         cat(Op, op("))  fi
  end:

StructType :=
  proc (sigma, e)
    if type(e, function) then op(0, e)
    elif type(e, name) then subs(sigma, e)
    else whattype(e) fi
  end:
```

References

[1] IEEE. *Logic In Computer Science*, Cambridge, Massachusetts, 16-18 June 1986.

[2] Adobe Systems, Inc. POSTSCRIPT *Language Tutorial and Cookbook.* Addison-Wesley, 1985.

[3] N.K. Bose, editor. *Multidimensional Systems Theory.* D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[4] Bruno Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In *[3]*, pages 184–232, D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[5] Bruno Buchberger, George E. Collins, Rüdiger Loos, and R. Albrecht, editors. *Computer Algebra: Symbolic and Algebraic Computation.* Springer-Verlag, second edition, 1983.

[6] Bruce W. Char, Gregory J. Fee, Keith O. Geddes, Gaston H. Gonnet, and Michael B. Monagan. A tutorial introduction to maple. *Journal of Symbolic Computation*, 2(2):197–200, 1986.

[7] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, and Stephen M. Watt. *MAPLE User's Guide (First Leaves: A Tutorial Introduction to Maple and Maple Reference Manual, 4th Edition).* WATCOM Publications Limited, 415 Phillip Street, Waterloo, Ontario N2L 3X2, Canada; Telephone: (519) 886-3700, Telex: 06-955458, 1985.

[8] Shang-Ching Chou. Proving elementary geometry theorems using Wu's algorithm. *Contemporary Mathematics*, 29:243–286, 1984. American Mathematical Society, Providence, Rhode Island, U.S.A.

[9] Shang-Ching Chou and Hai-Ping Ko. On mechanical theorem proving in Minkowskian plane geometry. In *[1]*, pages 187–192, 1986.

[10] Guy Cousineau and Gérard Pierre Huet. *The CAML Primer.* Projet Formel, Institut National de Recherche en Informatique et en Automatique, B.P. 105-78153 Le Chesnay CEDEX FRANCE, December 1987. Version 2.5.

[11] Lars Warren Ericson and Chee-Keng Yap. The design of LINETOOL, a geometric editor. In Bernard Chazelle, editor, *4th Computational Geometry Conference*, Association for Computing Machinery, June 1988.

[12] Rüdiger Loos. Computing in algebraic extensions. In *[5]*, pages 173–187, Springer-Verlag, 1983.

[13] A.C. Norman. Computing in transcendental extensions. In *[5]*, pages 169–172, Springer-Verlag, 1983.

[14] Jacob Theodore Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the Association for Computing Machinery*, 27(4):701–717, October 1980.

[15] Jean Vuillemin. *Exact real computer arithmetic with continued fractions.* Technical Report, Institut National de Recherche en Informatique et en Automatique, B.P. 105-78153 Le Chesnay CEDEX FRANCE, November 1987. Rapports de Recherche No. 760.

[16] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. *The CAML Reference Manual.* Projet Formel, Institut National de Recherche en Informatique et en Automatique, B.P. 105-78153 Le Chesnay CEDEX FRANCE, December 1987. Version 2.5.

Index

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique