IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 785

A LIBRARY FOR DOING POLYHEDRAL OPERATIONS

DORAN K. WILDE

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 38 38 32

A Library for Doing Polyhedral Operations

Doran K. Wilde**

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Publication interne n° 785 — December 1993 — 45 pages

Abstract: Polyhedra are geometric representations of linear systems of equations and inequalities. Since polyhedra are used to represent the iteration domains of nested loop programs, procedures for operating on polyhedra are useful for doing loop transformations and other program restructuring transformations which are needed in parallelizing compilers. Thus a need for a library of polyhedral operations has recently been recognized in the parallelizing compiler community.

Polyhedra are also used in the definition of domains of variables in systems of affine recurrence equations (SARE). ALPHA is a language which is based on the SARE formalism in which all variables are declared over finite unions of polyhedra. This report describes a library of polyhedral functions which was developed to support the ALPHA language environment, and which is general enough to satisfy the needs of researchers doing parallelizing compilers.

This report describes the data structures used to represent domains, gives the motivations for the major design decisions, and presents the algorithms used for doing polyhedral operations.

This library has been written and tested, and has been in use since the beginning of 1993 by research facilities in Europe and Canada. The library is freely distributed by ftp.

Key-words: Polyhedra, Duality, Geometry, Chernikova, Boolean Functions

(Résumé : tsvp)

*The author may be contacted at `wilde@irisa.fr`

**This work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Une bibliothèque pour Faire les Opérations Polyèdrales

Résumé : Les polyèdres sont des représentations géométriques de systèmes d'équations ou d'inéquations linéaires utilisés pour décrire le domaine d'itération des nids de boucles. Les procédures de calculs sur les polyèdres sont donc nécessaires pour transformer des boucles; elles servent aussi dans de nombreuses transformations effectuées par les compilateurs paralléliseurs. C'est pourquoi le besoin d'une bibliothèque de calculs sur les polyèdres se fait sentir dans la communauté scientifique intéressé par la parallélisation.

Les polyèdres sont aussi utilisés pour représenter les domaines des variables dans les systèmes d'équations récurrentes affines (SARE). Dans le langage ALPHA, basé sur ce formalisme, les domaines des variables sont des unions finies de polyèdres. Ce rapport décrit une bibliothèque de fonctions de calcul sur les polyèdres, développée pour le langage Alpha. Cette bibliothèque est assez générale pour satisfaire les besoins de chercheurs en parallélisation.

Dans ce rapport, on décrit les structures de données utilisées pour représenter les domaines, on donne les raisons des principaux choix effectués, et on présente les algorithmes permettant les calculs sur les polyèdres.

Cette bibliothèque a été programmée et testée, et est utilisée depuis le début de 1993 par plusieurs centres de recherche en Europe et au Canada. Elle est distribuée librement par ftp.
Mots-clé : Polyèdres, Dualité, Géometrie, Chernikova, Fonctions Booléanes

Chapter 1

INTRODUCTION

Much work has been done recently in the development of methods to do synthesis, analysis and verification of systems of recurrence equations in order to find equivalent parallel forms of these algorithms suitable for implementation. The ultimate goal is to transform an algorithm from a mathematical type of description into an equivalent form that can be implemented either with special purpose hardware (with systolic arrays for instance) or implemented as a program which is able to run on a multiprocessor system. The ALPHA language was invented to be able to do just this kind of program transformation [Mau89, in French] [LMQ91, in English].

The work presented here was done in connection with the implementation of an ALPHA environment based on the commercially available MATHEMATICA system. This environment is illustrated in figure 1.1.

The ALPHA environment is a toolchest from which different transformations can be selected to move a program from its current state toward a target state. A change of basis of a program variable is an example of an ALPHA program transformation. Transformations such as loop reindexing (e.g. index skewing, loop exchanges), uniformalization of communication and space-time mapping are all examples of doing changes of bases of program variables. Time schedules may be reflected back into the program by performing a change of basis on program variables, transforming one (or more) of the variable indices into time indices. To perform these transformations, there are certain computations involving unions of polyhedral domains that have to be performed.

1.1 The Role of the Polyhedral Library

In figure 1.1, there is a block labeled "Polyhedral Library". This is a library which operates on objects called *domains* made of unions of polyhedra [Wil93]. When specifying a system of affine recurrence equations, unions of polyhedra are used to describe the domains of computation of system variables. Whereas a polyhedron is a region containing an infinite number of rational[1] points, a *domain*, as the term is used in this report, refers to the set of integral points which are inside a polyhedron (or union of polyhedra). Figure 1.2 illustrates this difference.

Definition 1.1 *A polyhedral* **domain** *of dimension n is defined as*

$$\mathcal{D} : \{i \mid i \in \mathcal{Z}^n, i \in \mathcal{P}\} = \mathcal{Z}^n \cap \mathcal{P} \qquad (1.1)$$

where $\mathcal{P}$ is a union of polyhedra of dimension n.

[1] Polyhedra may also be defined over the reals, however, only rationals are considered in this report.

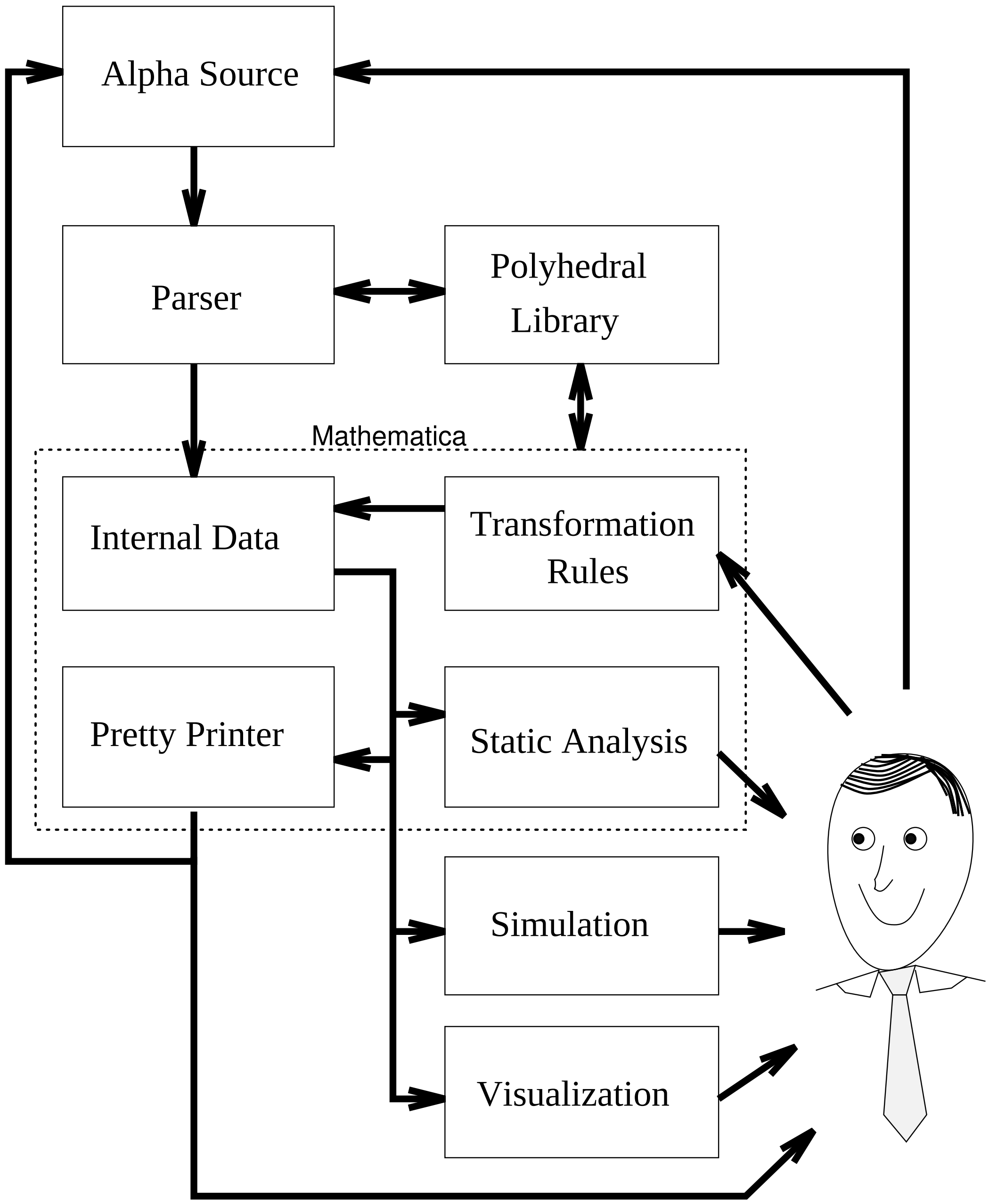


Figure 1.1: Alpha System Environment

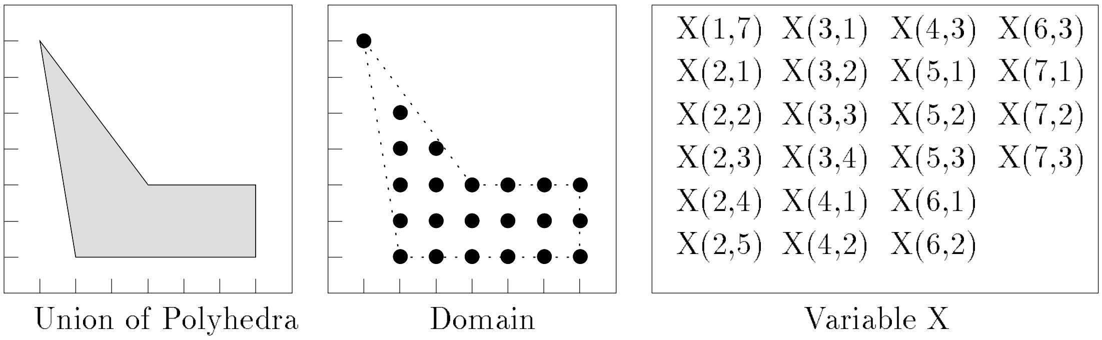


Figure 1.2: Comparison of Polyhedra, Domain, and Variable

In affine recurrence equations of the type considered here and in the ALPHA language, every variable is declared over a domain. Elements of a variable are in a one-to-one correspondence with points in a domain. Again, figure 1.2 illustrates this. Here, we formalize the definition of a variable.

Definition 1.2 *A* **variable** X *of type* "datatype" *declared over a domain* $\mathcal{D}$ *is defined as*

$$X ::= \{\ X_i \ :\ X_i \in \text{datatype},\ i \in \mathcal{D}\ \} \tag{1.2}$$

where X_i *is the element of* X *corresponding to the point* i *in domain* $\mathcal{D}$.

X can also be thought of as a function: $X : i \in \mathcal{D} \mapsto X_i \in \textit{datatype}$.

In order to be able to manipulate ALPHA variables, a library of "domain functions" is needed. This library is the geometric engine of the language and provides the capabilities needed for programs to be analyzed and transformed. Examples of domain operations which can be performed by the library are: Image, Preimage, Intersection, Difference, Union, ConvexHull, and Simplify. The implementation of these and other library functions are described in detail in chapter 4.

Even though the library was written to support the ALPHA environment, it is also general purpose enough to be used by other applications as well.

1.2 Summary of Chapters

Chapter 2 is background information and a review of the fundamental definitions relating to polyhedra. Chapter 3 discusses issues relating to how a polyhedron is represented in memory, and the polyhedron data structure is developed and presented in detail. Chapter 4 describes the polyhedral library itself, giving the basic algorithms for all of the operations. Chapter 5 is a conclusion and summary of the report.

Chapter 2

POLYHEDRA

Polyhedra have been studied in several related fields: from the geometric point of view by computational geometrists [Gru67], from the algebraic point of view by the operations research and linear programming communities [Sch86], and from the structural/lattice point of view by the combinatorics community [Ede87]. Each community has a different view of polyhedra so the notation and terminology are sometimes different between the different disciplines.

This chapter is a review of fundamental definitions relating to polyhedra and cones. I have taken the majority of this summary from the works of Grunbaum, "Convex Polytopes" [Gru67], and of Schrijver, "Theory of Linear and Integer Programming" [Sch86], and of Edelsbrunner, "Algorithms in Combinatorial Geometry" [Ede87]. Other references used are [Weh50, KT56].

2.1 Notation and Prerequisites

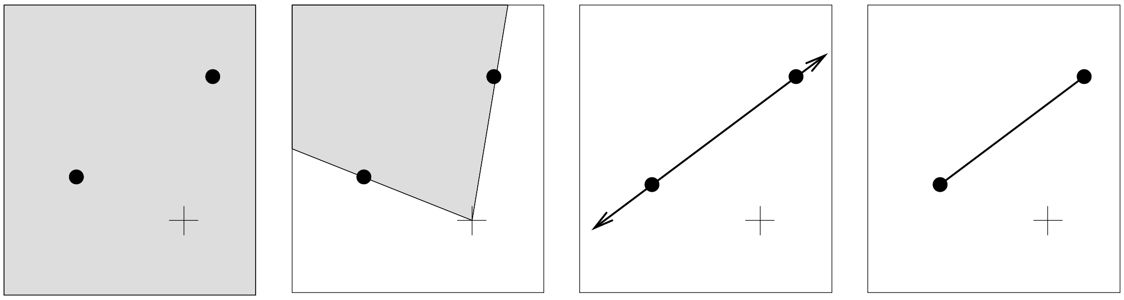
In this presentation, polyhedra are restricted to being in the n-dimensional rational Cartesian space, represented by the symbol $\mathcal{Q}^n$. All matrices, vectors, and scalars are thus assumed to be rational unless otherwise specified.

Definition 2.1 *The* **scalar product** $a \circ b$ *is defined as* $a \circ b = a^T b = \sum_{i=1}^{n} a_i b_i$ *where* $a = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$ *and* $b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$.
$a \circ b = 0$ *iff vectors* a, b *are* **orthogonal**.

Definition 2.2 *Given a vector x and a scalar coefficient vector λ, the following different combinations are defined:*
A **linear combination** $\sum \lambda_i x_i$
A **positive**[1] **combination** $\sum \lambda_i x_i$ *where all* $\lambda_i \geq 0$
An **affine combination** $\sum \lambda_i x_i$ *where* $\sum \lambda_i = 1$
A **convex combination** $\sum \lambda_i x_i$ *where* $\sum \lambda_i = 1$ *and all* $\lambda_i \geq 0$.

Figure 2.1 shows the geometries generated by the different combinations of two points in 2-space (with origin marked '+').

[1] Also called *non-negative* or *conic* combination



Linear Combination | Positive Combination | Affine Combination | Convex Combination

Figure 2.1: Geometric Interpretations of the Combinations of Two Points

2.2 Sets

A set, in this context, always refers to a set of points in space $\mathcal{Q}^n$. Most definitions have meanings on any set of points (not necessarily polyhedral). These definitions are introduced in this section.

Definition 2.3 *Given a non-zero vector y and a constant α, the following objects (sets of points) are defined:*
A **hyperplane** $\mathcal{H} = \{x \mid x \circ y = \alpha\}$
A **open half-space** $\mathcal{H} = \{x \mid x \circ y > \alpha\}$
A **closed half-space** $\mathcal{H} = \{x \mid x \circ y \geq \alpha\}$

Definition 2.4 *A* **vertex** *of a set $\mathcal{K}$ is any point in $\mathcal{K}$ which cannot be expressed as a convex combination of any other distinct points in $\mathcal{K}$.*

Definition 2.5 *A* **ray** *of $\mathcal{K}$ is a vector r, such that $x \in \mathcal{K}$ implies $(x + \mu r) \in \mathcal{K}$ for all $\mu \geq 0$.*

A ray is not a set of points, but a direction in which $\mathcal{K}$ is infinite. A ray may be considered as a point at infinity in the direction of r.

Definition 2.6 *A ray of $\mathcal{K}$ is an* **extreme ray** *if and only if it cannot be expressed as a positive combination of any other two distinct rays of $\mathcal{K}$. The set of extreme rays form a basis which describes all directions in which the convex set is open. Extremal rays are unique up to a multiplicative constant.*

An extreme ray may be considered as a vertex at infinity in the direction of r.

Definition 2.7 *A* **line** *(or* **bidirectional ray***) of $\mathcal{K}$ is a vector l, such that $x \in \mathcal{K}$ implies $(x+\mu l) \in \mathcal{K}$ for all μ.*

Allowing μ to have both positive and negative values creates a bidirectional ray in the direction of l and $-l$. Two rays in exactly opposite directions, therefore make a line. The definition of line is very much like the definition of ray (2.5), however, there is no such thing as an extreme line in general. Lines are used to describe n-spaces which are described in definition 2.12 and property 2.3.

Definition 2.8 *Given two points x, y, the closed (line)* **segment** *$Seg(x, y)$ is defined as the set of all convex combinations of x and y.*

Definition 2.9 *An* **affine transformation** *is a function T which maps a point x to a point $x.T = Ax + b$ where A is a constant matrix and b is a constant vector.*

Definition 2.10 *A set $\mathcal{K}$ is* **convex** *iff every convex combination of any two points in $\mathcal{K}$ is also a point in $\mathcal{K}$.*

Alternate definition:

A set $\mathcal{K}$ is **convex** *iff for each pair of points $a, b \in \mathcal{K}$, the closed segment with endpoints a,b is also included in set $\mathcal{K}$.*

Alternate definition:

A set $\mathcal{K}$ is **convex** *iff its intersection with any line is either empty or a connected set (line, half-line, line-segment).*

The following are important closure properties held by convex sets.

Property 2.1 *(Closure under intersection)*
The intersection of convex sets is convex.

Property 2.2 *(Closure under affine transformations)*
Affine transformations of convex sets are convex.

Definition 2.11 *A set of points are* **linearly independent** *iff no point in the set can be expressed as an linear combination of any other points in the set. A set of points are* **linearly dependent** *iff they are not linearly independent. A* **basis** *of a set is a linearly independent subset such that all points in the original set can be expressed as a linear combination of points in the basis. In general, the basis is not unique. The* **rank** *of a set is the size of its basis. Similary definitions for* **affinely independent** *and* **affinely dependent** *may be given in terms of affine combinations.*

Definition 2.12 *A set $\mathcal{K}$ is called a* **linear subspace**, *(also* **subspace** *or* **space**), *if it has the property: $x, y \in \mathcal{K}$ implies all linear combinations of x, y are in $\mathcal{K}$. The* **dimension** *of a space is the rank of a set of lines which span the space. A space of dimension n is called an n-***space**.

Property 2.3 *Each n-space contains n linearly independent lines. Any $n + 1$ membered set of lines in an n-space is linearly dependent.*

Definition 2.13 *A set $\mathcal{K}$ is called a* **flat** *if it has the property: $x, y \in \mathcal{K}$ implies all affine combinations of x, y are in $\mathcal{K}$. The* **dimension** *of a flat is the rank of a set of lines which span the flat. A flat of dimension n is called an n-***flat**. *A 0-flat, 1-flat, and 2-flat are called respectively a* **point**, **line**, *and* **plane**.

Property 2.4 *Each n-flat contains n affinely independent lines and $n + 1$ affinely independent points. Any $n + 2$ membered set of points in an n-flat is affinely dependent. Any $n + 1$ membered set of lines in an n-flat is affinely dependent.*

2.3 Hulls

Table 2.1 summarizes the four kinds of hulls (containers) corresponding to the four kinds of combinations. Also shown in the table are the largest contained subsets.

Definition 2.14 *The* **convex hull** *of $\mathcal{K}$ is the convex combination of all points in $\mathcal{K}$. It is the smallest convex set which contains all of $\mathcal{K}$.*

Definition 2.15 *The* **affine hull** *of $\mathcal{K}$ is the flat consisting of the affine combination of all points in $\mathcal{K}$. It is the smallest dimensional flat which contains all of $\mathcal{K}$.*

Definition 2.16 *The* **linear hull** *of $\mathcal{K}$ is the subspace consisting of the linear combination of all points in $\mathcal{K}$. It is the smallest dimensional linear subspace which contains all of $\mathcal{K}$.*

type ↓	Smallest Container	Largest Contained Subset
Linear	Linear Hull (definition 2.16)	Lineality Space (definition 2.22)
Positive	Conic Hull (definition 2.17)	Characteristic Cone (definition 2.21)
Affine	Affine Hull (definition 2.15)	
Convex	Convex Hull (definition 2.14)	

Table 2.1: Comparison of Containers

Definition 2.17 *The* **conic hull** *of $\mathcal{K}$ is the cone consisting of the positive combination of all points in $\mathcal{K}$. It is the smallest cone which contains all of $\mathcal{K}$.*

Definition 2.18 *A convex set $\mathcal{C}$ is a* **cone** *with* **apex** *0 provided λx is in $\mathcal{C}$ whenever x is in $\mathcal{C}$ and $\lambda \geq 0$. A set $\mathcal{C}$ is a cone with apex x_0 provided $\mathcal{C} - \{x_0\}$ is a cone with apex 0. A cone with apex x_0 is* **pointed** *provided x_0 is a vertex of $\mathcal{C}$.*

Property 2.5 *If a cone $\mathcal{C}$ is pointed, $\mathcal{C}$ is generated by a positive combination of its extremal rays.*

2.4 The Polyhedron

The following theorem was first published in 1894 by Farkas and has been sharpened through the years. It provides us the basis upon which to build a theory for polyhedra.

Theorem 2.1 Fundamental Theorem of Linear Inequalities
Let $a_1, \cdots, a_m, b$ be vectors in n-dim space.
Then either:
1. b is a positive combination of linearly independent vectors $a_1, \cdots, a_m$; or,
2. there exists a hyperplane $\{x \mid cx = 0\}$, containing $t-1$ linearly independent vectors from among $a_1, \cdots, a_m$, such that $cb < 0$ and $ca_1, ..., ca_m \geq 0$, where $t := rank\{a_1, \cdots, a_m, b\}$.

For a proof, refer to [Sch86, page 86].

Stated in more familiar terms, given a cone generated by a set of rays $\{a_1, \cdots, a_m\}$, then given another ray b, either
1. b is in the cone and is therefore a positive combination of rays $\{a_1, \cdots, a_m\}$, or
2. b is outside the cone, and there exists a hyperplane containing $(t-1)$ extreme rays from the set $\{a_1, \cdots, a_m\}$ which separates b from the cone.

2.4.1 The dual representations of polyhedra

Definition 2.19 *A* **polyhedron**, *$\mathcal{P}$ is a subspace of Q^n bounded by a finite number of hyperplanes.*

Alternate definition:
$\mathcal{P}$ is the intersection of a finite family of closed linear halfspaces of the form $\{x \mid ax \geq c\}$ where a is a non-zero row vector and c is a scalar constant.

Property 2.6 *All polyhedra are convex.*

A result of the fundamental theorem is that a polyhedron $\mathcal{P}$ has a dual representation, an implicit and a parametric representation. The set of solution points which satisfy a mixed system of constraints form a polyhedron $\mathcal{P}$ and serve as the *implicit definition* of the polyhedron

$$\mathcal{P} = \{x \mid Ax = b,\ Cx \geq d\} \tag{2.1}$$

given in terms of equations (rows of A, b) and inequalities (rows of C, d), where A, C are matrices, and b, d and x are vectors. This form corresponds to definition 2.19 above, where the set of closed halfspaces are defined by the inequalities: $Ax \geq b$, $Ax \leq b$, and $Cx \geq d$.

$\mathcal{P}$ has an equivalent dual *parametric representation* (also called the *Minkowski characterisation* after Minkowski— 1896 [Sch86, Page 87]) :

$$\mathcal{P} = \{x \mid x = L\lambda + R\mu + V\nu, \quad \mu, \nu \geq 0, \ \sum \nu = 1\} \tag{2.2}$$

in terms of a linear combination of lines (columns of matrix L), a convex combination of vertices[2] (columns of matrix V), and a positive combination of extreme rays (columns of matrix R). The parametric representation shows that a polyhedron can be generated from a set of lines, rays, and vertices. The fundamental theorem implies that two forms (eq. 2.1 and eq. 2.2) are equivalent.

Procedures exist to compute the dual representations of $\mathcal{P}$, that is, given A, b, C, d, compute L, V, R, and visa versa. Such a procedure is in the polyhedral library and will be described later in section 4.2.

2.5 The Polyhedral Cone

Polyhedral cones are a special case of polyhedra which have only a single vertex. (Without loss of generality, the vertex is at the origin.) A cone $\mathcal{C}$ is defined parametrically as

$$\mathcal{C} = \{x \mid x = L\lambda + R\mu,\ \mu \geq 0\} \tag{2.3}$$

where L and R are matrices whose columns are the lines and extreme rays, respectively, which specify the cone with rays $\{R, -L, L\}$ as defined in definition 2.18 and property 2.5. If L is empty, then the cone is pointed.

Since the origin is always a solution point in Eq. 2.1, the implicit description of a cone has the following form

$$\mathcal{C} = \{x \mid Ax = 0,\ Cx \geq 0\} \tag{2.4}$$

the solution of a mixed system of *homogeneous* inequalities and equations.

2.6 The structure of polyhedra

In this section, let $\mathcal{P}$ be a polyhedron as described in section 2.4.1.

Definition 2.20 *A set is a (convex)* **polytope** *iff it is the convex hull of finitely many vertices. A set $\mathcal{K}$ is a polytope iff $\mathcal{K}$ is bounded (contains no rays or lines).*

Definition 2.21 char.cone$\mathcal{P}$*, called the* **characteristic cone** *(or recession cone) of $\mathcal{P}$ is the cone* $\{y \mid x + y \in \mathcal{P},\ \forall x \in \mathcal{P}\} = \{y \mid Ay \geq 0\}$.

Theorem 2.2 Decomposition Theorem for Polyhedra *A set $\mathcal{P}$ is a polyhedron iff $\mathcal{P} = \mathcal{V} + \mathcal{C}$, where $\mathcal{V}$ is a polytope, and $\mathcal{C} =$ char.cone$\mathcal{P}$ is a polyhedral cone. [Motzkin 1936]*

[2] I am taking liberty with the term *vertices*. Here I use the term to mean the vertices of P less its lineality space.

The proof is in [Sch86, Page 88].

Definition 2.22 *The* **lineality space** *of $\mathcal{P}$ is defined as*
lin.space$\mathcal{P}$:= char.cone$\mathcal{P} \cap -$char.cone$\mathcal{P} = \{y \mid Ay = 0\}$. *The lineality space of a polyhedron is the dimensionally largest linear subspace contained in the polyhedron. If lineality space of $\mathcal{P}$ is empty then $\mathcal{P}$ is* **pointed**.

A lineality space is represented as a fundamental set of lines which form a basis of the subspace. The lineality space of a polyhedron is unique, although it may be represented using any appropriate basis of lines. The dimension of a lineality space is the rank of a set of lines which span the space (property 2.3).

2.6.1 Decomposition

In 1936, Motzkin gave the decomposition theorem (2.2) for polyhedra. Any polyhedron $\mathcal{P}$ can be uniquely decomposed into a polytope $\mathcal{V} = \text{conv.hull}\{v_1, \cdots, v_m\}$ generated by convex combination of the extreme vertices of $\mathcal{P}$, and a cone $\mathcal{C} = \text{char.cone}\mathcal{P}$ as follows[3]

$$\mathcal{P} = \mathcal{V} + \mathcal{C} \cdot \tag{2.5}$$

A non-pointed convex cone can in turn be partitioned into two parts,

$$\mathcal{C} = \mathcal{L} + \mathcal{R} \tag{2.6}$$

the combination of its lineality space $\mathcal{L}$ generated by a linear combination of the lines (bidirectional-rays) of $\mathcal{P}$, and a pointed cone $\mathcal{R}$ generated by positive combination of the extreme rays of $\mathcal{P}$. Combining equations 2.5 and 2.6, a polyhedron may be fully decomposed into

$$\mathcal{P} = \mathcal{V} + \mathcal{R} + \mathcal{L} \cdot \tag{2.7}$$

Decomposition implies that any polyhedron may be decomposed into its vertices, rays (unidirectional rays) and lines (bidirectional rays) which can be clearly seen in the parametric description in equation 2.2.

A decomposition of a polyhedron which has a practical application in the polyhedron library, is the decomposition of a polyhedron into its *lineality space* (definition 2.22) and its *ray space*. This division separates lines (bidirectional rays) from vertices and rays (unidirectional rays). In the alternate conic form of a polyhedron, developed in section 3.1, both rays and vertices are representable as unidirectional rays in the cone. In a cone, this decomposition simply separates lines and rays (equation 2.6). Table 2.2 summarizes all of the decompositions of a polyhedron.

2.7 Duality of Polyhedra

In this section, the concepts of combinatorial equivalence and duality are presented. These two concepts are used in developing a memory representation of a polyhedron in chapter 3. Then the idea of the polar mapping is presented along with its properties which are used in chapter 4 in the development of operations on polyhedra.

2.7.1 Combinatorial Types of Polytopes

Definition 2.23 *Two polyhedra, $\mathcal{P}$ and $\mathcal{P}'$ are* **combinatorially equivalent** *(or* **isomorphic***) provided there exists a 1-1 mapping between the set F of all faces of $\mathcal{P}$, and the set F' of all faces*

[3] The symbol '+' in the equation is called the Minkowski sum, and is defined: $R + S = \{r + s \ : \ r \in R, \ s \in S\}$.

Decomposition	Description
$\mathcal{V}+\mathcal{R}+\mathcal{L}$	$\mathcal{P} = \{x \mid x = L\lambda + R\mu + V\nu, \quad \mu, \nu \geq 0, \ \sum \nu = 1\}$, (eq 2.2)
$\mathcal{V}$	polytope $= \{x \mid x = V\nu, \ \nu \geq 0, \ \sum \nu = 1\}$, (definition 2.20)
$\mathcal{R}$	pointed cone $= \{x \mid x = R\mu, \ \mu \geq 0\}$, (definition 2.18)
$\mathcal{L}$	lineality space $= \{x \mid x = L\lambda\}$, (definition 2.22)
$\mathcal{V}+\mathcal{R}$	ray space of $\mathcal{P}$
$\mathcal{V}+\mathcal{L}$	set of minimal faces of $\mathcal{P}$
$\mathcal{R}+\mathcal{L}$	char.cone$\mathcal{P}$, a non-pointed cone, (definition 2.21)

Table 2.2: Table of Decompositions

of $\mathcal{P}'$, such that the mapping is inclusion preserving. In other words, F_1 is a face of F_2 iff map(F_1) is a face of map(F_2). Equivalently, the face lattices of $\mathcal{P}$ and $\mathcal{P}'$ are isomorphic. Combinatorial equivalence is an equivalence relation.

Definition 2.24 *Two d-polytopes, $\mathcal{P}$ and $\mathcal{P}*$ are said to be* **dual** *to each other provided there exists a 1-1 mapping between the set F of all faces of $\mathcal{P}$, and the set $F*$ of all faces of $\mathcal{P}*$, such that the mapping is inclusion-reversing. in other words, F_1 is a face of F_2 iff map(F_2) is a face of map(F_1).*

2.7.2 Polar mapping

Definition 2.25 *(Polar)*
Given a closed convex set P containing the point 0, then the polar *$P*$ is defined as*
$P* = \{y \mid \forall x \in \mathcal{P} \ : \ x \circ y \geq 0\}$.

Property 2.7 *(duality of polars)*
If $P*$ is the polar of P, then P and $P*$ are duals of each other.

Given $\mathcal{P} \overset{\text{DUAL}}{\longleftrightarrow} \mathcal{P}*$ where $\mathcal{P}$ is a closed convex set containing 0, then the following properties hold:

(i) if $\mathcal{P} = \text{conv.hull}\{0, x_1, \cdots, x_m\} + \text{cone}\{y_1, \cdots, y_t\}$
then $\mathcal{P}* = \{z \mid z^T x_i \leq 1 \text{ for } i = 1 \cdots m\} + \{z \mid z^T y_i \leq 0 \text{ for } i = 1 \cdots t\}$

(ii) $\mathcal{P}$ has dimension k iff lin.space($\mathcal{P}*$) has dimension $n-k$

(iii) $\mathcal{P}** = \mathcal{P}$

(iv) $\mathcal{A}* = \mathcal{B}*$ iff $\mathcal{A} = \mathcal{B}$

(v) $\mathcal{A}* \subseteq \mathcal{B}*$ iff $\mathcal{A} \subseteq \mathcal{B}$

(vi) $(\mathcal{A} \cup \mathcal{B})* = \mathcal{A}* \cap \mathcal{B}*$

(vii) $(\mathcal{A} \cap \mathcal{B})* = \text{convex.hull}(\mathcal{A}* \cup \mathcal{B}*)$

(viii) if $\mathcal{A}$ is a face of $\mathcal{B}$ then $\mathcal{B}*$ is a face of $\mathcal{A}*$.

(ix) there is a 1-1 correspondance between k-faces of $\mathcal{P}$ and $(n-k)$-faces of $\mathcal{P}*$.

The principle of duality is used in sections 3.2 and 3.3 when showing the duality between the parametric and implicit definitions of a polyhedron and in chapter 5 when discussing the lattices of dual polyhedra.

Chapter 3

REPRESENTATION OF POLYHEDRA

3.1 Equivalence of homogenous and inhomogenous systems

We want to be able to represent a *mixed inhomogeneous system* of equations as given in equations 2.1 and 2.2 and which is the most general type of constraint system. A memory representation of an n dimensional mixed inhomogeneous system of j equalities and k inequalities would require the storage of the following arrays: $A(j \times n),\ b(j \times 1),\ C(k \times n),\ d(k \times 1)$. The dual representation would require the storage of R, V, and L, the arrays representing the rays, vertices, and lines. The representation in memory can be simplified however, with a transformation $x \to \begin{pmatrix} \xi x \\ \xi \end{pmatrix}$, $\xi \geq 0$ that changes an inhomogeneous system $\mathcal{P}$ of dimension n into a homogenous system $\mathcal{C}$ of dimension $n+1$, as shown here:

$$
\begin{aligned}
\mathcal{P} &= \{x \mid Ax = b,\ Cx \geq d\} \\
&= \{x \mid Ax - b = 0,\ Cx - d \geq 0\} \\
\mathcal{C} &= \left\{ \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \mid \xi Ax - \xi b = 0,\ \xi Cx - \xi d \geq 0,\ \xi \geq 0 \right\} \\
&= \left\{ \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \mid \begin{pmatrix} A \mid -b \end{pmatrix} \begin{pmatrix} \xi x \\ \xi \end{pmatrix} = 0,\ \left(\begin{array}{c|c} C & -d \\ \hline 0 & 1 \end{array}\right) \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \geq 0 \right\} \\
&= \{\hat{x} \mid \hat{A}\hat{x} = 0,\ \hat{C}\hat{x} \geq 0\}
\end{aligned}
$$

The transformed system $\mathcal{C}$ is now an $(n+1)$ dimensional cone which contains the original n dimensional polyhedron. Goldman showed that the mapping $x \to \begin{pmatrix} \xi x \\ \xi \end{pmatrix}$ is one to one and inclusion preserving [Gol56] and thus by definition 2.23 the two are combinatorially equivalent. The original polyhedron $\mathcal{P}$ is in fact the intersection of the cone $\mathcal{C}$ with the hyperplane defined by the equality $\xi = 1$. Given any $\mathcal{P}$ as defined in equation 2.1, an unique homogeneous cone form exists defined as follows:

$$
\begin{aligned}
\mathcal{C} &= \{\hat{x} \mid \hat{A}\hat{x} = 0,\ \hat{C}\hat{x} \geq 0\} \\
&= \text{homogoneous.cone } \mathcal{P}, \\
\text{where } \hat{x} &= \begin{pmatrix} \xi x \\ \xi \end{pmatrix},\ \hat{A} = \begin{pmatrix} A \mid -b \end{pmatrix},\ \hat{C} = \left(\begin{array}{c|c} C & -d \\ \hline 0 & 1 \end{array}\right)
\end{aligned}
\tag{3.1}
$$

The storage requirement for the homogenous system is $\hat{A}(j \times (n+1))$, $\hat{C}((k+1) \times (n+1))$ which is about the same amount of memory needed for the original system (compare $(j+k)(n+1)+n)$ words for the cone versus $(j+k)(n+1)$ words for the polyhedron) and the cone representation is simpler (two matrices versus two matrices and two vectors). Likewise the dual representation of the cone is simpler. The decomposition of a cone is $\mathcal{R} + \mathcal{L}$, and thus only rays and lines have to be represented. During the transformation process from a polyhedron to a cone, vertices get transformed into rays. The vertices and rays of an inhomogeneous polyhedron have a unified and homogenous representation as rays in a polyhedral cone. Thus the rays of the cone represent both the vertices and rays of the original polyhedron. As before, the amount of memory needed to store the dual representation is the same, however the representation itself is simpler (two matrices versus three matrices). Table 3.1 shows the equivalent forms of inhomogenous and homogenous systems, polyhedra and cones, along with their dual implicit and parametric representations. The table highlights the fundamental relationships between the polyhedron and cone.

Using the homogeneous cone form not only simplifies the data structure used to represent the polyhedron, but also simplifies computation. From practical experience with the implementation of polyhedral operations, it is known that fewer array references have to be done and fewer 'end cases' have to be handled when computing with the homogeneous form. This results in slightly smaller and more efficient procedures.

A mixed system may also be transformed to a *non-mixed* systems of constraints by using the the transformation: $ax = 0 \quad \rightarrow \quad ax \geq 0$ and $ax \leq 0$ along with its dual: a line l can be represented as two rays l and $-l$. This reduces the entire representation to a non-mixed set of homogeneous inequalities (no equalities) and its dual to just an array of rays (no line or vertices). This simplification is tempting, however, it would increase the size of the memory representation (each equality and line require twice the storage). There is another advantage of keeping equalities and inequalities separate: there are different (and much more efficient— polynomial time) methods for solving equalities. Thus, by keeping equalities and inequalities distinct and separate, the memory requirement is kept at a minimum, and equalities can be treated specially using standard, efficient, and well loved methods such as Gauss elimination.

3.2 Dual representation of a polyhedron

A polyhedron may be fully described as either a system of constraints or by its dual form, a collection of rays and lines. Given either form, the other may be computed. However, since the duality computation is an expensive operation (see section 4.2) and since both forms are needed for computation of different operations, a decision to represent polyhedra redundantly using both forms was made. Even though the representation is redundant, keeping both forms in the data structure reduces the number of duality computations that have to be made and improves the efficiency of the polyhedral library. It is a basic memory / execution time tradeoff made in favor of execution time.

3.3 Saturation and the incidence matrix

After being transformed to a homogeneous coordinate system, a polyhedron is represented as a cone (equation 3.1). The dual representations of the cone are:

$$\begin{aligned} \mathcal{C} &= \{x \mid Ax = 0,\ Cx \geq 0\} && \text{(implicit form)} \\ &= \{x \mid x = L\lambda + R\mu,\ \mu \geq 0\} && \text{(parametric form)} \end{aligned}$$

	Inhomogenous System	Homogenous System
Structure	Polydedron $\mathcal{P}$, dimension d	Cone $\mathcal{C}$, dimension $d+1$
Implicit Representation using Hyperplanes	$\mathcal{P} = \{x \| Ax = b,\ Cx \geq d\}$	$\mathcal{C} = \{\hat{x} \mid \hat{A}\hat{x} = 0,\ \hat{C}\hat{x} \geq 0\}$, $\hat{x} = \begin{pmatrix} \xi x \\ \xi \end{pmatrix}$, $\hat{A} = \left(\begin{array}{c\|c} A & -b \end{array}\right)$, $\hat{C} = \left(\begin{array}{c\|c} C & -d \\ \hline 0 & 1 \end{array}\right)$
Parametric Representation using Vertices and Rays	$\mathcal{P} = \{x \mid x = L\lambda + R\mu + V\nu,\ \mu, \nu \geq 0, \sum \nu = 1\}$	$\mathcal{C} = \{\hat{x} \mid \hat{x} = \hat{L}\lambda + \hat{R}\mu,\ \mu \geq 0\}$
Vertices	$v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{pmatrix}, \quad v \in V$	$\hat{r}_v = \begin{pmatrix} \lambda v_1 \\ \lambda v_2 \\ \vdots \\ \lambda v_d \\ \lambda \end{pmatrix},\ \lambda > 0,\ \hat{r}_v \in \hat{R}$
Rays	$r = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_d \end{pmatrix}, \quad r \in R$	$\hat{r}_r = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_d \\ 0 \end{pmatrix}, \quad \hat{r}_r \in \hat{R}$
Lines	$l = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_d \end{pmatrix}, \quad l \in L$	$\hat{l} = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_d \\ 0 \end{pmatrix}, \quad \hat{l} \in \hat{L}$

Table 3.1: Duality between Polyhedra and Cones

Parametric Description	Implicit Description
Lineality Space	System of equalities
Ray Space	System of inequalities
Ray r	Homogeneous Inequality $r^T x \geq 0$
Vertex r/k	Inhomogeneous Inequality $r^T x + k \geq 0$
Line r with Vertex at 0	Homogeneous Equality $r^T x = 0$
Line r with Vertex not at 0	Inhomogeneous Equality $r^T x + k = 0$
Convex union of rays	Intersection of inequalities
Point at origin	Positivity Constraint
Universe Polyhedron	Empty set of Constraints
Empty Polyhedron	Overconstrained system

Table 3.2: Dual Concepts

Substituting the equation for x in the parametric form into the equations involving x in the implicit form, we obtain:

$$\forall(\mu \geq 0,\ \lambda)\ :\ \left\{ \begin{array}{l} AL\lambda + AR\mu = 0 \\ CL\lambda + CR\mu \geq 0 \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} AL = 0,\ AR = 0 \\ CL = 0,\ CR \geq 0 \end{array} \right. \tag{3.2}$$

where rows of A and C are equalities and inequalities, respectively, and where columns of L and R are lines and rays, respectively.

Using the above, we can show the duality of a system of constraints with its corresponding system of lines and rays. Let $\mathcal{C}$ be a cone and $\mathcal{C}*$ be another cone created by reinterpreting the inequalities and equalities of $\mathcal{C}$ as the lines and rays, respectively, of $\mathcal{C}*$. Then the two cones are defined as:

$$\begin{array}{rcl} \mathcal{C} & = & \{x \mid x = L\lambda + R\mu,\ \mu \geq 0\} \\ \mathcal{C}* & = & \{y \mid y = A^T\alpha + C^T\gamma,\ \gamma \geq 0\} \end{array}$$

then the inner product of a point $x \in \mathcal{C}$ and a point $y \in \mathcal{C}*$ is:

$$\begin{array}{rcl} x \circ y & = & y^T x = (A^T\alpha + C^T\gamma)^T \times (L\lambda + R\mu) \\ & = & (\alpha^T A + \gamma^T C) \times (L\lambda + R\mu) \\ & = & \alpha^T(AL\lambda + AR\mu) + \gamma^T(CL\lambda + CR\mu) \\ & \geq & 0 \quad \text{(by application of equation 3.2)} \\ \mathcal{C}* & = & \{y \mid \forall x \in \mathcal{C}\ :\ x \circ y \geq 0\} \end{array}$$

and thus $\mathcal{C}$ and $\mathcal{C}*$ are duals by property 2.7.

Before discussing the incidence matrix, the notion of saturation needs to be defined.

Definition 3.1 *A ray r is said to* **saturate** *an inequality $a^T x \geq 0$ when $a^T r = 0$, it* **verifies** *the inequality when $a^T r > 0$, and it* **does not verify** *the inequality when $a^T r < 0$. Likewise, a ray r is said to* **saturate** *an equality $a^T x = 0$ when $a^T r = 0$, and it* **does not verify** *the equality when*

Operation ↓	Polyhedra	Finite Unions of Polyhedra
Intersection	Closed	Closed
Convex Union	Closed	Closed
Affine Transformation	Closed	Closed
Union	Not Closed	Closed
Difference	Not Closed	Closed

Table 3.3: Closure of Operations

$a^T r \neq 0$. Equalities and inequalities are collectively called **constraints**. *A constraint is* **satisfied** *by a ray if the ray saturates or verifies the constraint.*

The incidence matrix S is a boolean matrix which has a row for every constraint (rows of A and C) and a column for every line or ray (columns of L and R). Each element s_{ij} in S is defined as follows:

$$s_{ij} = \begin{cases} 0, & \text{if constraint } c_i \text{ is saturated by ray(line) } r_j, \text{ i.e. } c_i^T r_j = 0 \\ 1, & \text{otherwise, i.e. } c_i^T r_j > 0 \end{cases}$$

From the demonstrations in equation 3.2 above, we know that all rows of the S matrix associated with equations (A) are 0, and all columns of the S matrix associated with lines (L) are also 0. Only entries associated with inequalities (C) and rays (R) can have 1's as well as 0's. This is illustrated in the following diagram representing the saturation matrix S.

S	L	R
A	(0)	(0)
C	(0)	(0 or 1)

3.4 Expanding the model to unions of polyhedra

Polyhedra are closed under intersection (property 2.1), convex union (convex.hull($A \cup B$), definition 2.14), and affine transformation (property 2.2). However, they are not closed under (simple) union since the union of any two polyhedra is not necessarily convex. Likewise, polyhedra are not closed under the difference operation. To obtain closure of these two operations (union and difference), it is necessary to expand the model from a simple polyhedron to a finite union of polyhedra. The table 3.3 shows the closures of different library operations. The polyhedral library supports the extended model of a union of polyhedra. Thus, all operations in the polyhedral library are closed.

3.5 Data structure for unions of polyhedra

In the previous sections, the motivations for the major design decisions made in defining the data structure have been presented. The data structure should represent polyhedron in the homogeneous cone format (section 3.1), in the redundant form (both constraints and rays represented)(section 3.2), and support the representation of a union of polyhedra (section 3.4). With these objectives in mind, a C-structure for a polyhedron was defined. The term "Ray" , as used in the library, needs some explaination. The term "Ray" is used to represent the vertices, rays, and lines in a polyhedron. Indeed in the homogenous cone form, vertices and rays are both representable as unidirectional rays and line is simply a bidirectional ray. Since no other good term really exists for the ensemble of geometric features of a polyhedron, the term "Ray" is used. The reader needs to differentiate it from a simple ray (definition 2.5) by context. The C-structure for a polyhedron is defined as:

```
typedef struct polyhedron
{ struct polyhedron *next;
  unsigned Dimension, NbConstraints, NbRays, NbEqualities, NbLines;
  int **Constraint;
  int **Ray;
  int *p_Init;
} Polyhedron;
```

The fields of the Polyhedron structure are described as follows:

`Dimension` the dimension of the space in which the inhomogeneous polyhedron resides.

`NbConstraints` the number of equalities (NbEqualities) and inequalities constraining the polyhedron.

`NbRays` the number of lines (NbLines), rays, and vertices in the geometry of the polyhedron.

`NbEqualities` the number of equalities in the constraint list.

`NbLines` the number of lines in the ray list.

`Constraint[i]` the i-th constraint (equation or inequality).

`Ray[i]` the i-th geometric feature (ray, vertex, or line).

`p_Init` for library use to do memory management.

`next` a link to another polyhedron, supporting domains which are finite unions of polyhedra.

The data structure is detailed in figure 3.1. Along with the main structure, three other arrays need to be allocated: an array of constraint pointers, an array of ray pointers, and finally the data array that holds the actual constraints and rays themselves. This entire data structure is created by the library function:

```
Polyhedron *Polyhedron_Alloc ( unsigned Dimension,
                               unsigned NbConstraints,
                               unsigned NbRays )
```

and is replicated by the library function:

```
Polyhedron *Polyhedron_Copy ( Polyhedron *p )
```

and is destroyed (and memory freed) by the library function:

```
void Polyhedron_Free ( Polyhedron *p )
```

Using the `next` pointer field, the several polyhedra whose union form a domain can be put into a single linked list structure. Thus the data structure works equally well for domains as well as for a single polyhedron. Accordingly, the procedures `Polyhedron_Copy` and `Polyhedron_Free` described above have domain equivalents which copy and free an entire linked list of polyhedra.

```
Polyhedron *DomainCopy ( Polyhedron *d )
```

returns a copy of the linked list of polyhedra (domain) pointed to by `d`.

```
void DomainFree ( Polyhedron *d )
```

frees memory allocated to the linked list of polyhedra (domain) pointed to by `d`.

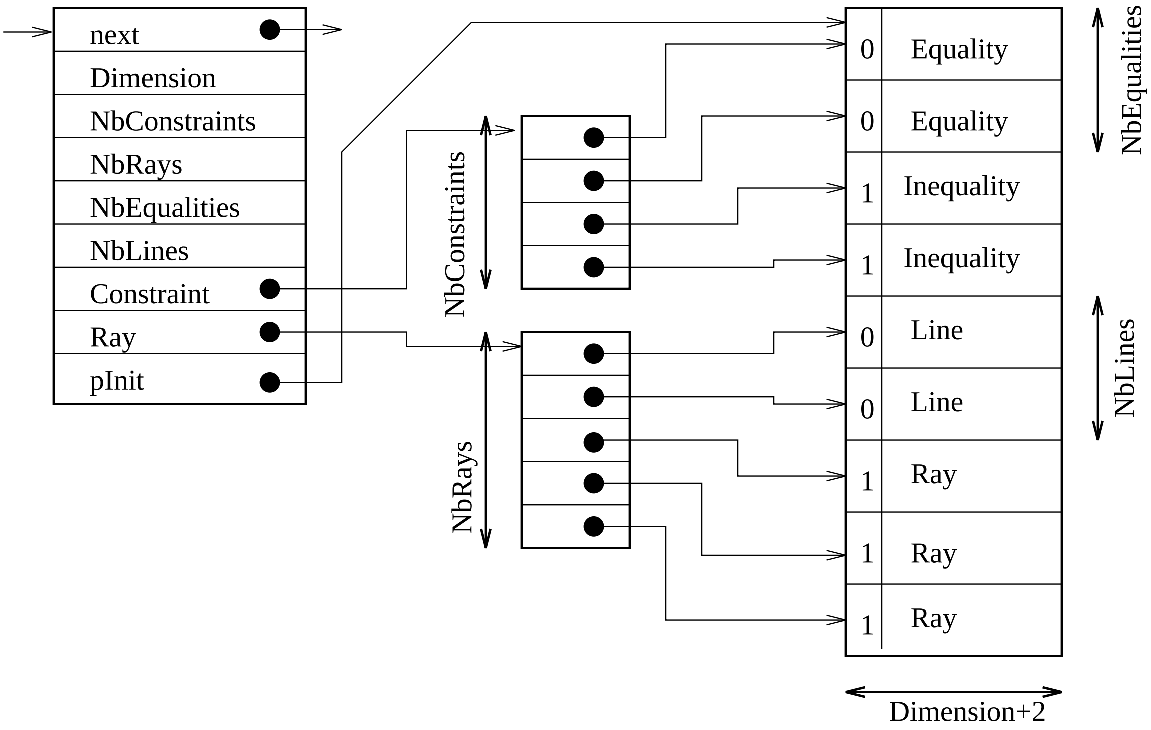


Figure 3.1: Data Structure for Polyhedron

Constraint Format

Each constraint (equality or inequality) consists of a vector of `Dimension+2` elements and has the format:

$(S,\ X_1,\ X_2,\ \cdots,\ X_n,\ K)$ representing the constraint:
if $S = 0$: $X_1 i + X_2 j + ... + X_n k + K = 0$
if $S = 1$: $X_1 i + X_2 j + ... + X_n k + K \geq 0$

which are defined over the n-space with coordinate system $(i,\ j,\ \cdots,\ k)$. The element S is a status word defined to be 0 for equalities and 1 for inequalities.

In an n dimensional system, the i-th constraint $(0 \leq i < NbConstraints)$ is referenced in the following manner:
`Constraint[i][0]` $= S$
`Constraint[i][1]` $= X_1$
`Constraint[i][2]` $= X_2$
$\vdots$
`Constraint[i][Dimension]` $= X_n$
`Constraint[i][Dimension+1]` $= K$

Ray Format

Each ray consists of a vector of `Dimension+2` elements and has the format:

$(S,\ X_1,\ X_2,\ \cdots,\ X_n,\ K)$ representing the geometric object:
if $S = 0$: *line* in direction $(X_1,\ X_2,\ \cdots,\ X_n)$
if $S = 1$: $K \neq 0$: *vertex* $(\frac{X_1}{K},\ \frac{X_2}{K},\ \cdots,\ \frac{X_n}{K})$

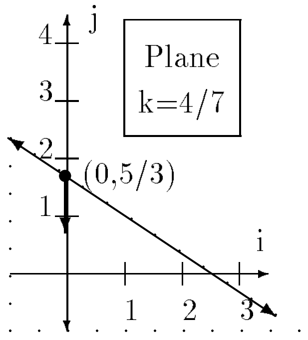
if $S = 1$: $K = 0$: *ray* in direction $(X_1,\ X_2,\ \cdots,\ X_n)$.
The element S is a status word defined to be 0 for lines and 1 for vertices and rays.

In an n dimensional system, the i-th ray ($0 \leq i < NbRays$) is referenced in the following manner:

`Ray[i][0]` $= S$
`Ray[i][1]` $= X_1$
`Ray[i][2]` $= X_2$
$\vdots$
`Ray[i][Dimension]` $= X_n$
`Ray[i][Dimension+1]` $= K$

The example in figure 3.2 shows the internal representation for a polyhedron.

$\{i, j, k \mid 7k = 4;\ 2i + 3j \leq 5\}$




```
Polyhedron
----------
Dimension      = 3
NbConstraints  = 3
NbRays         = 3
NbEqualities   = 1
NbLines        = 1
Constraint[0] = (  0   0   7  -4 )   Equality    7k = 4
Constraint[1] = ( -2  -3   0   5 )   Inequality 2i+3j <= 5
Constraint[2] = (  0   0   0   1 )   Inequality 1 >= 0
Ray[0]        = (  3  -2   0   0 )   Line        (3,-2, 0)
Ray[1]        = (  0  -1   0   0 )   Ray         (0,-1, 0)
Ray[2]        = (  0  35  12  21 )   Vertex      (0, 35/21, 12/21)
                                               = (0, 5/3, 4/7)
```

Figure 3.2: Example 1

3.6 Validity rules

All polyhedra (including empty and universe domains) generated by the polyhedral library satisfy three general rules. In this section, the consistency rules which govern the polyhedral data structure are described.

Given a polyhedron $\mathcal{P} = \mathcal{L} + \mathcal{R} + \mathcal{V}$, the following meanings of the term *dimension* are defined:

1. The **dimension** of a lineality space $\mathcal{L}$ is n where $\mathcal{L}$ is an n-space (see definition 2.12).
2. The **dimension** of the ray space is m where affine.hull($\mathcal{R}+\mathcal{V}$) is an m-flat (see definition 2.13).
3. The **dimension** of the polyhedron $\mathcal{P}$ is p where affine.hull $\mathcal{P}$ is an p-flat.

Property 3.1 *(Dimensionality Rule)*

a. The dimension of the lineality space is the number of irredundant lines.

b. The dimension of the polyhedron is the dimension of the ray space plus the dimension of the lineality space.

c. *The dimension of the ray space is the dimension of the system minus the number of irredundant lines minus the number of irredundant equalities.*

Proof:

Part (a). The dimension of the lineality space is the rank of a set of lines which span it (definition 2.12). The rank is the number of irredundant lines in a basis for the space. Any additional line is necessarily redundant (property 2.3).

Part (b). The dimension of a polyhedron is the dimension of the smallest flat which contains it (definition of dimension). That flat can be partitioned as follows:

$$\begin{aligned}
\text{convex.hull}(\mathcal{P}) &= \text{convex.hull}(\mathcal{L}+\mathcal{R}+\mathcal{V}) \\
&= \text{convex.hull}(\mathcal{L}) + \text{convex.hull}(\mathcal{R}+\mathcal{V}) \\
&= \text{lineality.space}(\mathcal{P}) + \text{convex.hull}(\text{ray.space}(\mathcal{P})) \\
\text{dimension}(\mathcal{P}) &= \text{dimension}(\text{lineality.space}(\mathcal{P})) + \text{dimension}(\text{ray.space}(\mathcal{P}))
\end{aligned}$$

The dimensions of the lineality space and ray space are unique and separable since no irredundant ray is equal to a linear combination of lines (else the ray is redundant) and no line is a linear combination of rays (else the basis of ray space is redundant). Thus, the lineality space and ray space of a polyhedron are dimensionally distinct and the sum of their dimensions is the dimension of the polyhedron.

Part (c). The set of equalities determine the flat in which $\mathcal{P}$ lies. Since each irredundant equality restricts the flat which contains the polyhedron by one dimension, thus

$$\begin{aligned}
\text{dimension}(\mathcal{P}) &= (\text{Dimension of system}) - (\text{Number of equalities}) \\
[\text{from part b.}] &= \text{dimension}(\text{lineality.space}(\mathcal{P})) + \text{dimension}(\text{ray.space}(\mathcal{P}))
\end{aligned}$$

and from part a. we have:

$$\text{dimension}(\text{lineality.space}(\mathcal{P})) = \text{Number of lines}$$

and combining the above three statements:

$$\begin{aligned}
\text{dimension}(\text{ray.space}(\mathcal{P})) &= (\text{Dimension of system}) \\
&\quad -(\text{Number of equalities}) - (\text{Number of lines})
\end{aligned}$$

and thus the dimension of the ray space is the dimension of the system less the number of equalities and less the number of lines.

□

The dimension of the ray space is an important number and is used in the determination of redundant rays and inequalities. It is the key number n used in the saturation rule, property 3.2. It is computed according to part–c of property 3.1, which when written in the library (C-code) is:

```
p->Dimension - p->NbLines - p->NbEqualities
```

Property 3.2 *(Saturation Rule)*
In an n-dimensional ray space,

a. *Every inequality must be saturated by at least n vertices/rays.*

b. *Every vertex must saturate at least n inequalities and a ray must saturate at least $n-1$ inequalities plus the positivity constraint.*

c. *Every equation must be saturated by all lines and vertices/rays.*

d. Every line must saturate all equalities and inequalities.

Proof:

All parts rely on the definition of saturate 3.1.

Part (a). In general, every k-face is the convex union of a minimum of $k+1$ vertices/rays since each k-face lies on a k-flat which is determined by any $k+1$ affinely independent points in the flat (property 2.4) and since vertices/rays are affinely independent (property 2.6), a minimum of $k+1$ of them can be used to determine a k-face. Since each inequality is associated with a $(n-1)$-face (facet) of the polyhedron and each $(n-1)$-face is saturated by $n-1+1=n$ vertices/rays, each inequality is also saturated by at least n vertices/rays.

Part (b). Each vertex is the intersection of at least n facets, and therefore saturates at least n inequalities. Each ray is the intersection of at least $n-1$ facets, and therefore saturates at least $n-1$ inequalities plus the positivity constraint (described in section 3.6.1) which is saturated by all rays (property 3.4).

Part (c) and Part(d). Shown by the derivation of equation 3.2.

□

The independence rule is an invariant of library in which only a minimal representation of a polyhedron is stored.

Property 3.3 *(Independence Rule)*

a. No inequality is a positive combination of any other two inequalities or equalities.

b. No ray is a linear combination of any other two rays or lines.

c. The set of equalities must be linearly independent.

d. The set of lines must be linearly independent.

Proof:

Part (a). Assume $a_r = a_1\alpha + a_2\beta$, with $\alpha \geq 0$, and $\beta \geq 0$. Given the inequalities $a_1^T x \geq 0$, and $a_2^T x \geq 0$, then $(a_1\alpha + a_2\beta)^T \geq 0$ and thus $a_r^T \geq 0$, and a_r is a redundant inequality and may be omitted from the system.

Part (b). By the definition of extreme ray 2.6.

Part (c) and (d). From definitions of flats (2.13) and subspaces (2.12), the dimension attribute is defined in terms of the basis of the lines, and by convention redundant lines and equalities are removed to keep the basis at a minimum. The number of lines and equalities are known and have be discussed in connection with the dimensionality rule (property 3.1).

□

Definition 3.2 *(Redundancy)*
Inequalities that don't satisfy property 3.2.a or property 3.3.a are **redundant**.
Vertices/rays that don't satisfy property 3.2.b or property 3.3.b are **redundant**.

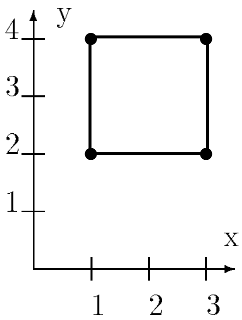
3.6.1 The Positivity Constraint

In the language of algebrists, the trivial constraint $1 \geq 0$ is called the "positivity[1] constraint". When true, you know that positive numbers are positive (a nice thing to know). It was generated as a side

[1] Also called the non-negativity constraint. Here the term *positive* is used in a non-strict way to include zero.

$\{x, y \mid 1 \leq x \leq 3;\ 2 \leq y \leq 4\}$

	x>=1	x<=3	y>=2	y<=4
vertex(1,2)	sat		sat	
vertex(1,4)	sat			sat
vertex(3,2)		sat	sat	
vertex(3,4)		sat		sat

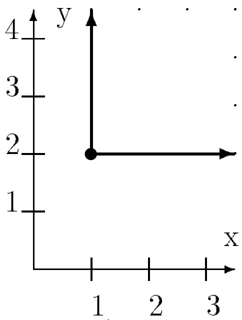


Every constraint saturates two vertices and every vertex saturates two inequalities. This is a perfectly non redundant system.

Figure 3.3: Example 2

$\{x, y \mid x \geq 1;\ y \geq 2\}$

	x>=1	y>=2	1>=0
vertex(1,2)	sat	sat	
ray(1,0)		sat	sat
ray(0,1)	sat		sat



Here, every constraint saturates two vertices/rays and every vertex/ray saturates two inequalities. This is also a non redundant system. However the positivity constraint is also irredundant... it is needed to support the presence of the two rays. Without it, the two rays are not supported and appear mistakenly to be redundant.

Figure 3.4: Example 3

effect of converting from an inhomogenous polyhedron to a homogeneous cone representation as can be seen in equation 3.1. As stated earlier, rays may be thought of as points at infinity. In this vein of thought, the positivity constraint generates the face that connects those points, creating a face at infinity which "closes" unbounded polyhedra. The following property gives the reasoning behind this.

Property 3.4 *All rays are saturated by the positivity constraint and no vertex is saturated by the positivity constraint.*

Proof: In the homogeneous form, the positivity constraint is $\lambda \geq 0$ represented by the vector $a = (0, \cdots, 0, 1)$, and rays are of the form $r = (r_1, \cdots, r_n, 0)$. Since $a \circ r = 0$, for all rays, all rays saturate the positivity constraint. Vertices are of the form $v = (v_1, \cdots, v_n, d)$, $d \neq 0$. Since $a \circ v = d \neq 0$, for all vertices, no vertex saturates the positivity constraint.

□

As surprising as it may seem, the positivity constraint is not always redundant, as was shown in the examples in figures 3.4 and 3.5. The following property gives a rule for when the positivity constraint will be needed.

Property 3.5 *The positivity constraint will be irredundant iff the size of the set of rays is $\geq n$, the dimension of the ray space, and the rank of the ray set is n.*

Proof: For the positivity constraint to be irredundant, it needs at least n vertices/rays which saturate it (property 3.2). Since only rays saturate the positivity constraint, at least n

$\{x \mid x \geq 1\}$

	x>=1	1>=0
line(0,1)	sat	sat
vertex(1,2)	sat	
ray(1,0)		sat

A halfplane.

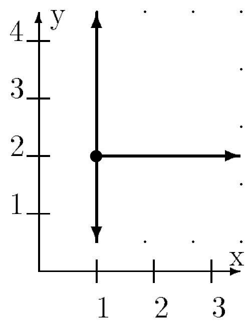


Figure 3.5: Example 4

$\{x, y \mid x = 2;\ y = 3\}$
Polyhedron consisting of a single point (2,3) is dimension 0. The dimension the system is 2, there are two equalities, and the dimension of the lineality space is 0, thus the dimension of the ray space is 2-2-0=0 (property 3.1).

Figure 3.6: Example 5

rays are needed (property 3.4). Thus in a system with n rays, the positivity constraint is irredundant.

□

Positivity constraints are included so there aren't invalid polyhedra floating around (according to properties 3.1 and 3.2). There are different strategies involving the use of this constraint. One strategy is to add the positivity constraint to all polyhedra (even if it is redundant) before doing any operation– and then filter it out of the answer at the end. This works, but may not be very efficient. To add the positivity constraint may require allocating memory and then copying the polyhedron plus the positivity constraint for each polyhedral operand before doing any operation. Another alternative is to keep the positivity constraint in polyhedra where it is needed (according to property 3.5). This works well for the library. The only problem is that it usually will have to be filtered out by the user when displaying the constraints (by a pretty printer).

3.6.2 Empty Polyhedra

An empty domain is a polyhedron which includes no points. It is caused by overcontraining a system such that no point can satisfy all of the constraints. Empty polyhedra have the following properties:

Property 3.6 *In an empty polyhedron*

a. *the dimension of the lineality space is 0.*

b. *the dimension of the ray space is -1.*

c. *there are no rays (vertices, to be more specific).*

Proof:

Part a. Since there are no points in an empty polyhedron, there are no lines, and the dimension of the lineality space is the number of lines = 0.

Part b. To overconstrain a system of dimension n requires $n + 1$ equalities. From property 3.1, the dimension of the ray space is (dimension of system)-(number of equalities)-(number of lines) $= n - (n + 1) - 0 = -1$

Part c. Since there are no points in an empty polyhedron, there are no vertices as well.

$\{x, y \mid 1 = 0\}$

```
Empty Polyhedron, Dimension 2
Constraints  ( 3 equalities, 0 inequalities )
   x = 0
   y = 0
   1 = 0
Lines/Rays ( 0 lines, 0 rays )
   -none-

dim(ray space) = dimension - numlines - numequalities
        = 2   - 0         - 3
        = -1
dim(lineality space) = numlines
        = 0
```

Figure 3.7: Example 6 — Empty Polyhedron

$\{x, y \mid 1 \geq 0\}$

```
Universe Polyhedron, Dimension 2
Constraints  ( 0 equalities, 1 inequality )
   1 >= 0
Lines/Rays ( 2 lines, 0 rays)
   line   (1,0) (x-axis)
   line   (0,1) (y-axis)
   vertex (0,0) (origin)

dim(ray space) = dimension - num_lines - num_equalities
        = 2   - 2         - 0
        = 0
dim(lineality space) = num_lines
        = 2
```

Figure 3.8: Example 7 — Universe Polyhedron

□

A test for an empty polyhedron may be performed by either of the following C-macros:

```
#define emptyQ(P) (P->NbEqualities==(P->Dimension+1))
#define emptyQ(P) (P->NbRays==0)
```

An empty polyhedron can be created by the library by a call to the procedure:

```
Polyhedron *EmptyPolyhedron ( unsigned Dimension )
```

3.6.3 Universe Polyhedron

A universe polyhedron is one that encompasses all points within a certain dimensional subspace. It is therefore unbounded in all directions. It is created by not constraining a system at all (except with the positivity constraint). A universe polyhedron has the following properties:

Property 3.7 *In an universe polyhedron*

a. *the dimension of the lineality space is the dimension of the polyhedron,* *b. the dimension of the ray space is 0,*

c. there are no constraints, other that the positivity constraint.

Proof:

Part a. An unconstrained system of dimension n is a n-space with a basis of n lines.

Part b. From property 3.1, the dimension of the ray space is (dimension of system)-(number of equalities)-(number of lines) $= n - 0 - n = 0$

Part c. Any constraint other that the positivity constraint would exclude points from the system, and is therefore inadmissable.

□

A test for a universe polyhedron may be performed by the following C-macro:

```
#define universeQ(P) (P->Dimension==P->NbLines)
```

A universe polyhedron can be created using the library with a call to the procedure:

```
Polyhedron *UniversePolyhedron ( unsigned Dimension )
```

Chapter 4

THE POLYHEDRAL LIBRARY

The polyhedral library creates, operates on, and frees objects called *domains* (described in section 1.1) made up of unions of polyhedra. The data structure for these domains was described in section 3.5 along with operations to create and free the data structure. This chapter builds on chapter 3 and describes the operational side of the library in detail. The algorithms used to operate on domains are fully described as well.

4.1 Description of Operations

The polyhedral library contains a full set of operations as described in this section.

External interface with library

In many operations there is a parameter called `NbMaxRays` which sets the size of a temporary work area. This work area is allocated by the library using a call to `malloc` at the beginning of an operation and is deallocated at the end. If the work area is not large enough, the operation will fail and a fault flag will be set. The following external domain functions are supported:

```
Polyhedron *DomainIntersection ( Polyhedron *d1, Polyhedron *d2,
                                 unsigned NbMaxRays )
```

returns the domain intersection of domains $d1$ and $d2$. The dimensions of domains $d1$ and $d2$ must be the same. Described in section 4.5.

```
Polyhedron *DomainUnion ( Polyhedron *d1, Polyhedron *d2,
                          unsigned NbMaxRays )
```

returns the domain union of domains $d1$ and $d2$. The dimensions of domains $d1$ and $d2$ must be the same. Described in section 4.6.

```
Polyhedron *DomainDifference ( Polyhedron *d1, Polyhedron *d2,
                               unsigned NbMaxRays )
```

returns the domain difference, $d1$ less $d2$. The dimensions of domains $d1$ and $d2$ must be the same. Described in section 4.7.

```
Polyhedron *DomainSimplify ( Polyhedron *d1, Polyhedron *d2,
                             unsigned NbMaxRays )
```

returns the domain equal to domain $d1$ simplified in the context of $d2$, i.e. all constraints in $d1$ that are not redundant with the constraints of $d2$. The dimensions of domains $d1$ and $d2$ must be the same. Described in section 4.8.

```
Polyhedron *DomainConvex ( Polyhedron *d, unsigned NbMaxRays )
```

returns the minimum polyhedron which encloses domain d. Described in section 4.9.

```
Polyhedron *DomainImage ( Polyhedron *d, Matrix *m,
                          unsigned NbMaxRays )
```

returns the image of domain d under affine transformation matrix m. The number of rows of matrix m must be equal to the dimension of the polyhedron plus one. Described in section 4.10.

```
Polyhedron *DomainPreimage ( Polyhedron *d, Matrix *m,
                             unsigned NbMaxRays )
```

returns the preimage of domain d under affine transformation matrix m. The number of columns of matrix m must be equal to the dimension of the polyhedron plus one. Described in section 4.11.

```
Polyhedron *Constraints2Polyhedron ( Matrix *m,
                                     unsigned NbMaxRays )
```

returns the largest polyhedron which satisfies all of the constraints in matrix m. Described in section 4.4.

```
Polyhedron *Rays2Polyhedron ( Matrix *m, unsigned NbMaxRays )
```

returns the smallest polyhedron which includes all of the vertices, rays, and lines in matrix m. Described in section 4.4.

```
Polyhedron *UniversePolyhedron ( unsigned Dimension )
```

return the universal polyhedron of dimension n. Described in section 3.6.3.

```
Polyhedron *EmptyPolyhedron ( unsigned Dimension )
```

return the empty polyhedron of dimension n. Described in section 3.6.2.

```
Polyhedron *DomainCopy ( Polyhedron *d )
```

returns a copy of domain d. Described in section 3.5.

```
void DomainFree ( Polyhedron *d )
```

frees the memory used for domain d. Described in section 3.5.

4.2 Computation of Dual Forms

An important problem in computing with polyhedral domains is being able to convert from a domain described implicitly in terms of linear equalities and inequalities (equation 2.1), to a parametric description (equation 2.2) given in terms of the geometric features of the polyhedron (lines, rays, and vertices). Inequalities and equalities are referred to collectively as *constraints*. An equivalent problem is called the *convex hull problem* which computes the facets of the convex hull surrounding given a set of points,

The algorithms to solve this problem are categorized into one of two general classes of algorithms, the pivoting and non-pivoting methods. [MR80] The pivoting methods are derivatives of the simplex method which finds new vertices located adjacent to known vertices using simplex pivot operations.

The algorithm used by the library belongs to the other class, that of the of nonpivoting methods. These methods find the dual by first setting up a tableau in which an initial polyhedron (such as the universe or the positive orthant) is simultaneously represented in both forms. The algorithm then iterates by adding one new inequality or equality at a time and computing the new polyhedron at each step by modifying the polyhedron from the previous step. The order in which constraints are selected does not change the final solution, but may have an effect on the run time of the procedure as a whole. The complexity of this problem is known to be $\mathcal{O}(n^{\lfloor \frac{d}{2} \rfloor})$, where n is the number of constraints and d is the dimension. This is the best that can be done, since the size of the output (i.e. the number of rays) is of the same order.

The nonpivoting methods are based on an algorithm called *the double description method* invented by Motzkin et al. in 1953 [MRTT53]. Motzkin described a general algorithm which iteratively solves the dual-computation problem for a cone. (Since polyhedra may be converted to cones, it works for all polyhedra.) In each iteration, a new constraint is added to the current cone in the tableau. Rays in the cone are divided into three groups, R^+ the rays which verify the constraint, R^0 the rays which saturate the constraint, and R^- the rays which do not verify the constraint. A new cone is then constructed from the ray sets R^+, R^0, plus the convex combinations of pairs of rays, one each from sets R^+ and R^-. The main problem with the nonpivoting methods is that they can generate a non-minimal set of rays by creating non-extreme or redundant rays. If allowed to stay, the number of rays would grow exponentially and would seriously test the memory capacity of the hardware as well as degrade the performance of the procedure. Motzkin proposed a simple and rather elegant test to solve this problem. He showed that a convex combination of a pair of rays ($r^- \in R^-$, $r^+ \in R^+$) will result in an extreme ray in the new cone if and only if the minimum face which contains them both: 1) is dimension one greater than r^- and r^+, and 2) only contains the two rays r^- and r^+. This test inhibits the production of unwanted rays and keeps the solution in a minimal form.

Chernikova [Che65, Rub75] described a similar algorithm to solve the restricted case of the mixed constraint problem with the additional constraint that variables are all non-negative ($x \geq 0$). Chernikova's method was the same as Motzkin's method, except that she used a slightly smaller and improved tableau. Fernández and Quinton [FQ88] extended the Chernikova method by removing the restriction that $x \geq 0$ and adding a heuristic to improve speed by ordering the constraints. A large portion of the computation time is spent doing the adjacency test. Le Verge [Le 92] improved the speed of the redundancy checking procedure used in [FQ88], which is the most time consuming part of the algorithm. Seidel described an algorithm for the equivalent convex hull problem [Sei91] which executes in $\mathcal{O}(n^{\lfloor \frac{d}{2} \rfloor})$ expected running time where n is the number of points and d is the dimension. This is provably the best one can do, since the output of the procedure is of the same order. He solves the adjacent ray problem (the adjacent facet problem in his case) by creating and maintaining a facet graph in which facets are vertices and adjacent facets are connected by edges. It takes a little extra code to maintain the graph, but then he does not need to do the Motzkin adjacency test on all pairs of vertices (facets).

McMullen [McM70, MS71] showed that for any d-polytope with n vertices, the number of k-faces, f_k is upper bounded by the number of k-faces of a cyclic d-polytope with the same number of vertices. One of the implications of this is that the number of facets, $f_{d-1} = \mathcal{O}(n^{\lfloor \frac{d}{2} \rfloor})$.

4.2.1 The Motzkin algorithm

The nonpivoting solvers successively refine their solution by adding one constraint at a time and modifying the solution polyhedron from the previous step to reflect the new constraint. An inequality $a^T x \geq 0$ is co-represented by the closed halfspace H^+ which is the set of points $\{x \ : \ a^T x \geq 0\}$. Likewise the equality $a^T x = 0$ is co-represented by the hyperplane H which is the set of points $\{x \ : \ a^T x = 0\}$. At each step of the algorithm, a new inequality or equality (represented by either H^+ or H, respectively) is introduced into the system. The polyhedron $\mathcal{P} = \mathcal{L} + \mathcal{R}$ (the combination of its lineality space and ray space) is constrained by the new constraint by intersecting $\mathcal{P}$ with either H^+ or H to produce a modified polyhedron $\mathcal{P}' = \mathcal{L}' + \mathcal{R}'$.

The algorithm **Dual** in figure 4.1 gives the algorithm given by Motzkin to find the dual of a set of constraints A. In **Dual**, there are three procedures which alter the polyhedron. They are **ConstrainL** which constrains the lineality space, **AugmentR** which augments the dimension of the ray space, and finally **ConstrainR** which constrains the ray space. These procedures are discussed below in greater detail.

The **ConstrainL** procedure shown in figure 4.3 constrains the lineality space L by slicing it with a new constraint, and if the new constraint cuts L, then L's dimension is reduced by one and a new ray r_{new} is generated which is added to the ray space. It is fairly straightforward and runs in $\mathcal{O}(n)$ time where n is the dimension of the lineality space.

There are two procedures which perform transformations on the ray space. The first one is **AugmentR** shown in figure 4.4 which adds a new ray r_{new} created by **ConstrainL** to the ray space. When r_{new} is added to the ray space R, it increases the dimension of R by one. It is of complexity $\mathcal{O}(r)$ time, where r is the number of rays.

The second operation **ConstrainR** shown in figure 4.5 constrains the ray space by slicing it with the hyperplane H and discarding the part of the ray space which lies outside of constraint. For inequalities, the part of the polyhedron which lies outside of the halfspace H^+ is removed. For equalities, the part of the polyhedron which lies outside of the hyperplane H is removed. In either case, the new face lying on the cutting hyperplane surface is computed. **ConstrainR** computes a new pointed cone by adding a new constraint. Rays which verify and saturate the constraint are added. Rays which do not verify the constraint are combined with *adjacent* rays which verify the constraint to create new rays which saturate the constraint. Motzkins adjacency test is used to find adjacent pairs of rays. The Motzkin adjacency test is used to test every pair of rays to determine if that pair will combine to produce an extreme ray or not. This is done by computing what constraints the pair of rays have in common and making sure that no other ray also saturates that same set of constraints. Thus the list of rays produced by **ConstrainR** is always extreme (non-redundant). The entire **ConstrainR** procedure has an $\mathcal{O}(n^3 k)$ complexity where n is the number of rays and k is the number of constraints. Much of this time is spent in performing the adjacency tests.

The procedure **Combine** shown in figure 4.2 is where all of the actual computation takes place. It uses as input two rays, r^+ and r^-, as well as a constraint a. It then computes the ray $r^=$ which firstly is a linear combination of r^+ and r^- ($r^= = \lambda_1 r^+ + \lambda_2 r^-$), and secondly, saturates constraint a, ($a^T r^= = 0$).

Procedure **Dual(A)**, returns L, R

$L :=$ basis for d-dimensional lineality space.
$R :=$ point at the origin.
For each constraint $a \in A$ Do
    $r_{new} :=$ **ConstrainL** $(L,\ a)$
    If $r_{new} \neq 0$ Then **AugmentR** $(R,\ a,\ r_{new})$ Else **ConstrainR** $(R,\ a)$
End
Return L and R.

Figure 4.1: Procedure to compute **Dual(A)**

Procedure **Combine(r_1, r_2, a)**, returns r_3

$D = \mathrm{GCD}(a^T r_1, a^T r_2)$
$\lambda_1 = a^T r_2 / D$
$\lambda_2 = -a^T r_1 / D$
$r_3 = \lambda_1 r_1 + \lambda_2 r_2$

Figure 4.2: Procedure to compute **Combine(r_1, r_2, a)**

Procedure **ConstrainL** (L, a), modifies L, returns r_{new}

Find an $l_1 \in L$ such that $a^T l_1 \neq 0$, (l_1 does not saturate constraint a)
If l_1 does not exist Then ($L \cap H$ is L itself and r_{new} is empty) Return 0.
$L' :=$ empty.
For each line $l_2 \in L$ such that $l_2 \neq l_1$ Do
    $L' := L' +$ **Combine** $(l_1,\ l_2,\ a)$
End
If $a^T l_1 > 0$, (l_1 verifies constraint a) Then Create ray r_{new} equal to l_1
Else ($a^T l_1 < 0$) Create ray r_{new} equal to $-l_1$
$L := L'$
Return r_{new}

Figure 4.3: Procedure to compute **ConstrainL(L, a)**

Procedure **AugmentR** (R, a, r_{new}), modifies R

Set $R' :=$ empty.
For each ray $r \in R$ do
    If $a^T r = 0$ Then $R' := R' + r$
    If $a^T r > 0$ Then $R' := R' +$ **Combine** $(r,\ -r_{new},\ a)$
    If $a^T r < 0$ Then $R' := R' +$ **Combine** $(r,\ r_{new},\ a)$
End
If a is an inequality Then $R' := R' + r_{new}$
$R := R'$

Figure 4.4: Procedure to compute **AugmentR(R, a, r_{new})**

Procedure **ConstrainR** (R, a), modifies R

Partition $R = R^+ + R^0 + R^-$.
    $R^0 := \{r \mid r \in R, a^T r = 0\}$, the rays which saturate constraint a.
    $R^+ := \{r \mid r \in R, a^T r > 0\}$, the rays which verify constraint a.
    $R^- := \{r \mid r \in R, a^T r < 0\}$, the rays which do not verify constraint a.
If constraint a is an inequality, Then set $R' := R^+ + R^0$.
Else (constraint a is an equality) set $R' := R^0$.
For each ray $r^+ \in R^+$ do
    For each ray $r^- \in R^-$ do
        Adjacency test on (r^+, r^-)
        c := set of common constraints saturated by both (r^+, r^-)
        For each ray $r \in R \mid r \neq r^+, \quad r \neq r^-$ Do
            If r also saturates all of the contraints in set c Then
                (r^+ and r^- are not adjacent.) Continue to next ray r^-.
        End
        (r^+ and r^- are adjacent.) $R' := R' +$ **Combine** (r^+, r^-, a)
    End
End
$R := R'$

Figure 4.5: Procedure to compute **ConstrainR(**R, a**)**

4.2.2 Implementation

The procedure **Dual** is implemented in the polyhedral library as the procedure:

```
static int Chernikova ( Matrix *Constraints,
                        Matrix *Rays,
                        Matrix *Sat,
                        unsigned NbLines,
                        unsigned NbMaxRays,
                        unsigned FirstConstraint )
```

It is named “Chernikova” for historical reasons, however, a more suitable name would have been “Motzkin”, since the procedure is primarily due to him. It is somewhat different than the basic one described in section 4.2.1 in that it allows a new set of constraints to be added to an already existing polyhedron (there may be preexisting constraints and rays). The entire list of constraints (both the old and the new) is passed as the parameter `Matrix *Constraints`, with the parameter `unsigned FirstConstraint` indicating which is the first “new” constraint. The preexisting lines and rays are passed in as the parameter `Matrix *Rays`. The lines must be grouped together at the beginning of the matrix, and the parameter `unsigned NbLines` indicates how many lines are in the `Rays` matrix. The parameter `unsigned NbMaxRays` is the allocated dimension of the `Rays` matrix and limits the number of lines and rays that can be stored at any one time. And finally, the parameter `Matrix *Sat` contains the incidence matrix (defined in section 3.3) between the original constraints and rays of the input polyhedron. The procedure adds the new constraints to the polyhedron, updating the `Rays` and `Sat` matrices. The updated matrices are returned by the procedure. This procedure has been benchmarked and times published in [Le 92].

4.3 Producing a minimal representation

After computing the dual of a set of constraints, the set of rays produced is guaranteed to be non-redundant, by virtue of the adjacency test which is done when each ray was produced. However, the constraints are still possibly redundant. There remain a number of simplifications which can still be done on the resulting polyhedron, among which are:

1. Detection of implicit lines such as line(1,2) given that there exist rays (1,2) and (-1,-2).
2. Finding a reduced basis for the lines.
3. Removing redundant positivity constraints $1 \geq 0$.
4. Detection of trivial redundant inequalities such as $y \geq 4$ given $y \geq 3$, or $x \geq 2$ given $x = 1$.
5. Detection of redundant inequalities such as $x + y \geq 5$ given $x \geq 3$ and $y \geq 2$.
6. Solving the system of equalities and eliminating as many variables as possible.

The algorithm to do all of these reductions is sketched out below. In the procedure, each constraint and each ray needs a status word which is provided for in the polyhedron structure (see section 3.5).

Reduce(Constraints, Rays, Sat**)**, returns a Polyhedron structure.

Step 0 Count the number of vertices among the rays while initializing the ray status counts to 0. If no vertices are found, quit the prodedure and return an empty polyhedron as the result.

Step 1 Compute status counts for both rays and inequalities. For each constraint, count the number of vertices/rays saturated by that constraint, and put the result in the status words. At the same time, for each vertex/ray, count the number of constaints saturated by it.
Delete any positivity constraints, but give rays credit in their status counts for saturating the positivity constraint.

Step 2 Sort equalities out from among the constraints, leaving only inequalities. Equalities are constraints which saturate all of the rays. (Status count = number of rays)

Step 3 Perform gaussian elimination on the list of equalities. Obtain a minimal basis by solving for as many variables as possible. Use this solution to reduce the inequalities by elimating as many variables as possible. Set NbEq2 to the rank of the system of equalities.

Step 4 Sort lines out from among the rays, leaving only unidirectional rays. Lines are rays which saturate all of the constraints (status count = number of constraints + 1(for positivity constraint)).

Step 5 Perform gaussian elimination of on the lineality space to obtain a minimal basis of lines.
Use this basis to reduce the representation of the unidirectional rays. Set NbBid2 to the rank of the system of lines.

Step 6 Do a first pass filter of inequalities and equality identification.
New positivity constraints may have been created by step 3. Check for and elimate them.
Count the irredundant inequalities and store count in NbIneq.
if (Status==0) Constraint is redundant.
else if (Status<Dim) Constraint is redundant.
else if (Status==NbRays) Constraint is an equality.
else Constraint is a irredundant inequality.

Step 7 Do first pass filter of rays and identification of lines.
Count the irredundant Rays and store count in NbUni.
if (Status<Dim) Ray is redundant.
else if (Status==(NbConstraints+1)) Ray is a line.
else Ray is an irredundant unidirectional ray.

Step 8 Create the polyhedron. Allocate the polyhedron (using approximate sizes).
Number of constraints = NbIneq+NbEq2+1
Number of rays = NbUni+NbBid2
Partially fill the Polyhedron structure with the lines computed in step 3 and the equalities computed in step 5.

Step 9 Final pass filter of inequalities.
Every 'good' inequality must saturate at least Dimension rays and be unique.
The final list of inequalities is written to polyhedron.

Step 10 Final pass filter of rays and detection of redundants rays.
The final list of rays is written to polyhedron.

Step 11 Return polyhedron.

In the polyhedral library, the **Reduce** algorithm described above is implemented as an internal library procedure defined as:

```
static Polyhedron *Remove_Redundants ( Matrix *Constraints,
                                       Matrix *Rays,
                                       Matrix *Sat,
                                       unsigned *Filter )
```

It takes the list of constraints and rays as generated by the `Chernikova` procedure, as well as the incidence matrix `Sat` relating the two. It assumes that either the unidirectional rays are non redundant, *or* that the inequalities are non redundant. This is guaranteed by the `Chernikova` procedure. The procedure performs the reductions on the lists of constraints and rays, then builds a `Polyhedron` structure from the results. The parameter `unsigned *Filter`, if non-zero, points to a bit vector with one bit for each ray. The `Remove_Redundants` procedure sets the bits corresponding to the rays which it finds non-redundant. The `Filter` vector is used in the implementation of the `DomSimplify` function.

4.4 Conversion of rays/constraints to polyhedron

Given a set of rays, the corresponding polyhedron is computed by simply running the `Chernikova` procedure to get the dual list of constraints and then the `Remove_Redundants` procedure to reduce the ray/constraint lists and create a polyhedron.

Likewise, starting from a list of constraints, the corresponding polyhedron is computed by running the `Chernikova` procedure to get the dual list of rays, and then the `Remove_Redundants` procedure to reduce the ray/constraint lists and create a polyhedron. The conversion of a list of rays or a list of contraints to a polyhedron is the most basic application of the `Chernikova` and `Remove_Redundants` procedures.

4.5 Intersection

Intersection is performed by concatenating the lists of constraints from two (or more) polyhedra into one list, and finding the polyhedron which satisfies all of the combined constraints. This is done by finding the extremal rays which satisfy the combined constraints, (finding the dual of the list of constraints), and then reducing both the constraints and rays into one polyhedron. This procedure is illustrated in figure 4.6.

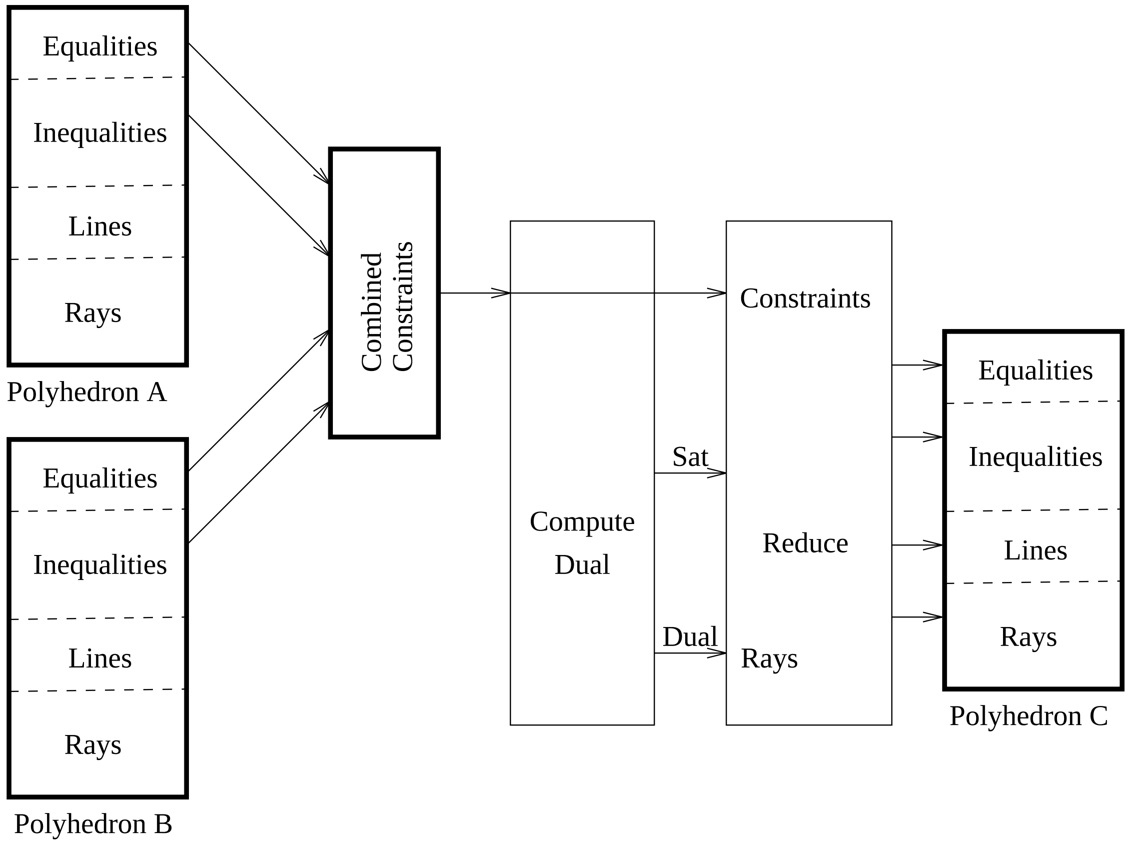


Figure 4.6: Computation of Intersection

To intersect two domains, A and B, which are unions of polyhedra, $A = \cup_i A_i$ and $B = \cup_j B_j$, the pairwise intersection of the component polyhedra from A and B must be computed, and the union of the results is the resulting domain of intersection, as shown below:

$$\begin{aligned} A \cap B &= (\cup_i A_i) \cap (\cup_j B_j) \\ &= \cup_{i,j}(A_i \cap B_j) \end{aligned}$$

4.6 Union

The domain (non-convex) union operation simply combines two domains into one. The lists of polyhedra associated with the domains are combined into a single list. However, combining the two lists blindly may create non-minimal representations. For instance, if in forming the union of domains $A = \{i \mid i \geq 1\}$ and $B = \{i \mid i \geq 2\}$, the fact that $A \supset B$ is taken into consideration, then the union can be reduced to simply A. The algorithm used in the library performs this kind of simplification during the union operation. Before adding any new polyhedron to an existing list of polyhedra, it first checks to see if that polyhedron is covered by some polyhedron already in the domain. If it *is* covered, then the new polyhedron is not added to the domain. Likewise, polyhedra in the existing list may be deleted if they are covered by the new polyhedron. In the new combined list, no polyhedron is a subset of any other polyhedron.

The test for when one polyhedron covers another is performed by the library procedure:

```
int  PolyhedronIncludes(p1, p2)
     Polyhedron *p1;
     Polyhedron *p2;
```

which returns a 1 if $p1 \supseteq p2$, (polyhedron `p1` includes (covers) polyhedron `p2`), and returns a 0 otherwise. The test for when a polyhedron $p1$ covers or includes another polyhedron $p2$ is straight forward using the dual representation of polyhedra in the library: $p1 \supseteq p2$ if all of the rays of $p2$ *satisfy* (see definition 3.1) all of the constraints of $p1$. This is a case in point of when the dual representation comes in handy. The constraint representation of $p1$ and the dual ray representation of $p2$ are used to determine $p1 \supseteq p2$. Since both representations are kept in the data structure, the dual does not need to be (re)computed in order to do this test.

4.7 Difference

Domain difference $A - B$ computes the domain which is part of A but not part of B. It is equivalent to $A \cap \sim B$, where $\sim B$ is the *complement* domain of B. If B is the intersection of a set of hyperplanes (representing the equalities) and halfspaces (representing the inequalities), then the inverse of B is computed as follows:

$$
\begin{array}{rcl}
\sim B & = & \sim (\cap_i H_i) \\
 & = & \cup_i (\sim H_i) \\
 & & \text{where} \\
\sim H_i & = & \left\{ \begin{array}{ll} \{x \mid a^T x < 0\} & \text{when} \quad H_i = \{x \mid a^T x \geq 0\} \\ \{x \mid (a^T x < 0 \ \cup \ a^T x > 0)\} & \text{when} \quad H_i = \{x \mid a^T x = 0\} \end{array} \right. \\
 & & \text{and normalizing for integer lattice domains :} \\
 & = & \left\{ \begin{array}{ll} \{x \mid -a^T x + 1 \geq 0\} & \text{when} \quad H_i = \{x \mid a^T x \geq 0\} \\ \{x \mid (-a^T x + 1 \geq 0 \ \cup \ a^T x - 1 \geq 0)\} & \text{when} \quad H_i = \{x \mid a^T x = 0\} \end{array} \right. .
\end{array}
$$

The computation of difference is the same as the computation of intersection after taking the inverse of B. Since the inverse of B is a union of polyhedra, the difference of two polyhedra can be a union of polyhedra. Thus, polyhedra are not closed under the operation difference, where as unions of polyhedra are closed under this operation.

4.8 Simplify

The operation *simplify* is defined as follows:

> Given domains A and B, then Simplify$(A, B) = C$, when $C \cap B = A \cap B$, $C \supseteq A$ and there does not exist any other domain $C' \supset C$ such that $C' \cap B = A \cap B$.

The domain B is called the *context*. The simplify operation therefore finds the largest domain set (or smallest list of constraints) that, when intersected with the context B is equal to $A \cap B$. This operation is used in Alpha to simplify case statements, as shown in the example in figure 4.7.

The simplify operation is done by computing the intersection $A \cap B$ and while doing the `Remove_Redundants` procedure, recording which constraints of A are "redundant" with the intersection. The result of the simplify operation is then the domain A with the "redundant" constraints removed.

The ALPHA program fragment:

```
var  A:{t,p | 0<=t; 1<=p<=4};
A = case
      {t,p | t=0; 1<=p<=4} ... ;
      {t,p | t>0; 1<=p<=3} ... ;
      {t,p | t>0; p=4}     ... ;
    esac;
```

is operationally equivalent to the following fragment:

```
var  A:{t,p | 0<=t; 1<=p<=4};
A = case
      {t,p | t=0}       ... ;
      {t,p | t>0; p<=3} ... ;
      {t,p | t>0; p=4}  ... ;
    esac;
```

which has simplified case conditions on the domain A. The above simplifications can be found using the simplify operation using the domain of A as the context, and simplifying the case condition domains.

Figure 4.7: Application of DomSimplify

An interesting subproblem in the simplify operation occurs when the intersection of A and its context B are empty. In this case, simplify should find the minimal set of constraints of A which contradict all of the constraints of B. This is believed to be an NP-hard problem and a heuristic is employed to solve it in the library.

4.9 Convex Union

Convex union is performed by concatenating the lists of rays and lines of the two (or more) polyhedra in a domain into one combined list, and finding the set of constraints which tightly bound all of those objects. This is done by finding the dual of the list of rays and lines, and then reducing both the constraints and rays into one polyhedron. This procedure is illustrated in figure 4.8. This procedure is very similar to the intersection procedure which has already be described in section 4.5. Convex union finds the polyhedron generated from the union of the lines and rays of the two input polyhedra. Intersection finds the polyhedron generated from the union of the equalities and inequalities of the inputs.

4.10 Image

The function *image* transforms a domain $\mathcal{D}$ into another domain $\mathcal{D}'$ according to a given affine mapping function, $Tx + t$ (see definition 2.9 and property 2.2). The resulting domain $\mathcal{D}'$ is defined as:

$$\mathcal{D}' = \{x' \mid x' = Tx + t,\ x \in \mathcal{D}\}$$

In homogeneous terms, the transformation is expressed as

$$\mathcal{C}' = \{\begin{pmatrix} \xi x' \\ \xi \end{pmatrix} \mid \begin{pmatrix} \xi x' \\ \xi \end{pmatrix} = \left(\begin{array}{c|c} T & t \\ \hline 0 & 1 \end{array}\right) \begin{pmatrix} \xi x \\ \xi \end{pmatrix},\ \begin{pmatrix} \xi x \\ \xi \end{pmatrix} \in \mathcal{C}\}$$

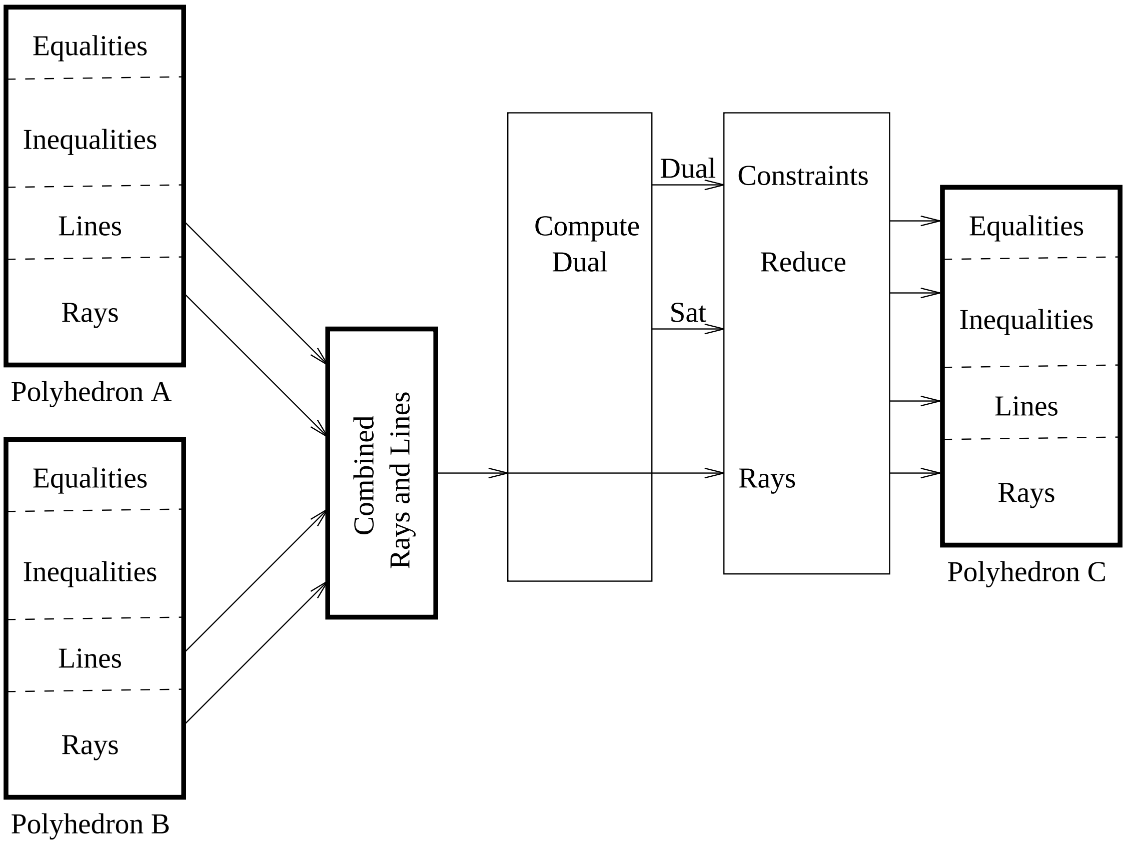


Figure 4.8: Computation of Convex Union

Thus in the homogeneous representation, an affine transfer function becomes a linear transfer function (no constant added in). In the analysis that follows, we will treat the transfer function from the linear point of view. The transformation function $\left(\begin{array}{c|c} T & t \\ \hline 0 & 1 \end{array}\right)$ is a matrix dimensioned by $(n+1) \times (m+1)$, where n and m are the dimensions of x and x', respectively. This transformation matrix is passed as a parameter to the image procedure. If $n = m$, x and x' are the same dimension. If $n \neq m$, the transformed space is of a larger (or smaller) dimension. The transformation does not have to be one-to-one, and therefore may not be invertable. Also, if $\det T \neq 1$, then the volume of the domain (the number of points in the domain) will be scaled by the determinant. To compute an image of D, given the full redundant representation:

$$D = \{x \mid Ax \geq 0,\ x = \mu R,\ AR \geq 0,\ \mu \geq 0\}$$

and given the transformation $x' = Tx$, the result D' is

$$\begin{aligned} D' &= \{x' \mid A'x' \geq 0,\ x' = TR\mu,\ A'TR \geq 0,\ \mu \geq 0\} \\ &= \{x' \mid A'x' \geq 0,\ x' = R'\mu,\ A'R' \geq 0,\ \mu \geq 0\} \end{aligned}$$

A' can be computed as the dual of R'. Thus, $R' = TR$ and $A' = \text{dual}(R')$. This computation is illustrated in figure 4.10 The image of a domain is simply the union of the images of the component

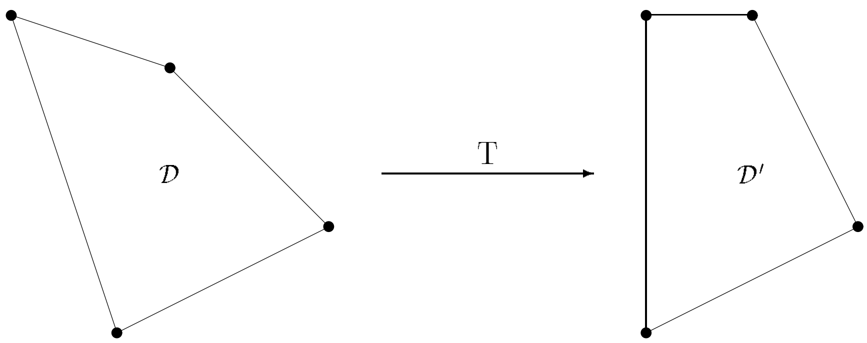


Figure 4.9: Affine transformation of $\mathcal{D}$ to $\mathcal{D}'$

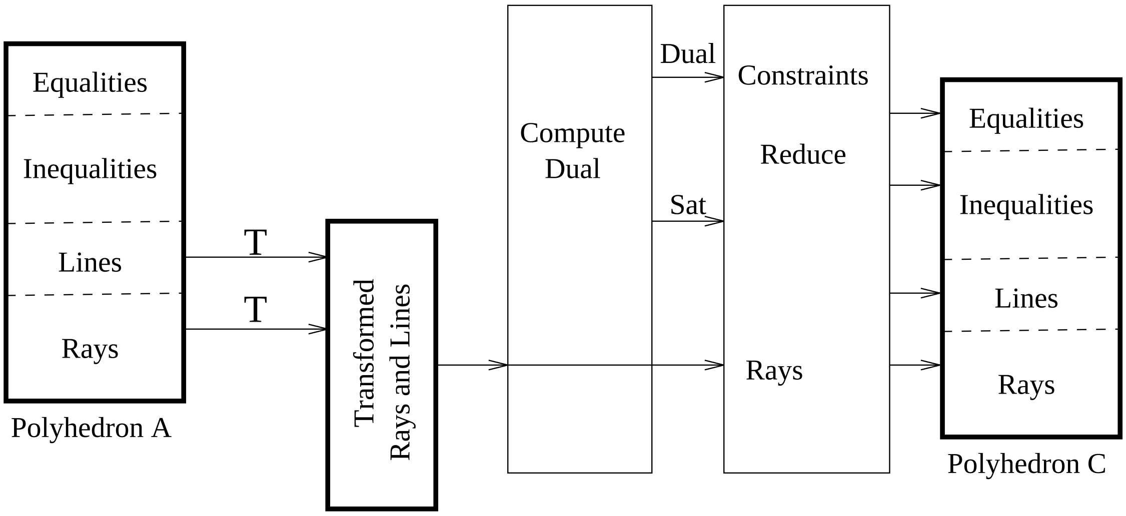


Figure 4.10: Computation of Image

polyhedra contained in the domain, as follows:

$$\begin{aligned} T.\mathcal{D} &= T.(\cup_i \mathcal{P}_i) \\ &= \cup_i (T.\mathcal{P}_i) \end{aligned}$$

4.11 Preimage

Preimage is the inverse operation of image. That is given a domain D' defined as

$$D' = \{x' \mid A'x' \geq 0,\ x' = R'\mu,\ A'R' \geq 0,\ \mu \geq 0\}$$

and a transformation T, find the domain D which when transformed by T gives D'. The relation $x = Tx'$ still holds. (Refer again to figure 4.9.) The result D is

$$\begin{aligned} D &= \{x \mid A'Tx \geq 0,\ x = R\mu,\ A'TR \geq 0,\ \mu \geq 0\} \\ &= \{x \mid Ax \geq 0,\ x = R\mu,\ AR \geq 0,\ \mu \geq 0\} \end{aligned}$$

In the result, $A = A'T$ and $R = \text{dual}(A)$. This procedure is illustrated in figure 4.11. The preimage

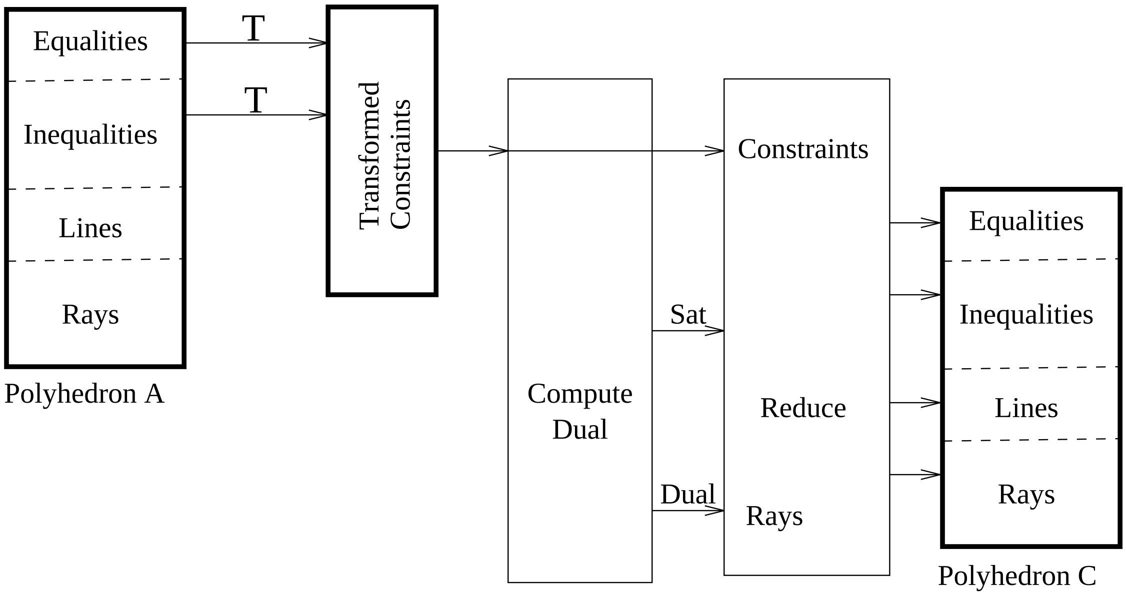


Figure 4.11: Computation of Preimage

of a domain is simply the union of the preimages of the component polyhedra contained in the domain, as follows:

$$\begin{aligned} T^{-1}.\mathcal{D} &= T^{-1}.(\cup_i \mathcal{P}_i) \\ &= \cup_i (T^{-1}.\mathcal{P}_i) \end{aligned}$$

T^{-1} here is simple a notation for the preimage operation and does not mean to imply that T is invertable.

4.12 The implementation of the polyhedral library

The polyhedral library has been implemented in the C-language and is currently in use in the Alpha environment, as well as at other sites. The library code is composed of five files: three headers and two code files as follows:

- `types.h` The header file defining the data structures used in the library.
- `vector.h` The header file containing the forward definitions of the external vector operation procedures.
- `vector.c` The code file containing the code for vector operations.
- `polyhedron.h` The header file containing the forward definitions of the external vector polyhedral and domain procedures.
- `polyhedron.c` The code file containing the code for polyhedral and domain operations.

The code itself is too large to be included as part of this report, but is available by ftp at host `ftp.irisa.fr` in the directory `/local/API/`. It has been nicknamed the Chernikova library by its users.

Chapter 5

CONCLUSION

The polyhedral library described in this report implements basic geometrical operations on stuctures called domains which consist of finite unions of convex polyhedra [Wil93]. The domains are described in a dual representation consisting of:

1. a set of constraints— inequalities and equalities, and
2. a set of geometric features— lines, rays, and vertices.

It is convenient to represent polyhedra in their pointed cone form, which results from the transformation $x \longrightarrow \begin{pmatrix} \xi x \\ \xi \end{pmatrix}$. This transformation maps inhomogeneous constraints to homogeneous constraints and maps both the vertices and rays of a polyhedron to rays in a pointed cone. The pointed cone is thus stored as a list of rays, and the lineality space as a list of basis lines.

Domains can be created starting with a list of constraints or a list of geometric features. Given one representation, the other is computed by the procedure **Dual** which has been described in section 4.2.1. The library is written in the C–language and may be linked in with an application to provide capability for doing geometric operations, such as union, intersection, difference, and simplification, on domains.

The library was originally written to support the ALPHA environment in which all variables are based on domains. Subsequently, the library was placed in the public domain (`ftp.irisa.fr`, directory `/local/API/`).

In the ALPHA environment, the library is used extensively in carrying out program transformations and in doing static analysis of ALPHA programs. To support ALPHA development, the library has been interfaced to the MATHEMATICA symbolic mathematics package.

The library is easy to interface to and use. Appendix A contains an example C–program which demonstrates the use of the library. This sample program was written to show how a C–program may be interfaced with the library.

The library code is composed of five files: three headers and two code files as follows:

`types.h`	The header file defining the data structures used in the library.
`vector.h`	The header file containing the forward definitions of the external vector operation procedures.
`vector.c`	The code file containing the code for vector operations.
`polyhedron.h`	The header file containing the forward definitions of the external vector polyhedral and domain procedures.
`polyhedron.c`	The code file containing the code for polyhedral and domain operations.

The library is made available by ftp from `ftp.irisa.fr` and can be found in the directory `/local/API/`.

Appendix A

EXAMPLE C–PROGRAM

The following is an example program written to demonstrate how the library is called from a C-program. The first section has the code itself, the second section has the input for the program, and the last section has the output.

A.1 Program Code

```
/* main.c
   This file along with polyhedron.c and vector.c do the following functions:
    − Extraction of a minimal set of constraints from some set of constraints
    − Intersection of two convexes
    − Application of a linear function to some convex
    − Verification that a convex is included in some other convex */

#include "types.h"
#include "vector.h"
#include "polyhedron.h"

int main()
{   Matrix *a, *b, *t;
    Polyhedron *A, *B, *C, *D;

    /* read in a matrix containing your equations */
    /* for example, run this program and type in these five  lines:
        4 4
        1 0 1 −1
        1 −1 0 6
        1 0 −1 7
        1 1 0 −2
       This is a matrix for the following inequalities
        1 = inequality,  0x +  1y −1 ≥0  −−>     y ≥ 1
        1 = inequality, −1x +  0y +6 ≥0  −−>     x ≤ 6
        1 = inequality,  0x + −1y +7 ≥0  −−>     y ≤ 7
        1 = inequality,  1x +  0y −2 ≥0  −−>     x ≥ 2
       If the first number is a 0 instead of a 1, then that constraint
    is an 'equality' instead of an 'inequality'.
    */
    a = Matrix_Read();

    /* read in a second matrix containing a second set of constraints:
       for example :
        4 4
```

```
      1 1 0 -1
      1 -1 0 3
      1 0 -1 5
      1 0 1 -2
*/
b = Matrix_Read();

/* Convert the constraints to a Polyhedron.
This operation 1. Computes the dual ray/vertex form of the
system, and 2. Eliminates redundant constraints and reduces
them to a minimal form. */
A = Constraints2Polyhedron(a, 200);
B = Constraints2Polyhedron(b, 200);

/* the 200 is the size of the working space (in terms of number
of rays) that is allocated temporarily
-- you can enlarge or reduce it as needed */

/* There is likewise a rays to polyhedron procedure */

/* Since you are done with the matrices a and b, be a good citizen
and clean up your garbage */
Matrix_Free(a);
Matrix_Free(b);

/* If you want the the reduced set of equations back, you can
get the matrix back in the same format it started in... */
a = Polyhedron2Constraints(A);
b = Polyhedron2Constraints(B);

/* Take a look at them if you want */
printf("\na = ");  Matrix_Print("%4d", a);
printf("\nb = ");  Matrix_Print("%4d", b);

/* To intersect the two systems, use the polyhedron formats...
   Again, the 200 is the size of the working space. */
C = DomainIntersection(A, B, 200);

/* This time, lets look a the polyhedron itself... */
printf("\nC = A and B = ");  Polyhedron_Print("%4d", C);

/* The operations DomainUnion, DomainDifference, DomainConvex,
and DomainSimplify are also available */

/* read in a third matrix containing a transformation matrix,
   this one swaps the indices (x,y --> y,x):
      3 3
      0 1 0
      1 0 0
      0 0 1
*/
t = Matrix_Read();

/* Take the preimage (transform the equations) of the domain C to
get D. */
D = Polyhedron_Preimage(C, t, 200);

/* cleanup */
```

```
    Matrix_Free(t);

    printf("\nD␣=␣transformed␣C␣=");
    Polyhedron_Print("%4d", D);    Domain_Free(D);

    /* in a similar way, Polyhedron_Image(dom, mat, 200), takes the image
    of dom under matrix mat  (transforms the vertices/rays) */

    /* The function PolyhedronIncludes(Pol1, Pol2) returns 1 if Pol1
    includes (covers) Pol2, and a 0 otherwise */

    if (PolyhedronIncludes(A,C))
    printf("\nWe␣expected␣A␣to␣cover␣C␣since␣C␣=␣A␣intersect␣B\n");
    if (!PolyhedronIncludes(C,B))
    printf("and␣C␣does␣not␣cover␣B...\n");

    Domain_Free(A);
    Domain_Free(B);
    Domain_Free(C);

    return 0;
}
```

A.2 Program Input

```
4 4
1 0 1 -1
1 -1 0 6
1 0 -1 7
1 1 0 -2
4 4
1 1 0 -1
1 -1 0 3
1 0 -1 5
1 0 1 -2
3 3
0 1 0
1 0 0
0 0 1
```

A.3 Program Output

```
a =4 4
   1   0   1  -1
   1  -1   0   6
   1   0  -1   7
   1   1   0  -2

b =4 4
   1   1   0  -1
   1  -1   0   3
   1   0  -1   5
   1   0   1  -2

C = A and B =POLYHEDRON Dimension:2
           Constraints:4  Equations:0  Rays:4  Lines:0
```

```
Constraints 4 4
Inequality: [   1   0  -2 ]
Inequality: [  -1   0   3 ]
Inequality: [   0  -1   5 ]
Inequality: [   0   1  -2 ]
Rays 4 4
Vertex: [   3   5 ]/1
Vertex: [   2   5 ]/1
Vertex: [   2   2 ]/1
Vertex: [   3   2 ]/1

D = transformed C =POLYHEDRON Dimension:2
           Constraints:4  Equations:0  Rays:4  Lines:0
Constraints 4 4
Inequality: [   0   1  -2 ]
Inequality: [   0  -1   3 ]
Inequality: [  -1   0   5 ]
Inequality: [   1   0  -2 ]
Rays 4 4
Vertex: [   5   3 ]/1
Vertex: [   5   2 ]/1
Vertex: [   2   2 ]/1
Vertex: [   2   3 ]/1

We expected A to cover C since C = A intersect B
and C does not cover B...
```

Bibliography

[Che65] N. V. Chernikova. Algorithm for finding a general formula for the non- negative solutions of a system of linear inequalities. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228–133, 1965.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Volume 10 of *Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin, 1987.

[FQ88] F. Fernandez and P. Quinton. *Extension of Chernikova's Algorithm for Solving General Mixed Linear Programming Problems.* Technical Report 437, IRISA, Rennes,Fr, Oct 1988.

[Gol56] A. J. Goldman. Resolution and separation theorems for polyhedral convex sets. In H. W. Kuhn and A. W. Tucker, editors, *Linear inequalities and related systems*, Princeton University, Princeton,NJ, 1956.

[Gru67] B. Grunbaum. *Convex Polytopes.* Volume 16 of *Pure and Applied Mathematics*, John Wiley & Sons, London, 1967.

[Kle66] V. Klee. Convex polytopes and linear programming. *Proceedings IBM Scientific Computation Symposium: Combinational Problems*, 123–158, 1966.

[KT56] H. W. Kuhn and A. W. Tucker. *Linear inequalities and related systems. Annals of Mathematics Studies*, Princeton University, Princeton,NJ, 1956.

[Le 92] H. Le Verge. *A note on Chernikova's Algorithm.* Technical Report 635, IRISA, Feb 1992.

[LMQ91] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, September 1991.

[Mau89] Christophe Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, Rennes, France, Dec 1989.

[McM70] P. McMullen. The maximum number of faces of a convex polytope. *Mathematica*, XVII:179–184, 1970.

[MR80] T. H. Mattheiss and D. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5(2):167–185, May 1980.

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. *Theodore S. Motzkin: Selected Papers*, 1953.

[MS71] P. McMullen and G. C. Shepard. Convex polytopes and the upper bound conjecture. In *London Mathematical Society Lecture Notes Series*, Cambridge University Press, London, 1971.

[Rub75] D. Rubin. Vertex generation and cardinality constrained linear programs. *Operations Research*, 23(3):555–565, May 1975.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* John Wiley and Sons, NY, 1986.

[Sei91] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[Weh50] H. Wehl. The elementary theory of convex polyhedra. In *Annals of Mathematics Studies, Number 24*, pages 3–18, Princeton University Press, Princeton,NJ, 1950. written in 1933, originally published in German in 1935.

[Wil93] D. Wilde. *A library for Doing Polyhedral Operations.* Master's thesis, Oregon State University, Corvallis, Oregon, Dec 1993.