The Parma Polyhedra Library User's Manual[*] (version 0.5)

Roberto Bagnara[†]
Patricia M. Hill[‡]
Elisa Ricci[§]
Enea Zaffanella[¶]

based on previous work also by

Sara Bonini
Andrea Pescetti
Angela Stazzone
Tatiana Zolo[‖]

April 27, 2003

[*] This work has been partly supported by: University of Parma's FIL scientific research project (ex 60%) "Pure and Applied Mathematics"; MURST project "Automatic Program Certification by Abstract Interpretation"; MURST project "Abstract Interpretation, Type Systems and Control-Flow Analysis"; MURST project "Automatic Aggregate- and Number-Reasoning for Computing: from Decision Algorithms to Constraint Programming with Multisets, Sets, and Maps"; MURST project "Constraint Based Verification of Reactive Systems".

[†] bagnara@cs.unipr.it, Department of Mathematics, University of Parma, Italy.
[‡] hill@comp.leeds.ac.uk, School of Computing, University of Leeds, U.K.
[§] ericci@cs.unipr.it, Department of Mathematics, University of Parma, Italy.
[¶] zaffanella@cs.unipr.it, Department of Mathematics, University of Parma, Italy.
[‖] zolo@cs.unipr.it, Department of Mathematics, University of Parma, Italy.

For the most up-to-date information see the Parma Polyhedra Library WWW site:

`http://www.cs.unipr.it/ppl/`

Contents

1 Convex Polyhedra and the PPL

1.1 A Library for Convex Polyhedra

The Parma Polyhedra Library (PPL) is a modern C++ library for the manipulation of rational convex polyhedra. Informally, a rational convex polyhedron is a set of points (in some n-dimensional vector space) that satisfies a finite number of linear inequalities having rational coefficients. The domain of convex polyhedra is employed in several systems for the analysis and verification of hardware and software components, with applications spanning imperative, functional and logic programming languages, synchronous languages and synchronization protocols, real-time and hybrid systems. Even though the PPL library is not meant to target a particular problem, the design of its interface has been largely influenced by the needs of the above class of applications. That is the reason why the library implements a few operators that are more or less specific to static analysis applications, while lacking some other operators that might be useful when working, e.g., in the field of computational geometry.

The main features of the library are the following:

- it is user friendly: you write `x + 2*y + 5*z <= 7` when you mean it;
- it is fully dynamic: available virtual memory is the only limitation to the dimension of anything;
- it provides full support for the manipulation of convex polyhedra that are not topologically closed;
- it is written in standard C++: meant to be portable;
- it is exception-safe: never leaks resources or leaves invalid object fragments around;
- it is rather efficient: and we hope to make it even more so;
- it is thoroughly documented: perhaps not literate programming but close enough;
- it is free software: distributed under the terms of the GNU General Public License.

In the following sections we describe the polyhedra and the different representations and operations supported by the PPL in more detail. For more information about the definitions and results stated here see [BRZH02b], [Fuk98], [NW88], and [Wil93].

1.2 An Introduction to Convex Polyhedra

In this section we introduce convex polyhedra, as considered by the library, in more detail.

Vectors, Matrices and Scalar Products

We denote by $\mathbb{R}^n$ the n-dimensional vector space on the field of real numbers $\mathbb{R}$, endowed with the standard topology. The set of all non-negative reals is denoted by $\mathbb{R}_+$. For each $i \in \{0, \ldots, n-1\}$, v_i denotes the i-th component of the (column) vector $\boldsymbol{v} = (v_0, \ldots, v_{n-1})^{\mathrm{T}} \in \mathbb{R}^n$. We denote by $\mathbf{0}$ the vector of $\mathbb{R}^n$, called *the origin*, having all components equal to zero. A vector $\boldsymbol{v} \in \mathbb{R}^n$ can be also interpreted as a matrix in $\mathbb{R}^{n \times 1}$ and manipulated accordingly using the usual definitions for addition, multiplication (both by a scalar and by another matrix), and transposition, denoted by $\boldsymbol{v}^{\mathrm{T}}$.

The *scalar product* of $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$, denoted $\langle \boldsymbol{v}, \boldsymbol{w} \rangle$, is the real number

$$\boldsymbol{v}^{\mathrm{T}} \boldsymbol{w} = \sum_{i=0}^{n-1} v_i w_i.$$

For any $S_1, S_2 \subseteq \mathbb{R}^n$, the *Minkowski's sum* of S_1 and S_2 is: $S_1 + S_2 = \{\, \boldsymbol{v}_1 + \boldsymbol{v}_2 \mid \boldsymbol{v}_1 \in S_1, \boldsymbol{v}_2 \in S_2 \,\}$.

Affine Hyperplanes and Half-spaces

For each vector $\boldsymbol{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, where $\boldsymbol{a} \neq \mathbf{0}$, and for each relation symbol $\bowtie \in \{=, \geq, >\}$, the linear constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b$ defines:

- an affine hyperplane if it is an equality constraint, i.e., if $\bowtie \in \{=\}$;
- a topologically closed affine half-space if it is a non-strict inequality constraint, i.e., if $\bowtie \in \{\geq\}$;
- a topologically open affine half-space if it is a strict inequality constraint, i.e., if $\bowtie \in \{>\}$.

Note that each hyperplane $\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b$ can be defined as the intersection of the two closed affine half-spaces $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \geq b$ and $\langle -\boldsymbol{a}, \boldsymbol{x} \rangle \geq -b$. Also note that, when $\boldsymbol{a} = \boldsymbol{0}$, the constraint $\langle \boldsymbol{0}, \boldsymbol{x} \rangle \bowtie b$ is either a tautology (i.e., always true) or inconsistent (i.e., always false), so that it defines either the whole vector space $\mathbb{R}^n$ or the empty set $\emptyset$.

Convex Polyhedra

The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a *not necessarily closed convex polyhedron* (*NNC polyhedron*, for short) if and only if either $\mathcal{P}$ can be expressed as the intersection of a finite number of (open or closed) affine half-spaces of $\mathbb{R}^n$ or $n = 0$ and $\mathcal{P} = \emptyset$. The set of all NNC polyhedra on the vector space $\mathbb{R}^n$ is denoted $\mathbb{P}_n$.

The set $\mathcal{P} \in \mathbb{P}_n$ is a *closed convex polyhedron* (*closed polyhedron*, for short) if and only if either $\mathcal{P}$ can be expressed as the intersection of a finite number of closed affine half-spaces of $\mathbb{R}^n$ or $n = 0$ and $\mathcal{P} = \emptyset$. The set of all closed polyhedra on the vector space $\mathbb{R}^n$ is denoted $\mathbb{CP}_n$.

When ordering NNC polyhedra by the set inclusion relation, the empty set $\emptyset$ and the vector space $\mathbb{R}^n$ are, respectively, the smallest and the biggest elements of both $\mathbb{P}_n$ and $\mathbb{CP}_n$. The vector space $\mathbb{R}^n$ is also called the *universe* polyhedron.

In theoretical terms, $\mathbb{P}_n$ is a *lattice* under set inclusion and $\mathbb{CP}_n$ is a *sub-lattice* of $\mathbb{P}_n$.

Bounded Polyhedra

An NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ is *bounded* if there exists a $\lambda \in \mathbb{R}_+$ such that

$$\mathcal{P} \subseteq \bigl\{ \boldsymbol{x} \in \mathbb{R}^n \bigm| -\lambda \leq x_j \leq \lambda \text{ for } j = 0, \ldots, n-1 \bigr\}.$$

A bounded polyhedron is also called a *polytope*.

1.3 Representations of Convex Polyhedra

NNC polyhedra can be specified by using two possible representations, the constraints (or implicit) representation and the generators (or parametric) representation.

Constraints representation

In the sequel, we will simply write "equality" and "inequality" to mean "linear equality" and "linear inequality", respectively; also, we will refer to either an equality or an inequality as a *constraint*.

By definition, each polyhedron $\mathcal{P} \in \mathbb{P}_n$ is the set of solutions to a *constraint system*, i.e., a finite number of constraints. By using matrix notation, we have

$$P = \{ \boldsymbol{x} \in \mathbb{R}^n \mid A_1 \boldsymbol{x} = \boldsymbol{b}_1, A_2 \boldsymbol{x} \geq \boldsymbol{b}_2, A_3 \boldsymbol{x} > \boldsymbol{b}_3 \},$$

where, for all $i \in \{1, 2, 3\}$, $A_i \in \mathbb{R}^{m_i} \times \mathbb{R}^n$ and $\boldsymbol{b}_i \in \mathbb{R}^{m_i}$, and $m_1, m_2, m_3 \in \mathbb{N}$ are the number of equalities, the number of non-strict inequalities, and the number of strict inequalities, respectively.

Combinations and Hulls

Let $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \subseteq \mathbb{R}^n$ be a finite set of vectors. For all scalars $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$, the vector $\boldsymbol{v} = \sum_{j=1}^{k} \lambda_j \boldsymbol{x}_j$ is said to be a *linear* combination of the vectors in S. Such a combination is said to be

- a *positive* (or *conic*) combination, if $\forall j \in \{1, \ldots, k\} : \lambda_j \in \mathbb{R}_+$;
- an *affine* combination, if $\sum_{j=1}^{k} \lambda_j = 1$;
- a *convex* combination, if it is both positive and affine.

We denote by $\mathrm{linear.hull}(S)$ (resp., $\mathrm{conic.hull}(S)$, $\mathrm{affine.hull}(S)$, $\mathrm{convex.hull}(S)$) the set of all the linear (resp., positive, affine, convex) combinations of the vectors in S.

Let $P, C \subseteq \mathbb{R}^n$, where $P \cup C = S$. We denote by $\mathrm{nnc.hull}(P, C)$ the set of all convex combinations of the vectors in S such that $\lambda_j > 0$ for some $\boldsymbol{x}_j \in P$ (informally, we say that there exists a vector of P that plays an active role in the convex combination). Note that $\mathrm{nnc.hull}(P, C) = \mathrm{nnc.hull}(P, P \cup C)$ so that, if $C \subseteq P$,

$$\mathrm{convex.hull}(P) = \mathrm{nnc.hull}(P, \varnothing) = \mathrm{nnc.hull}(P, P) = \mathrm{nnc.hull}(P, C).$$

It can be observed that $\mathrm{linear.hull}(S)$ is an affine space, $\mathrm{conic.hull}(S)$ is a topologically closed convex cone, $\mathrm{convex.hull}(S)$ is a topologically closed polytope, and $\mathrm{nnc.hull}(P, C)$ is an NNC polytope.

Points, Closure Points, Rays and Lines

Let $\mathcal{P} \in \mathbb{P}_n$ be an NNC polyhedron. Then

- a vector $\boldsymbol{p} \in \mathcal{P}$ is called a *point* of $\mathcal{P}$;
- a vector $\boldsymbol{c} \in \mathbb{R}^n$ is called a *closure point* of $\mathcal{P}$ if it is a point of the topological closure of $\mathcal{P}$;
- a vector $\boldsymbol{r} \in \mathbb{R}^n$, where $\boldsymbol{r} \neq \boldsymbol{0}$, is called a *ray* (or direction of infinity) of $\mathcal{P}$ if $\mathcal{P} \neq \varnothing$ and $\boldsymbol{p} + \lambda \boldsymbol{r} \in \mathcal{P}$, for all points $\boldsymbol{p} \in \mathcal{P}$ and all $\lambda \in \mathbb{R}_+$;
- a vector $\boldsymbol{l} \in \mathbb{R}^n$ is called a *line* of $\mathcal{P}$ if both $\boldsymbol{l}$ and $-\boldsymbol{l}$ are rays of $\mathcal{P}$.

A point of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ is a *vertex* if and only if it cannot be expressed as a convex combination of any other pair of distinct points in $\mathcal{P}$. A ray $\boldsymbol{r}$ of a polyhedron $\mathcal{P}$ is an *extreme ray* if and only if it cannot be expressed as a positive combination of any other pair $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ of rays of $\mathcal{P}$, where $\boldsymbol{r} \neq \lambda \boldsymbol{r}_1$, $\boldsymbol{r} \neq \lambda \boldsymbol{r}_2$ and $\boldsymbol{r}_1 \neq \lambda \boldsymbol{r}_2$ for all $\lambda \in \mathbb{R}_+$ (i.e., rays differing by a positive scalar factor are considered to be the same ray).

Generators Representation

Each NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ can be represented by finite sets of lines L, rays R, points P and closure points C of $\mathcal{P}$. The 4-tuple $\mathcal{G} = (L, R, P, C)$ is said to be a *generator system* for $\mathcal{P}$, in the sense that

$$\mathcal{P} = \mathrm{linear.hull}(L) + \mathrm{conic.hull}(R) + \mathrm{nnc.hull}(P, C),$$

where the symbol '$+$' denotes the Minkowski's sum.

When $\mathcal{P} \in \mathbb{CP}_n$ is a closed polyhedron, then it can be represented by finite sets of lines L, rays R and points P of $\mathcal{P}$. In this case, the 3-tuple $\mathcal{G} = (L, R, P)$ is said to be a *generator system* for $\mathcal{P}$ since we have

$$\mathcal{P} = \mathrm{linear.hull}(L) + \mathrm{conic.hull}(R) + \mathrm{convex.hull}(P).$$

Thus, in this case, every closure point of $\mathcal{P}$ is a point of $\mathcal{P}$.

For any $\mathcal{P} \in \mathbb{P}_n$ and generator system $\mathcal{G} = (L, R, P, C)$ for $\mathcal{P}$, we have $\mathcal{P} = \varnothing$ if and only if $P = \varnothing$. Also P must contain all the vertices of $\mathcal{P}$ although $\mathcal{P}$ can be non-empty and have no vertices. In this case, as P is necessarily non-empty, it must contain points of $\mathcal{P}$ that are *not* vertices. For instance, the half-space of $\mathbb{R}^2$ corresponding to the single constraint $y \geq 0$ can be represented by the generator system $\mathcal{G} = (L, R, P, C)$ such that $L = \bigl\{(1, 0)^{\mathrm{T}}\bigr\}$, $R = \bigl\{(0, 1)^{\mathrm{T}}\bigr\}$, $P = \bigl\{(0, 0)^{\mathrm{T}}\bigr\}$, and $C = \varnothing$. It is also worth noting that the only ray in R is *not* an extreme ray of $\mathcal{P}$.

Minimized Representations

A constraints system $\mathcal{C}$ for an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ is said to be *minimized* if no proper subset of $\mathcal{C}$ is a constraint system for $\mathcal{P}$.

Similarly, a generator system $\mathcal{G} = (L, R, P, C)$ for an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ is said to be *minimized* if there does not exist a generator system $\mathcal{G}' = (L', R', P', C') \neq \mathcal{G}$ for $\mathcal{P}$ such that $L' \subseteq L$, $R' \subseteq R$, $P' \subseteq P$ and $C' \subseteq C$.

Double Description

Any NNC polyhedron $\mathcal{P}$ can be described by using a constraint system $\mathcal{C}$, a generator system $\mathcal{C}$, or both by means of the *double description pair (DD pair)* $(\mathcal{C}, \mathcal{G})$. The *double description method* is a collection of well-known as well as novel theoretical results showing that, given one kind of representation, there are algorithms for computing a representation of the other kind and for minimizing both representations by removing redundant constraints/generators.

Such changes of representation form a key step in the implementation of many operators on NNC polyhedra: this is because some operators, such as intersections and poly-hulls, are provided with a natural and efficient implementation when using one of the representations in a DD pair, while being rather cumbersome when using the other.

Topologies and Topological-compatibility

As indicated above, when an NNC polyhedron $\mathcal{P}$ is necessarily closed, we can ignore the closure points contained in its generator system $\mathcal{G} = (L, R, P, C)$ (as every closure point is also a point) and represent $\mathcal{P}$ by the triple (L, R, P). Similarly, $\mathcal{P}$ can be represented by a constraint system that has no strict inequalities. Thus a necessarily closed polyhedron can have a smaller representation than one that is not necessarily closed. Moreover, operators restricted to work on closed polyhedra only can be implemented more efficiently. For this reason the library provides two alternative "topological kinds" for a polyhedron, *NNC* and *C*. We shall abuse terminology by referring to the topologcal kind of a polyhedron as its *topology*.

In the library, the topology of each polyhedron object is fixed once for all at the time of its creation and must be respected when performing operations on the polyhedron.

Unless it is otherwise stated, all the polyhedra, constraints and/or generators in any library operation must obey the following *topological-compatibility* rules:

- polyhedra are topologically-compatible if and only if they have the same topology;
- all constraints except for strict inequality constraints and all generators except for closure points are topologically-compatible with both C and NNC polyhedra;
- strict inequality constraints and closure points are topologically-compatible with a polyhedron if and only if it is NNC.

Wherever possible, the library provides methods that, starting from a polyhedron of a given topology, build the corresponding polyhedron having the other topology.

Space Dimensions and Dimension-compatibility

The *space dimension* of an NNC polyhedron $P \in \mathbb{P}_n$ (resp., a C polyhedron $P \in \mathbb{CP}_n$) is the dimension $n \in \mathbb{N}$ of the corresponding vector space $\mathbb{R}^n$. The space dimension of constraints, generators and other objects of the library is defined similarly.

Unless it is otherwise stated, all the polyhedra, constraints and/or generators in any library operation must obey the following space *dimension-compatibility* rules:

- polyhedra are dimension-compatible if and only if they have the same space dimension;
- the constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b$ where $\bowtie \in \{=, \geq, >\}$ and $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^m$, is dimension-compatible with a polyhedron having space dimension n if and only if $m \leq n$;
- the generator $\boldsymbol{x} \in \mathbb{R}^m$ is dimension-compatible with a polyhedron having space dimension n if and only if $m \leq n$;
- a system of constraints (resp., generators) is dimension-compatible with a polyhedron if and only if all the constraints (resp., generators) in the system are dimension-compatible with the polyhedron.

While the space dimension of a constraint, a generator or a system thereof is automatically adjusted when needed, the space dimension of a polyhedron can only be changed by explicit calls to operators provided for that purpose.

Rational Polyhedra

An NNC polyhedron is called *rational* if it can be represented by a constraint system where all the constraints have rational coefficients. It has been shown that an NNC polyhedron is rational if and only if it can be represented by a generator system where all the generators have rational coefficients.

The library only supports rational polyhedra. The restriction to rational numbers applies not only to polyhedra, but also to the other numeric arguments that may be required by the operators considered, such as the coefficients defining (rational) affine transformations and (rational) bounding boxes.

1.4 Operations on Convex Polyhedra

In this section we briefly describe operations on NNC polyhedra that are provided by the library.

Intersection and Convex Polyhedral Hull

For any pair of NNC polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_n$, the *intersection* of $\mathcal{P}_1$ and $\mathcal{P}_2$, defined as the set intersection $\mathcal{P}_1 \cap \mathcal{P}_2$, is the biggest NNC polyhedron included in both $\mathcal{P}_1$ and $\mathcal{P}_2$; similarly, the *convex polyhedral hull* (or *poly-hull*) of $\mathcal{P}_1$ and $\mathcal{P}_2$, denoted by $\mathcal{P}_1 \uplus \mathcal{P}_2$, is the smallest NNC polyhedron that includes both $\mathcal{P}_1$ and $\mathcal{P}_2$. The intersection and poly-hull of any pair of closed polyhedra in $\mathbb{CP}_n$ is also closed.

In theoretical terms, the intersection and poly-hull operators defined above are the binary *meet* and the binary *join* operators on the lattices $\mathbb{P}_n$ and $\mathbb{CP}_n$.

Convex Polyhedral Difference

For any pair of NNC polyhedra $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{P}_n$, the *convex polyhedral difference* (or *poly-difference*) of $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as the poly-hull of the set-theoretic difference of $\mathcal{P}_1$ and $\mathcal{P}_2$.

In general, even though $\mathcal{P}_1, \mathcal{P}_2 \in \mathbb{CP}_n$ are topologically closed polyhedra, their poly-difference may be a convex polyhedron that is not topologically closed. For this reason, when computing the poly-difference of two C polyhedra, the library will enforce the topological closure of the result.

Adding New Dimensions to the Vector Space

The library provides two operators for increasing the space dimension of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$, therefore transforming it into a new NNC polyhedron $\mathcal{Q} \in \mathbb{P}_m$, where $m > n$. In both cases, the added dimensions of the vector space are those having the highest indices.

The operator *embedding* the polyhedron $\mathcal{P}$ into the new vector space will return the polyhedron $\mathcal{Q}$ defined by all and only the constraints defining $\mathcal{P}$ (the variables corresponding to the added dimensions are unconstrained). For instance, when starting from a polyhedron $\mathcal{P} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the polyhedron

$$\mathcal{Q} = \bigl\{\, (x_0, x_1, x_2)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x_0, x_1)^{\mathrm{T}} \in \mathcal{P} \,\bigr\}.$$

In contrast, the operator *projecting* the polyhedron $\mathcal{P}$ into the new vector space will return the polyhedron $\mathcal{Q}$ whose constraint system, besides the constraints defining $\mathcal{P}$, will include additional constraints on the added dimensions. Namely, the corresponding variables are all constrained to be equal to 0. For instance, when starting from a polyhedron $\mathcal{P} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the polyhedron

$$\mathcal{Q} = \bigl\{\, (x_0, x_1, 0)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x_0, x_1)^{\mathrm{T}} \in \mathcal{P} \,\bigr\}.$$

Removing Dimensions from the Vector Space

The library provides two operators for decreasing the space dimension of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$, therefore transforming it into a new NNC polyhedron $\mathcal{Q} \in \mathbb{P}_m$, where $m < n$.

Given a set of variables, there is an operator that will remove all the space dimensions corresponding to the variables in this set. For instance, letting $\mathcal{P} \in \mathbb{P}_4$ be the singleton set $\bigl\{(3, 1, 0, 2)^{\mathrm{T}}\bigr\} \subseteq \mathbb{R}^4$, then after invoking this operator with the set of variables $\{x_1, x_2\}$ the resulting polyhedron is

$$\mathcal{Q} = \bigl\{(3,2)^{\mathrm{T}}\bigr\} \subseteq \mathbb{R}^2.$$

Another operator removes from the vector space all the dimensions having an index greater than or equal to m. For instance, letting $\mathcal{P} \in \mathbb{P}_4$ defined as before, by invoking this operator with $m = 2$ the resulting polyhedron will be

$$\mathcal{Q} = \bigl\{(3,1)^{\mathrm{T}}\bigr\} \subseteq \mathbb{R}^2.$$

Mapping the Dimensions of the Vector Space

The library provides an operator to map the dimensions of the vector space $\mathbb{R}^n$ according to a partial injective function $\rho\colon \{0,\dots,n-1\} \rightarrowtail \mathbb{N}$ such that $\rho\bigl(\{0,\dots,n-1\}\bigr) = \{0,\dots,m-1\}$ with $m \leq n$. Dimensions corresponding to indices that are not mapped by ρ are removed.

If $m = 0$, i.e., if the function ρ is undefined everywhere, then the operator projects the argument polyhedron $\mathcal{P} \in \mathbb{P}_n$ onto the zero-dimension space $\mathbb{R}^0$; otherwise the result is $\mathcal{Q} \in \mathbb{P}_m$ given by

$$\mathcal{Q} \stackrel{\mathrm{def}}{=} \Bigl\{ \bigl(v_{\rho^{-1}(0)}, \dots, v_{\rho^{-1}(m-1)}\bigr)^{\mathrm{T}} \Bigm| (v_0, \dots, v_{n-1})^{\mathrm{T}} \in \mathcal{P} \Bigr\}.$$

Affine Images and Preimages

For each function mapping $\phi\colon \mathbb{R}^n \to \mathbb{R}^m$, we denote by $\phi(S) \subseteq \mathbb{R}^m$ the *image* under ϕ of the set $S \subseteq \mathbb{R}^n$; formally,

$$\phi(S) = \bigl\{\, \phi(\boldsymbol{v}) \in \mathbb{R}^m \bigm| \boldsymbol{v} \in S \,\bigr\}.$$

Similarly, we denote by $\phi^{-1}(S') \subseteq \mathbb{R}^n$ the *preimage* under ϕ of $S' \subseteq \mathbb{R}^m$, that is the largest set $S \subseteq \mathbb{R}^n$ such that $\phi(S) \subseteq S'$; formally,

$$\phi^{-1}(S') = \bigl\{\, \boldsymbol{v} \in \mathbb{R}^n \bigm| \phi(\boldsymbol{v}) \in S' \,\bigr\}.$$

The function mapping $\phi\colon \mathbb{R}^n \to \mathbb{R}^m$ is an *affine transformation* if there exist a matrix $A \in \mathbb{R}^m \times \mathbb{R}^n$ and a vector $\boldsymbol{b} \in \mathbb{R}^m$ such that, for all $\boldsymbol{x} \in \mathbb{R}^n$, we have $\phi(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$. If $n = m$, then the function ϕ is said to be *space-dimension preserving*.

Both $\mathbb{P}_n$ and $\mathbb{CP}_n$ are closed under the application of any space-dimension preserving affine image and preimage operators.

The library provides two operators, one computes an affine image and the other an affine preimage of a polyhedron $\mathcal{P} \in \mathbb{P}_n$ for a given variable x_k and linear expression $\mathit{expr} = \sum_{i=0}^{n-1} a_i x_i + b$. This variable and expression determine the affine transformation ϕ that is to be used by the operator. That is, ϕ is the transformation defined by the matrix and vector

$$A = \begin{pmatrix}
1 & & 0 & 0 & \cdots & \cdots & 0 \\
 & \ddots & & \vdots & & & \vdots \\
0 & & 1 & 0 & \cdots & \cdots & 0 \\
a_0 & \cdots & a_{k-1} & a_k & a_{k+1} & \cdots & a_{n-1} \\
0 & \cdots & \cdots & 0 & 1 & & 0 \\
\vdots & & & \vdots & & \ddots & \\
0 & \cdots & \cdots & 0 & 0 & & 1
\end{pmatrix}, \qquad
\boldsymbol{b} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

where the a_i (resp., b) occurs in the $(k+1)$st row in A (resp., position in $\boldsymbol{b}$). Thus ϕ transforms any point $(x_0, \dots, x_{n-1})^{\mathrm{T}}$ in the polyhedron $\mathcal{P}$ to

$$\Bigl(x_0, \dots, \bigl(\textstyle\sum_{i=0}^{n-1} a_i x_i + b\bigr), \dots, x_{n-1}\Bigr)^{\mathrm{T}}.$$

The affine image operator computes the affine image of $\mathcal{P}$ under ϕ. For instance, suppose the polyhedron $\mathcal{P}$ to be transformed is the square in $\mathbb{R}^2$ generated by the set of points $\{(0,0)^{\mathrm{T}}, (0,3)^{\mathrm{T}}, (3,0)^{\mathrm{T}}, (3,3)^{\mathrm{T}}\}$. Then, for example if the considered variable is x_0 and the linear expression $x_0 + 2x_1 + 4$ (so that $k = 0$, $a_0 = 1, a_1 = 2, b = 4$), the affine image operator will translate $\mathcal{P}$ to the parallelogram $\mathcal{P}_1$ generated by the set of points $\{(4,0)^{\mathrm{T}}, (10,3)^{\mathrm{T}}, (7,0)^{\mathrm{T}}, (13,3)^{\mathrm{T}}\}$ with height equal to the side of the square and oblique sides parallel to the line $x_0 - 2x_1$. If the considered variable is as before (i.e., $k = 0$) but the linear expression is x_1 (so that $a_0 = 0, a_1 = 1, b = 0$), then the resulting polyhedron $\mathcal{P}_2$ is the positive diagonal of the square.

The affine preimage operator computes the affine preimage of $\mathcal{P}$ under ϕ. For instance, suppose now that we apply the affine preimage operator as given in the first example using variable x_0 and linear expression $x_0 + 2x_1 + 4$ to the parallelogram $\mathcal{P}_1$; then we get the original square $\mathcal{P}$ back. If, on the other hand, we apply the affine preimage operator as given in the second example using variable x_0 and linear expression x_1 to $\mathcal{P}_2$, then the resulting polyhedron is a line that corresponds to the x_1 axes.

Observe that provided the coefficient a_k of the considered variable in the linear expression is non-zero, the affine transformation is invertible.

Generalized Affine Images

The library provides another operator which is a generalization of the affine image operator. Given a polyhedron $\mathcal{P} \in \mathbb{P}_n$, an affine expression $\mathit{lhs} = \sum_{i=0}^{n-1} a'_i x_i + b'$, a relation symbol $\bowtie \in \{<, \leq, =, \geq, >\}$, and an affine expression $\mathit{rhs} = \sum_{i=0}^{n-1} a_i x_i + b$, the image of $\mathcal{P}$ with respect to the transfer function $\mathit{lhs} \bowtie \mathit{rhs}$ is defined as

$$\left\{ (w_0, \ldots, w_{n-1})^{\mathrm{T}} \in \mathbb{R}^n \;\middle|\; \begin{array}{l} (v_0, \ldots, v_{n-1})^{\mathrm{T}} \in \mathcal{P}, \\ \bigl(i \in \{0, \ldots, n-1\} \land a'_i = 0 \implies w_i = v_i\bigr), \\ \sum_{i=0}^{n-1} a'_i w_i + b' \bowtie \sum_{i=0}^{n-1} a_i v_i + b \end{array} \right\}.$$

Note that, when $\mathit{lhs} = x_k$ and $\bowtie \in \{=\}$, then the above operator is equivalent to the application of the standard affine image of $\mathcal{P}$ with respect to the variable x_k and the affine expression rhs (hence the name given to this operator).

Time-Elapse Operator

The *time-elapse* operator has been defined in [HPR97]. Actually, the time-elapse operator provided by the library is a slight generalization of that one, since it also works on NNC polyhedra. For any two NNC polyhedra $\mathcal{P}, \mathcal{Q} \in \mathbb{P}_n$, the time-elapse between $\mathcal{P}$ and $\mathcal{Q}$, denoted $\mathcal{P} \nearrow \mathcal{Q}$, is the smallest NNC polyhedron containing the set

$$\left\{ \boldsymbol{p} + \lambda \boldsymbol{q} \in \mathbb{R}^n \;\middle|\; \boldsymbol{p} \in \mathcal{P}, \boldsymbol{q} \in \mathcal{Q}, \lambda \in \mathbb{R}_+ \right\}.$$

Note that, if $\mathcal{P}, \mathcal{Q} \in \mathbb{CP}_n$ are closed polyhedra, the above set is also a closed polyhedron. In contrast, when $\mathcal{Q}$ is not topologically closed, the above set might not be an NNC polyhedron.

Relation-with Operators

The library provides operators for checking the relation holding between an NNC polyhedron and either a constraint or a generator.

Suppose $\mathcal{P}$ is an NNC polyhedron and $\mathcal{C}$ an arbitrary constraint system representing $\mathcal{P}$. Suppose also that $c = \bigl(\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b\bigr)$ is a constraint with $\bowtie \in \{=, \geq, >\}$ and $\mathcal{Q}$ the set of points that satisfy c. The possible relations between $\mathcal{P}$ and c are as follows.

- $\mathcal{P}$ *is disjoint* from c if $\mathcal{P} \cap \mathcal{Q} = \varnothing$; that is, adding c to $\mathcal{C}$ gives us the empty polyhedron.
- $\mathcal{P}$ *strictly intersects* c if $\mathcal{P} \cap \mathcal{Q} \neq \varnothing$ and $\mathcal{P} \cap \mathcal{Q} \subset \mathcal{P}$; that is, adding c to $\mathcal{C}$ gives us a non-empty polyhedron strictly smaller than $\mathcal{P}$.
- $\mathcal{P}$ *is included* in c if $\mathcal{P} \subseteq \mathcal{Q}$; that is, adding c to $\mathcal{C}$ leaves $\mathcal{P}$ unchanged.

- $\mathcal{P}$ *saturates* c if $\mathcal{P} \subseteq \mathcal{H}$, where $\mathcal{H}$ is the hyperplane induced by constraint c, i.e., the set of points satisfying the equality constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b$; that is, adding the constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b$ to $\mathcal{C}$ leaves $\mathcal{P}$ unchanged.

The polyhedron $\mathcal{P}$ *subsumes* the generator g if adding g to any generator system representing $\mathcal{P}$ does not change $\mathcal{P}$.

Intervals, boxes and bounding boxes

An *interval* in $\mathbb{R}$ is a pair of *bounds*, called *lower* and *upper*. Each bound can be either (1) *closed and bounded*, (2) *open and bounded*, or (3) *open and unbounded*. If the bound is *bounded*, then it has a value in $\mathbb{R}$. An n-dimensional *box* $\mathcal{B}$ in $\mathbb{R}^n$ is a sequence of n intervals in $\mathbb{R}$.

The polyhedron $\mathcal{P}$ *represents a box* $\mathcal{B}$ in $\mathbb{R}^n$ if $\mathcal{P}$ is described by a constraint system in $\mathbb{R}^n$ that consists of one constraint for each bounded bound (lower and upper) in an interval in $\mathcal{B}$: Letting $\boldsymbol{e}_i = (0, \dots, 1, \dots, 0)^{\mathrm{T}}$ be the vector in $\mathbb{R}^n$ with 1 in the i'th position and zeroes in every other position; if the lower bound of the i'th interval in $\mathcal{B}$ is bounded, the corresponding constraint is defined as $\langle \boldsymbol{e}_i, \boldsymbol{x} \rangle \bowtie b$, where b is the value of the bound and $\bowtie$ is $\geq$ if it is a closed bound and $>$ if it is an open bound. Similarly, if the upper bound of the i'th interval in $\mathcal{B}$ is bounded, the corresponding constraint is defined as $\langle \boldsymbol{e}_i, \boldsymbol{x} \rangle \bowtie b$, where b is the value of the bound and $\bowtie$ is $\leq$ if it is a closed bound and $<$ if it is an open bound.

If every bound in the intervals defining a box $\mathcal{B}$ is either closed and bounded or open and unbounded, then $\mathcal{B}$ represents a closed polyhedron.

The *bounding box* of an NNC polyhedron $\mathcal{P}$ is the smallest n-dimensional box containing $\mathcal{P}$.

The library provides operations for computing the bounding box of an NNC polyhedron and conversely, for obtaining the NNC polyhedron representing a given bounding box.

Widening Operators

The library provides two widening operators for the domain of NNC polyhedra. The first one, that we call *H79-widening*, mainly follows the specification provided in the PhD thesis of N. Halbwachs [Hal79], also described in [HPR97]. There are a few differences between the H79-widening and the widening described in the cited paper. In particular, the H79-widening of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ using the NNC polyhedron $\mathcal{Q} \in \mathbb{P}_n$:

- allows for equalities in $\mathcal{P}$ and $\mathcal{Q}$ (the original definition is restricted to inequalities);
- requires as a precondition that $\mathcal{Q} \subseteq \mathcal{P}$.

The second widening operator, that we call *BHRZ03-widening*, is an instance of the specification provided in [BHRZ03]. This operator also requires as a precondition that $\mathcal{Q} \subseteq \mathcal{P}$ and it is guaranteed to provide a result which is at least as precise as the H79-widening.

Both widening operators can be applied to polyhedra that are not topologically closed. The user is warned that, in such a case, the results may not closely match the geometric intuition which is at the base of the specification of the two widenings. The reason is that, in the current implementation, the widenings are not directly applied to the NNC polyhedra, but rather to their internal representations. Implementation work is in progress and future versions of the library may provide an even better integration of the two widenings with the domain of NNC polyhedra.

Widening with Tokens

When approximating a fixpoint computation using widening operators, a common tactic to improve the precision of the final result is to delay the application of widening operators. The usual approach is to fix a parameter k and only apply widenings starting from the k-th iteration.

The library also supports an improved widening delay strategy, that we call *widening with tokens* [BHRZ03]. A token is a sort of wildcard allowing for the replacement of the widening application by the exact upper bound computation: the token is used (and thus consumed) only when the widening would have resulted in an actual precision loss (as opposed to the *potential* precision loss of the classical delay strategy). Thus, all widening operators can be supplied with an optional argument, recording the number of available tokens, which is decremented when tokens are used. The approximated fixpoint computation will start with a fixed number k of tokens, which will be used if and when needed. When there are no tokens left, the widening is always applied.

Extrapolation Operators

Besides the two widening operators, the library also implements several *extrapolation* operators, which differ from widenings in that their use along an upper iteration sequence does not ensure convergence in a finite number of steps.

In particular, for each of the two widenings there is a corresponding *limited* extrapolation operator, which can be used to implement the *widening "up to"* technique as described in [HPR97]. Each limited extrapolation operator takes a constraint system as an additional parameter and uses it to improve the approximation yielded by the corresponding widening operator. Note that a convergence guarantee can only be obtained by suitably restricting the set of constraints that can occur in this additional parameter. For instance, in [HPR97] this set is fixed once and for all before starting the computation of the upward iteration sequence.

The *bounded* extrapolation operators further enhance each one of the limited extrapolation operators described above, by ensuring that their results cannot be worse than the smallest bounding box enclosing the two argument polyhedra.

A Note on the Implementation of the Operators

When adopting the double description method, the implementation of the above operators on polyhedra may require an explicit conversion from one of the two representations into the other one, leading to algorithms having a worst-case exponential complexity. However, thanks to the adoption of lazy and incremental computation techniques, the library turns out to be rather efficient in many practical cases.

In earlier versions of the library, a number of operators were introduced in two flavors: a *lazy* version and an *eager* version, the latter having the operator name ending with `_and_minimize`. In principle, only the lazy versions should be used. The eager versions were added to help a knowledgeble user obtain better performance in particular cases. Basically, by invoking the eager version of an operator, the user is trading laziness to better exploit the incrementality of the inner library computations. Starting from version 0.5, the lazy and incremental computation techniques have been refined to achieve a better integration: as a consequence, the lazy versions of the operators are now almost always more efficient than the eager versions.

The only case when an eager computation still makes sense is when the well-known *fail-first* principle comes into play. For instance, if you have to compute the intersection of several polyhedra and you strongly suspect that the result will become empty after a few of these intersections, then you may obtain a better performance by calling the eager version of the intersection operator, since the minimization process also enforces an emptyness check. Note anyway that the same effect can be obtained by interleaving the calls of the lazy operator with explicit emptyness checks.

1.5 Bibliography

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747-789, 1999.

[BHRZ03] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. Quaderno, Dipartimento di Matematica, Università di Parma, Italy, 2003. Available at `http://www.cs.unipr.it/Publications/`.

[BHZ02a] R. Bagnara, P. M. Hill, and E. Zaffanella. A new encoding and implementation of not necessarily closed convex polyhedra. Quaderno 305, Dipartimento di Matematica, Università di Parma, Italy, 2002.

[BHZ02b] R. Bagnara, P. M. Hill, and E. Zaffanella. A new encoding of not necessarily closed convex polyhedra. In M. Carro, C. Vacheret, and K.-K. Lau, editors, *Proceedings of the 1st CoLogNet Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems*, pages 147-153, Madrid, Spain, 2002. Published as TR Number CLIP4/02.0, Universidad Politécnica de Madrid, Facultad de Informática.

[BJT99] F. Besson, T. P. Jensen, and J.-P. Talpin. Polyhedral analysis for synchronous languages. In A. Cortesi and G. Filé, editors, *Static Analysis: Proceedings of the 6th International Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 51-68, Venice, Italy, 1999. Springer-Verlag, Berlin.

[BRZH02a] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 213-229, Madrid, Spain, 2002. Springer-Verlag, Berlin.

[BRZH02b] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. Quaderno 286, Dipartimento di Matematica, Università di Parma, Italy, 2002. See also [BRZH02c]. Available at `http://www.cs.unipr.it/Publications/`.

[BRZH02c] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Errata for technical report "Quaderno 286". Available at `http://www.cs.unipr.it/Publications/`, 2002. See [BRZH02b].

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269-295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84-96, Tucson, Arizona, 1978. ACM Press.

[Che64] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of system of linear equations. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 4(4):151-158, 1964.

[Che65] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of system of linear inequalities. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228-233, 1965.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282-293, 1968.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[FP96] K. Fukuda and A. Prodon. Double description method revisited. In M. Deza, R. Euler, and Y. Manoussakis, editors, *Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Brest, France, July 3-5, 1995, Selected Papers*, volume 1120 of *Lecture Notes in Computer Science*, pages 91-111. Springer-Verlag, Berlin, 1996.

[Fuk98] K. Fukuda. Polyhedral computation FAQ. Swiss Federal Institute of Technology, Lausanne and Zurich, Switzerland, available at `http://www.ifor.math.ethz.ch/~fukuda/fukuda.html`, 1998.

[GJ00] E. Gawrilow and M. Joswig. `polymake`: a framework for analyzing convex polytopes. In G. Kalai and G. M. Ziegler, editors, *Polytopes - Combinatorics and Computation*, pages 43-74. Birkhäuser, 2000.

[GJ01] E. Gawrilow and M. Joswig. `polymake`: an approach to modular software design in computational geometry. In *Proceedings of the 17th Annual Symposium on Computational Geometry*, pages 222-231. ACM, 2001. June 3-5, 2001, Medford, MA.

[Hal79] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Computer Aided Verification: Proceedings of the 5th International Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 333-346, Elounda, Greece, 1993. Springer-Verlag, Berlin.

[HH95] T. A. Henzinger and P.-H. Ho. A note on abstract interpretation strategies for hybrid automata. In P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 252-264. Springer-Verlag, Berlin, 1995.

[HKP95] N. Halbwachs, A. Kerbrat, and Y.-E. Proy. *POLyhedra INtegrated Environment*. Verimag, France, version 1.0 of POLINE edition, September 1995. Documentation taken from source code.

[HLW94] V. Van Dongen H. Le Verge and D. K. Wilde. Loop nest synthesis using the polyhedral library. *Publication interne* 830, IRISA, Campus de Beaulieu, Rennes, France, 1994.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Static Analysis: Proceedings of the 1st International Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 223-237, Namur, Belgium, 1994. Springer-Verlag, Berlin.

[HPR97] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157-185, 1997.

[HPWT01] T. A. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the hytech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887-2892. IEEE Computer Society Press, 2001.

[Jea02] B. Jeannet. *Convex Polyhedra Library*, release 1.1.3c edition, March 2002. Documentation of the "New Polka" library available at `http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html`.

[Kuh56] H. W. Kuhn. Solvability and consistency for linear equations and inequalities. *American Mathematical Monthly*, 63:217-232, 1956.

[Le 92] 92 H. Le Verge. A note on Chernikova's algorithm. *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France, 1992.

[Loe99] V. Loechner. *PolyLib*: A library for manipulating parameterized polyhedra. Available at `http://icps.u-strasbg.fr/~loechner/polylib/`, March 1999. Declares itself to be a continuation of [Wil93].

[LW97] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525-549, 1997.

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games - Volume II*, number 28 in Annals of Mathematics Studies, pages 51-73. Princeton University Press, Princeton, New Jersey, 1953.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1988.

[Sri93] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):315-343, 1993.

[SW70] J. Stoer and C. Witzgall. *Convexity and Optimization in Finite Dimensions I*. Springer-Verlag, Berlin, 1970.

[Wey35] H. Weyl. Elementare theorie der konvexen polyeder. *Commentarii Mathematici Helvetici*, 7:290-306, 1935. English translation in [Wey50].

[Wey50] H. Weyl. The elementary theory of convex polyhedra. In H. W. Kuhn, editor, *Contributions to the Theory of Games - Volume I*, number 24 in Annals of Mathematics Studies, pages 3-18. Princeton University Press, Princeton, New Jersey, 1950. Translated from [Wey35] by H. W. Kuhn.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, December 1993. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.

2 PPL Module Index

2.1 PPL Modules

Here is a list of all modules:

3 PPL Namespace Index

3.1 PPL Namespace List

Here is a list of all documented namespaces with brief descriptions:

4 PPL Hierarchical Index

4.1 PPL Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

5 PPL Compound Index

5.1 PPL Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

6 PPL Module Documentation

6.1 The Library

The core implementation of the Parma Polyhedra Library is written in C++. See Namespace, Hierarchical and Compound indexes for additional information.

6.2 Library Defines

Defines

- #define **PPL_VERSION_MAJOR** 0

  The major number of the PPL version.

- #define **PPL_VERSION_MINOR** 5

  The minor number of the PPL version.

- #define **PPL_VERSION_REVISION** 0

  The revision number of the PPL version.

- #define **PPL_VERSION_BETA** 0

  The beta number of the PPL version. This is zero for official releases and nonzero for development snapshots.

6.3 C Language Interface

Initialization, Error Handling and Auxiliary Functions

- int **ppl_max_space_dimension** (ppl_dimension_type *m)

  Writes to `m` *the maximum space dimension this library can handle.*

- int **ppl_not_a_dimension** (ppl_dimension_type *m)

  Writes to `m` *a value that does not designate a valid dimension.*

- int **ppl_initialize** (void)

  Initializes the Parma Polyhedra Library. This function must be called before any other function.

- int **ppl_finalize** (void)

  Finalizes the Parma Polyhedra Library. This function must be called after any other function.

- int ppl_set_error_handler (void(*h)(enum ppl_enum_error_code code, const char *description))

  Installs the user-defined error handler pointed by `h`*.*

Functions Related to Coefficients

- int **ppl_new_Coefficient** (ppl_Coefficient_t *pc)

  Creates a new coefficient with value 0 and writes an handle for the newly created coefficient at address `pc`.

- int **ppl_new_Coefficient_from_mpz_t** (ppl_Coefficient_t *pc, mpz_t z)

  Creates a new coefficient with the value given by the GMP integer `z` *and writes an handle for the newly created coefficient at address* `pc`.

- int **ppl_new_Coefficient_from_Coefficient** (ppl_Coefficient_t *pc, ppl_const_Coefficient_t c)

  Builds a coefficient that is a copy of `c`*; writes an handle for the newly created coefficient at address* `pc`.

- int **ppl_assign_Coefficient_from_mpz_t** (ppl_Coefficient_t dst, mpz_t z)

  Assign to `dst` *the value given by the GMP integer* `z`.

- int **ppl_assign_Coefficient_from_Coefficient** (ppl_Coefficient_t dst, ppl_const_Coefficient_t src)

  Assigns a copy of the coefficient `src` *to* `dst`.

- int **ppl_delete_Coefficient** (ppl_const_Coefficient_t c)

  Invalidates the handle `c`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_Coefficient_to_mpz_t** (ppl_const_Coefficient_t c, mpz_t z)

  Sets the value of the GMP integer `z` *to the value of* `c`.

- int **ppl_Coefficient_OK** (ppl_const_Coefficient_t c)

  Returns a positive integer if `c` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `c` *is broken. Useful for debugging purposes.*

Functions Related to Linear Expressions

- int **ppl_new_LinExpression** (ppl_LinExpression_t *ple)

  Creates a new linear expression corresponding to the constant 0 in a zero-dimensional space; writes an handle for the new linear expression at address `ple`.

- int **ppl_new_LinExpression_with_dimension** (ppl_LinExpression_t *ple, ppl_dimension_type d)

  Creates a new linear expression corresponding to the constant 0 in a `d`*-dimensional space; writes an handle for the new linear expression at address* `ple`.

- int **ppl_new_LinExpression_from_LinExpression** (ppl_LinExpression_t *ple, ppl_const_LinExpression_t le)

  Builds a linear expression that is a copy of `le`*; writes an handle for the newly created linear expression at address* `ple`.

- int **ppl_new_LinExpression_from_Constraint** (ppl_LinExpression_t *ple, ppl_const_Constraint_t c)

  Builds a linear expression corresponding to constraint `c`*; writes an handle for the newly created linear expression at address* `ple`.

- int **ppl_new_LinExpression_from_Generator** (ppl_LinExpression_t *ple, ppl_const_Generator_t g)

Builds a linear expression corresponding to generator `g`*; writes an handle for the newly created linear expression at address* `ple`*.*

- int **ppl_delete_LinExpression** (ppl_const_LinExpression_t le)

  Invalidates the handle `le`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_LinExpression_from_LinExpression** (ppl_LinExpression_t dst, ppl_const_LinExpression_t src)

  Assigns a copy of the linear expression `src` *to* `dst`*.*

- int **ppl_LinExpression_add_to_coefficient** (ppl_LinExpression_t le, ppl_dimension_type var, ppl_const_Coefficient_t n)

  Adds `n` *to the coefficient of variable* `var` *in the linear expression* `le`*. The space dimension is set to be the maximum between* `var` *+ 1 and the old space dimension.*

- int **ppl_LinExpression_add_to_inhomogeneous** (ppl_LinExpression_t le, ppl_const_Coefficient_t n)

  Adds `n` *to the inhomogeneous term of the linear expression* `le`*.*

- int **ppl_add_LinExpression_to_LinExpression** (ppl_LinExpression_t dst, ppl_const_LinExpression_t src)

  Adds the linear expression `src` *to* `dst`*.*

- int **ppl_subtract_LinExpression_from_LinExpression** (ppl_LinExpression_t dst, ppl_const_LinExpression_t src)

  Subtracts the linear expression `src` *from* `dst`*.*

- int **ppl_multiply_LinExpression_by_Coefficient** (ppl_LinExpression_t le, ppl_const_Coefficient_t n)

  Multiply the linear expression `dst` *by* `n`*.*

- int **ppl_LinExpression_space_dimension** (ppl_const_LinExpression_t le)

  Returns the space dimension of `le`*.*

- int **ppl_LinExpression_coefficient** (ppl_const_LinExpression_t le, ppl_dimension_type var, ppl_Coefficient_t n)

  Copies into `n` *the coefficient of variable* `var` *in the linear expression* `le`*.*

- int **ppl_LinExpression_inhomogeneous_term** (ppl_const_LinExpression_t le, ppl_Coefficient_t n)

  Copies into `n` *the inhomogeneous term of linear expression* `le`*.*

- int **ppl_LinExpression_OK** (ppl_const_LinExpression_t le)

  Returns a positive integer if `le` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `le` *is broken. Useful for debugging purposes.*

Functions Related to Constraints

- int **ppl_new_Constraint** (ppl_Constraint_t *pc, ppl_const_LinExpression_t le, enum ppl_enum_Constraint_Type rel)

Creates the new constraint '`le rel` 0' and writes an handle for it at address `pc`. *The space dimension of the new constraint is equal to the space dimension of* `le`.

- int **ppl_new_Constraint_zero_dim_false** (ppl_Constraint_t *pc)

  Creates the unsatisfiable (zero-dimension space) constraint $0 = 1$ *and writes an handle for it at address* `pc`.

- int **ppl_new_Constraint_zero_dim_positivity** (ppl_Constraint_t *pc)

  Creates the true (zero-dimension space) constraint $0 \leq 1$*, also known as* positivity constraint. *An handle for the newly created constraint is written at address* `pc`.

- int **ppl_new_Constraint_from_Constraint** (ppl_Constraint_t *pc, ppl_const_Constraint_t c)

  Builds a constraint that is a copy of `c`*; writes an handle for the newly created constraint at address* `pc`.

- int **ppl_delete_Constraint** (ppl_const_Constraint_t c)

  Invalidates the handle `c`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_Constraint_from_Constraint** (ppl_Constraint_t dst, ppl_const_Constraint_t src)

  Assigns a copy of the constraint `src` *to* `dst`.

- int **ppl_Constraint_space_dimension** (ppl_const_Constraint_t c)

  Returns the space dimension of `c`.

- int **ppl_Constraint_type** (ppl_const_Constraint_t c)

  Returns the type of constraint `c`.

- int **ppl_Constraint_coefficient** (ppl_const_Constraint_t c, ppl_dimension_type var, ppl_Coefficient_t n)

  Copies into `n` *the coefficient of variable* `var` *in constraint* `c`.

- int **ppl_Constraint_inhomogeneous_term** (ppl_const_Constraint_t c, ppl_Coefficient_t n)

  Copies into `n` *the inhomogeneous term of constraint* `c`.

- int **ppl_Constraint_OK** (ppl_const_Constraint_t c)

  Returns a positive integer if `c` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `c` *is broken. Useful for debugging purposes.*

Functions Related to Constraint Systems

- int **ppl_new_ConSys** (ppl_ConSys_t *pcs)

  Builds an empty system of constraints and writes an handle to it at address `pcs`.

- int **ppl_new_ConSys_zero_dim_empty** (ppl_ConSys_t *pcs)

  Builds a zero-dimensional, unsatisfiable constraint system and writes an handle to it at address `pcs`.

- int **ppl_new_ConSys_from_Constraint** (ppl_ConSys_t *pcs, ppl_const_Constraint_t c)

  Builds the singleton constraint system containing only a copy of constraint `c`*; writes an handle for the newly created system at address* `pcs`.

- int **ppl_new_ConSys_from_ConSys** (ppl_ConSys_t *pcs, ppl_const_ConSys_t cs)

Builds a constraint system that is a copy of `cs`*; writes an handle for the newly created system at address* `pcs`*.*

- int **ppl_delete_ConSys** (ppl_const_ConSys_t cs)

  Invalidates the handle `cs`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_ConSys_from_ConSys** (ppl_ConSys_t dst, ppl_const_ConSys_t src)

  Assigns a copy of the constraint system `src` *to* `dst`*.*

- int **ppl_ConSys_space_dimension** (ppl_const_ConSys_t cs)

  Returns the dimension of the vector space enclosing `cs`*.*

- int **ppl_ConSys_clear** (ppl_ConSys_t cs)

  Removes all the constraints from the constraint system `cs` *and sets its space dimension to 0.*

- int **ppl_ConSys_insert_Constraint** (ppl_ConSys_t cs, ppl_const_Constraint_t c)

  Inserts a copy of the constraint `c` *into* `cs`*; the space dimension is increased, if necessary.*

- int **ppl_ConSys_OK** (ppl_const_ConSys_t c)

  Returns a positive integer if `cs` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `cs` *is broken. Useful for debugging purposes.*

- int **ppl_new_ConSys_const_iterator** (ppl_ConSys_const_iterator_t *pcit)

  Builds a new 'const iterator' and writes an handle to it at address `pcit`*.*

- int **ppl_new_ConSys_const_iterator_from_ConSys_const_iterator** (ppl_ConSys_const_iterator_t *pcit, ppl_const_ConSys_const_iterator_t cit)

  Builds a const iterator system that is a copy of `cit`*; writes an handle for the newly created const iterator at address* `pcit`*.*

- int **ppl_delete_ConSys_const_iterator** (ppl_const_ConSys_const_iterator_t cit)

  Invalidates the handle `cit`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_ConSys_const_iterator_from_ConSys_const_iterator** (ppl_ConSys_const_iterator_t dst, ppl_const_ConSys_const_iterator_t src)

  Assigns a copy of the const iterator `src` *to* `dst`*.*

- int **ppl_ConSys_begin** (ppl_const_ConSys_t cs, ppl_ConSys_const_iterator_t cit)

  Assigns to `cit` *a const iterator "pointing" to the beginning of the constraint system* `cs`*.*

- int **ppl_ConSys_end** (ppl_const_ConSys_t cs, ppl_ConSys_const_iterator_t cit)

  Assigns to `cit` *a const iterator "pointing" past the end of the constraint system* `cs`*.*

- int **ppl_ConSys_const_iterator_dereference** (ppl_const_ConSys_const_iterator_t cit, ppl_const_Constraint_t *pc)

  Dereference `cit` *writing a const handle to the resulting constraint at address* `pc`*.*

- int **ppl_ConSys_const_iterator_increment** (ppl_ConSys_const_iterator_t cit)

  Increment `cit` *so that it "points" to the next constraint.*

- int **ppl_ConSys_const_iterator_equal_test** (ppl_const_ConSys_const_iterator_t x, ppl_const_ConSys_const_iterator_t y)

  Returns a positive integer if the iterators corresponding to `x` *and* `y` *are equal; return 0 if they are different.*

Functions Related to Generators

- int **ppl_new_Generator** (ppl_Generator_t ∗pg, ppl_const_LinExpression_t le, enum ppl_enum_Generator_Type t, ppl_const_Coefficient_t d)

  Creates a new generator of direction `le` *and type* `t`*. If the generator to be created is a point or a closure point, the divisor* `d` *is applied to* `le`*. For other types of generators* `d` *is simply disregarded. An handle for the new generator is written at address* `pg`*. The space dimension of the new generator is equal to the space dimension of* `le`*.*

- int **ppl_new_Generator_zero_dim_point** (ppl_Generator_t ∗pg)

  Creates the point that is the origin of the zero-dimensional space $\mathbb{R}^0$*. Writes an handle for the new generator at address* `pg`*.*

- int **ppl_new_Generator_zero_dim_closure_point** (ppl_Generator_t ∗pg)

  Creates, as a closure point, the point that is the origin of the zero-dimensional space $\mathbb{R}^0$*. Writes an handle for the new generator at address* `pg`*.*

- int **ppl_new_Generator_from_Generator** (ppl_Generator_t ∗pg, ppl_const_Generator_t g)

  Builds a generator that is a copy of `g`*; writes an handle for the newly created generator at address* `pg`*.*

- int **ppl_delete_Generator** (ppl_const_Generator_t g)

  Invalidates the handle `g`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_Generator_from_Generator** (ppl_Generator_t dst, ppl_const_Generator_t src)

  Assigns a copy of the generator `src` *to* `dst`*.*

- int **ppl_Generator_space_dimension** (ppl_const_Generator_t g)

  Returns the space dimension of `g`*.*

- int **ppl_Generator_type** (ppl_const_Generator_t g)

  Returns the type of generator `g`*.*

- int **ppl_Generator_coefficient** (ppl_const_Generator_t g, ppl_dimension_type var, ppl_Coefficient_t n)

  Copies into `n` *the coefficient of variable* `var` *in generator* `g`*.*

- int **ppl_Generator_divisor** (ppl_const_Generator_t g, ppl_Coefficient_t n)

  If `g` *is a point or a closure point assigns its divisor to* `n`*.*

- int **ppl_Generator_OK** (ppl_const_Generator_t g)

  Returns a positive integer if `g` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `g` *is broken. Useful for debugging purposes.*

Functions Related to Generator Systems

- int **ppl_new_GenSys** (ppl_GenSys_t *pgs)

  Builds an empty system of generators and writes an handle to it at address `pgs`.

- int **ppl_new_GenSys_from_Generator** (ppl_GenSys_t *pgs, ppl_const_Generator_t g)

  Builds the singleton generator system containing only a copy of generator `g`*; writes an handle for the newly created system at address* `pgs`.

- int **ppl_new_GenSys_from_GenSys** (ppl_GenSys_t *pgs, ppl_const_GenSys_t gs)

  Builds a generator system that is a copy of `gs`*; writes an handle for the newly created system at address* `pgs`.

- int **ppl_delete_GenSys** (ppl_const_GenSys_t gs)

  Invalidates the handle `gs`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_GenSys_from_GenSys** (ppl_GenSys_t dst, ppl_const_GenSys_t src)

  Assigns a copy of the generator system `src` *to* `dst`.

- int **ppl_GenSys_space_dimension** (ppl_const_GenSys_t gs)

  Returns the dimension of the vector space enclosing `gs`.

- int **ppl_GenSys_clear** (ppl_GenSys_t gs)

  Removes all the generators from the generator system `gs` *and sets its space dimension to 0.*

- int **ppl_GenSys_insert_Generator** (ppl_GenSys_t gs, ppl_const_Generator_t g)

  Inserts a copy of the generator `g` *into* `gs`*; the space dimension is increased, if necessary.*

- int **ppl_GenSys_OK** (ppl_const_GenSys_t c)

  Returns a positive integer if `gs` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `gs` *is broken. Useful for debugging purposes.*

- int **ppl_new_GenSys_const_iterator** (ppl_GenSys_const_iterator_t *pgit)

  Builds a new 'const iterator' and writes an handle to it at address `pgit`.

- int **ppl_new_GenSys_const_iterator_from_GenSys_const_iterator** (ppl_GenSys_const_iterator_t *pgit, ppl_const_GenSys_const_iterator_t git)

  Builds a const iterator system that is a copy of `git`*; writes an handle for the newly created const iterator at address* `pgit`.

- int **ppl_delete_GenSys_const_iterator** (ppl_const_GenSys_const_iterator_t git)

  Invalidates the handle `git`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_assign_GenSys_const_iterator_from_GenSys_const_iterator** (ppl_GenSys_const_iterator_t dst, ppl_const_GenSys_const_iterator_t src)

  Assigns a copy of the const iterator `src` *to* `dst`.

- int **ppl_GenSys_begin** (ppl_const_GenSys_t gs, ppl_GenSys_const_iterator_t git)

  Assigns to `git` *a const iterator "pointing" to the beginning of the generator system* `gs`.

- int **ppl_GenSys_end** (ppl_const_GenSys_t gs, ppl_GenSys_const_iterator_t git)

Assigns to `git` *a const iterator "pointing" past the end of the generator system* `gs`.

- int **ppl_GenSys_const_iterator_dereference** (ppl_const_GenSys_const_iterator_t git, ppl_const_Generator_t *pg)

  Dereference `git` *writing a const handle to the resulting generator at address* `pg`.

- int **ppl_GenSys_const_iterator_increment** (ppl_GenSys_const_iterator_t git)

  Increment `git` *so that it "points" to the next generator.*

- int **ppl_GenSys_const_iterator_equal_test** (ppl_const_GenSys_const_iterator_t x, ppl_const_GenSys_const_iterator_t y)

  Return a positive integer if the iterators corresponding to `x` *and* `y` *are equal; return 0 if they are different.*

Functions Related to Polyhedra

- int **ppl_new_C_Polyhedron_from_dimension** (ppl_Polyhedron_t *pph, ppl_dimension_type d)

  Builds an universe closed polyhedron of dimension `d` *and writes an handle to it at address* `pph`.

- int **ppl_new_NNC_Polyhedron_from_dimension** (ppl_Polyhedron_t *pph, ppl_dimension_type d)

  Builds an universe NNC polyhedron of dimension `d` *and writes an handle to it at address* `pph`.

- int **ppl_new_C_Polyhedron_empty_from_dimension** (ppl_Polyhedron_t *pph, ppl_dimension_type d)

  Builds an empty closed polyhedron of dimension `d` *and writes an handle to it at address* `pph`.

- int **ppl_new_NNC_Polyhedron_empty_from_dimension** (ppl_Polyhedron_t *pph, ppl_dimension_type d)

  Builds an empty NNC polyhedron of dimension `d` *and writes an handle to it at address* `pph`.

- int **ppl_new_C_Polyhedron_from_C_Polyhedron** (ppl_Polyhedron_t *pph, ppl_const_Polyhedron_t ph)

  Builds a closed polyhedron that is a copy of `ph`*; writes an handle for the newly created polyhedron at address* `pph`.

- int **ppl_new_C_Polyhedron_from_NNC_Polyhedron** (ppl_Polyhedron_t *pph, ppl_const_Polyhedron_t ph)

  Builds a closed polyhedron that is a copy of of the NNC polyhedron `ph`*; writes an handle for the newly created polyhedron at address* `pph`.

- int **ppl_new_NNC_Polyhedron_from_C_Polyhedron** (ppl_Polyhedron_t *pph, ppl_const_Polyhedron_t ph)

  Builds an NNC polyhedron that is a copy of of the closed polyhedron `ph`*; writes an handle for the newly created polyhedron at address* `pph`.

- int **ppl_new_NNC_Polyhedron_from_NNC_Polyhedron** (ppl_Polyhedron_t *pph, ppl_const_Polyhedron_t ph)

  Builds an NNC polyhedron that is a copy of `ph`*; writes an handle for the newly created polyhedron at address* `pph`.

- int **ppl_new_C_Polyhedron_from_ConSys** (ppl_Polyhedron_t *pph, ppl_const_ConSys_t cs)

Builds a new closed polyhedron from the system of constraints `cs` *and writes an handle for the newly created polyhedron at address* `pph`. *The new polyhedron will inherit the space dimension of* `cs`.

- int ppl_new_C_Polyhedron_recycle_ConSys (ppl_Polyhedron_t ∗pph, ppl_ConSys_t cs)

  Builds a new closed polyhedron recycling the system of constraints `cs` *and writes an handle for the newly created polyhedron at address* `pph`. *Since* `cs` *will be* the *system of constraints of the new polyhedron, the space dimension is also inherited.*

- int **ppl_new_NNC_Polyhedron_from_ConSys** (ppl_Polyhedron_t ∗pph, ppl_const_ConSys_t cs)

  Builds a new NNC polyhedron from the system of constraints `cs` *and writes an handle for the newly created polyhedron at address* `pph`. *The new polyhedron will inherit the space dimension of* `cs`.

- int ppl_new_NNC_Polyhedron_recycle_ConSys (ppl_Polyhedron_t ∗pph, ppl_ConSys_t cs)

  Builds a new NNC polyhedron recycling the system of constraints `cs` *and writes an handle for the newly created polyhedron at address* `pph`. *Since* `cs` *will be* the *system of constraints of the new polyhedron, the space dimension is also inherited.*

- int **ppl_new_C_Polyhedron_from_GenSys** (ppl_Polyhedron_t ∗pph, ppl_const_GenSys_t gs)

  Builds a new closed polyhedron from the system of generators `gs` *and writes an handle for the newly created polyhedron at address* `pph`. *The new polyhedron will inherit the space dimension of* `gs`.

- int ppl_new_C_Polyhedron_recycle_GenSys (ppl_Polyhedron_t ∗pph, ppl_GenSys_t gs)

  Builds a new closed polyhedron recycling the system of generators `gs` *and writes an handle for the newly created polyhedron at address* `pph`. *Since* `gs` *will be* the *system of generators of the new polyhedron, the space dimension is also inherited.*

- int **ppl_new_NNC_Polyhedron_from_GenSys** (ppl_Polyhedron_t ∗pph, ppl_const_GenSys_t gs)

  Builds a new NNC polyhedron from the system of generators `gs` *and writes an handle for the newly created polyhedron at address* `pph`. *The new polyhedron will inherit the space dimension of* `gs`.

- int ppl_new_NNC_Polyhedron_recycle_GenSys (ppl_Polyhedron_t ∗pph, ppl_GenSys_t gs)

  Builds a new NNC polyhedron recycling the system of generators `gs` *and writes an handle for the newly created polyhedron at address* `pph`. *Since* `gs` *will be* the *system of generators of the new polyhedron, the space dimension is also inherited.*

- int ppl_new_C_Polyhedron_from_bounding_box (ppl_Polyhedron_t ∗pph, ppl_dimension_type(∗space_dimension)(void), int(∗is_empty)(void), int(∗get_lower_bound)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d), int(∗get_upper_bound)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d))

  Builds a new C polyhedron corresponding to an interval-based bounding box, writing a handle for the newly created polyhedron at address `pph`.

- int ppl_new_NNC_Polyhedron_from_bounding_box (ppl_Polyhedron_t ∗pph, ppl_dimension_type(∗space_dimension)(void), int(∗is_empty)(void), int(∗get_lower_bound)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d), int(∗get_upper_bound)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d))

  Builds a new C polyhedron corresponding to an interval-based bounding box, writing a handle for the newly created polyhedron at address `pph`.

- int **ppl_assign_C_Polyhedron_from_C_Polyhedron** (ppl_Polyhedron_t dst, ppl_const_Polyhedron_t src)

  Assigns a copy of the closed polyhedron `src` *to the closed polyhedron* `dst`.

- int **ppl_assign_NNC_Polyhedron_from_NNC_Polyhedron** (ppl_Polyhedron_t dst, ppl_const_-Polyhedron_t src)

  Assigns a copy of the NNC polyhedron `src` *to the NNC polyhedron* `dst`.

- int **ppl_delete_Polyhedron** (ppl_const_Polyhedron_t ph)

  Invalidates the handle `ph`*: this makes sure the corresponding resources will eventually be released.*

- int **ppl_Polyhedron_space_dimension** (ppl_const_Polyhedron_t ph)

  Returns the dimension of the vector space enclosing `ph`.

- int **ppl_Polyhedron_constraints** (ppl_const_Polyhedron_t ph, ppl_const_ConSys_t ∗pcs)

  Writes a const handle to the constraint system defining the polyhedron `ph` *at address* `pcs`.

- int **ppl_Polyhedron_minimized_constraints** (ppl_const_Polyhedron_t ph, ppl_const_ConSys_t ∗pcs)

  Writes a const handle to the minimized constraint system defining the polyhedron `ph` *at address* `pcs`.

- int **ppl_Polyhedron_generators** (ppl_const_Polyhedron_t ph, ppl_const_GenSys_t ∗pgs)

  Writes a const handle to the generator system defining the polyhedron `ph` *at address* `pgs`.

- int **ppl_Polyhedron_minimized_generators** (ppl_const_Polyhedron_t ph, ppl_const_GenSys_t ∗pgs)

  Writes a const handle to the minimized generator system defining the polyhedron `ph` *at address* `pgs`.

- int ppl_Polyhedron_relation_with_Constraint (ppl_const_Polyhedron_t ph, ppl_const_Constraint_t c)

  Checks the relation between the polyhedron `ph` *with the constraint* `c`.

- int ppl_Polyhedron_relation_with_Generator (ppl_const_Polyhedron_t ph, ppl_const_Generator_t g)

  Checks the relation between the polyhedron `ph` *with the generator* `g`.

- int ppl_Polyhedron_shrink_bounding_box (ppl_const_Polyhedron_t ph, unsigned int complexity, void(∗set_empty)(void), void(∗raise_lower_bound)(ppl_dimension_type k, int closed, ppl_const_-Coefficient_t n, ppl_const_Coefficient_t d), void(∗lower_upper_bound)(ppl_dimension_type k, int closed, ppl_const_Coefficient_t n, ppl_const_Coefficient_t d))

  Use `ph` *to shrink a generic, interval-based bounding box. The bounding box is abstractly provided by means of the parameters,.*

- int **ppl_Polyhedron_is_empty** (ppl_const_Polyhedron_t ph)

  Returns a positive integer if `ph` *is empty; returns 0 if* `ph` *is not empty.*

- int **ppl_Polyhedron_is_universe** (ppl_const_Polyhedron_t ph)

  Returns a positive integer if `ph` *is a universe polyhedron; returns 0 if it is not.*

- int **ppl_Polyhedron_is_bounded** (ppl_const_Polyhedron_t ph)

  Returns a positive integer if `ph` *is bounded; returns 0 if* `ph` *is unbounded.*

- int **ppl_Polyhedron_bounds_from_above** (ppl_const_Polyhedron_t ph, ppl_const_LinExpression_-t le)

  Returns a positive integer if `le` *is bounded from above in* `ph`*; returns 0 otherwise.*

- int **ppl_Polyhedron_bounds_from_below** (ppl_const_Polyhedron_t ph, ppl_const_LinExpression_-t le)

Returns a positive integer if `le` *is bounded from below in* `ph`*; returns 0 otherwise.*

- int **ppl_Polyhedron_is_topologically_closed** (ppl_const_Polyhedron_t ph)

  Returns a positive integer if `ph` *is topologically closed; returns 0 if* `ph` *is not topologically closed.*

- int **ppl_Polyhedron_contains_Polyhedron** (ppl_const_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Returns a positive integer if `x` *contains or is equal to* `y`*; returns 0 if it does not.*

- int **ppl_Polyhedron_strictly_contains_Polyhedron** (ppl_const_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Returns a positive integer if `x` *strictly contains* `y`*; returns 0 if it does not.*

- int **ppl_Polyhedron_is_disjoint_from_Polyhedron** (ppl_const_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Returns a positive integer if `x` *and* `y` *are disjoint; returns 0 if they are not.*

- int ppl_Polyhedron_equals_Polyhedron (ppl_const_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Returns a positive integer if `x` *and* `y` *are the same polyhedron; return 0 if they are different.*

- int **ppl_Polyhedron_OK** (ppl_const_Polyhedron_t ph)

  Returns a positive integer if `ph` *is well formed, i.e., if it satisfies all its implementation invariants; returns 0 and perhaps make some noise if* `ph` *is broken. Useful for debugging purposes.*

- int **ppl_Polyhedron_add_constraint** (ppl_Polyhedron_t ph, ppl_const_Constraint_t c)

  Adds a copy of the constraint `c` *to the system of constraints of* `ph`*.*

- int **ppl_Polyhedron_add_constraint_and_minimize** (ppl_Polyhedron_t ph, ppl_const_Constraint_t c)

  Adds a copy of the constraint `c` *to the system of constraints of* `ph`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `ph` *is guaranteed to be minimized.*

- int **ppl_Polyhedron_add_generator** (ppl_Polyhedron_t ph, ppl_const_Generator_t g)

  Adds a copy of the generator `g` *to the system of generators of* `ph`*.*

- int **ppl_Polyhedron_add_generator_and_minimize** (ppl_Polyhedron_t ph, ppl_const_Generator_t g)

  Adds a copy of the generator `g` *to the system of generators of* `ph`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `ph` *is guaranteed to be minimized.*

- int ppl_Polyhedron_add_constraints (ppl_Polyhedron_t ph, ppl_ConSys_t cs)

  Adds the system of constraints `cs` *to the system of constraints of* `ph`*.*

- int ppl_Polyhedron_add_constraints_and_minimize (ppl_Polyhedron_t ph, ppl_ConSys_t cs)

  Adds the system of constraints `cs` *to the system of constraints of* `ph`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `ph` *is guaranteed to be minimized.*

- int ppl_Polyhedron_add_generators (ppl_Polyhedron_t ph, ppl_GenSys_t gs)

  Adds the system of generators `gs` *to the system of generators of* `ph`*.*

- int ppl_Polyhedron_add_generators_and_minimize (ppl_Polyhedron_t ph, ppl_GenSys_t gs)

Adds the system of generators `gs` *to the system of generators of* `ph`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `ph` *is guaranteed to be minimized.*

- int **ppl_Polyhedron_intersection_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Intersects `x` *with polyhedron* `y` *and assigns the result* `x`*.*

- int **ppl_Polyhedron_intersection_assign_and_minimize** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Intersects `x` *with polyhedron* `y` *and assigns the result* `x`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `x` *is also guaranteed to be minimized.*

- int **ppl_Polyhedron_poly_hull_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Assigns to `x` *the poly-hull of the set-theoretic union of* `x` *and* `y`*.*

- int **ppl_Polyhedron_poly_hull_assign_and_minimize** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Assigns to `x` *the poly-hull of the set-theoretic union of* `x` *and* `y`*. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return,* `x` *is also guaranteed to be minimized.*

- int **ppl_Polyhedron_poly_difference_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Assigns to `x` *the poly-hull of the set-theoretic difference of* `x` *and* `y`*.*

- int ppl_Polyhedron_affine_image (ppl_Polyhedron_t ph, ppl_dimension_type var, ppl_const_LinExpression_t le, ppl_const_Coefficient_t d)

  Transforms the polyhedron `ph`*, assigning an affine expression to the specified variable.*

- int ppl_Polyhedron_affine_preimage (ppl_Polyhedron_t ph, ppl_dimension_type var, ppl_const_LinExpression_t le, ppl_const_Coefficient_t d)

  Transforms the polyhedron `ph`*, substituting an affine expression to the specified variable.*

- int ppl_Polyhedron_generalized_affine_image (ppl_Polyhedron_t ph, ppl_dimension_type var, enum ppl_enum_Constraint_Type relsym, ppl_const_LinExpression_t le, ppl_const_Coefficient_t d)

  Assigns to `ph` *the image of* `ph` *with respect to the generalized affine transfer function* $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$*, where* $\bowtie$ *is the relation symbol encoded by* `relsym`*.*

- int ppl_Polyhedron_generalized_affine_image_lhs_rhs (ppl_Polyhedron_t ph, ppl_const_LinExpression_t lhs, enum ppl_enum_Constraint_Type relsym, ppl_const_LinExpression_t rhs)

  Assigns to `ph` *the image of* `ph` *with respect to the generalized affine transfer function* $\mathrm{lhs}' \bowtie \mathrm{rhs}$*, where* $\bowtie$ *is the relation symbol encoded by* `relsym`*.*

- int **ppl_Polyhedron_time_elapse_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Assigns to `x` *the time-elapse between the polyhedra* `x` *and* `y`*.*

- int **ppl_Polyhedron_BHRZ03_widening_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, unsigned *tp)

  If the polyhedron `y` *is contained in (or equal to) the polyhedron* `x`*, assigns to* `x` *the BHRZ03-widening of* `x` *and* `y`*. If* `tp` *is not the null pointer, the widening with tokens delay technique is applied with* `*tp` *available tokens.*

- int **ppl_Polyhedron_BHRZ03_widening_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the BHRZ03-widening of `x` and `y`.

- int **ppl_Polyhedron_limited_BHRZ03_extrapolation_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs, unsigned *tp)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the BHRZ03-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`. If `tp` is not the null pointer, the widening with tokens delay technique is applied with `*tp` available tokens.

- int **ppl_Polyhedron_limited_BHRZ03_extrapolation_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the BHRZ03-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`.

- int **ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs, unsigned *tp)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the BHRZ03-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`, further intersected with all the constraints of the form $\pm v \leq r$ and $\pm v < r$, with $r \in \mathbb{Q}$, that are satisfied by all the points of `x`. If `tp` is not the null pointer, the widening with tokens delay technique is applied with `*tp` available tokens.

- int **ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the BHRZ03-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`, further intersected with all the constraints of the form $\pm v \leq r$ and $\pm v < r$, with $r \in \mathbb{Q}$, that are satisfied by all the points of `x`.

- int **ppl_Polyhedron_H79_widening_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, unsigned *tp)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the H79-widening of `x` and `y`. If `tp` is not the null pointer, the widening with tokens delay technique is applied with `*tp` available tokens.

- int **ppl_Polyhedron_H79_widening_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the H79-widening of `x` and `y`.

- int **ppl_Polyhedron_limited_H79_extrapolation_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs, unsigned *tp)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the H79-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`. If `tp` is not the null pointer, the widening with tokens delay technique is applied with `*tp` available tokens.

- int **ppl_Polyhedron_limited_H79_extrapolation_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the H79-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`.

- int **ppl_Polyhedron_bounded_H79_extrapolation_assign_with_tokens** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs, unsigned *tp)

  If the polyhedron `y` is contained in (or equal to) the polyhedron `x`, assigns to `x` the H79-widening of `x` and `y` intersected with the constraints in `cs` that are satisfied by all the points of `x`, further intersected with all the constraints of the form $\pm v \leq r$ and $\pm v < r$, with $r \in \mathbb{Q}$, that are satisfied by all the points of `x`. If `tp` is not the null pointer, the widening with tokens delay technique is applied with `*tp` available tokens.

- int **ppl_Polyhedron_bounded_H79_extrapolation_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y, ppl_const_ConSys_t cs)

  If the polyhedron `y` *is contained in (or equal to) the polyhedron* `x`*, assigns to* `x` *the H79-widening of* `x` *and* `y` *intersected with the constraints in* `cs` *that are satisfied by all the points of* `x`*, further intersected with all the constraints of the form* $\pm v \leq r$ *and* $\pm v < r$*, with* $r \in \mathbb{Q}$*, that are satisfied by all the points of* `x`*.*

- int **ppl_Polyhedron_topological_closure_assign** (ppl_Polyhedron_t ph)

  Assigns to `ph` *its topological closure.*

- int **ppl_Polyhedron_add_dimensions_and_embed** (ppl_Polyhedron_t ph, ppl_dimension_type d)

  Adds `d` *new dimensions to the space enclosing the polyhedron* `ph` *and to* `ph` *itself.*

- int **ppl_Polyhedron_add_dimensions_and_project** (ppl_Polyhedron_t ph, ppl_dimension_type d)

  Adds `d` *new dimensions to the space enclosing the polyhedron* `ph`*.*

- int **ppl_Polyhedron_concatenate_assign** (ppl_Polyhedron_t x, ppl_const_Polyhedron_t y)

  Seeing a polyhedron as a set of tuples (its points), assigns to `x` *all the tuples that can be obtained by concatenating, in the order given, a tuple of* `x` *with a tuple of* `y`*.*

- int **ppl_Polyhedron_remove_dimensions** (ppl_Polyhedron_t ph, ppl_dimension_type ds[ ], size_t n)

  Removes from `ph` *and its containing space the dimensions that are specified in first* `n` *positions of the array* `ds`*. The presence of duplicates in* `ds` *is a waste but an innocuous one.*

- int **ppl_Polyhedron_remove_higher_dimensions** (ppl_Polyhedron_t ph, ppl_dimension_type d)

  Removes the higher dimensions from `ph` *and its enclosing space so that, upon successful return, the new space dimension is* `d`*.*

- int ppl_Polyhedron_map_dimensions (ppl_Polyhedron_t ph, ppl_dimension_type maps[ ], size_t n)

  Remaps the dimensions of the vector space according to a partial function. This function is specified by means of the `maps` *array, which has* `n` *entries.*

Typedefs

- typedef size_t **ppl_dimension_type**

  An unsigned integral type for representing space dimensions.

- typedef ppl_Coefficient_tag $*$ **ppl_Coefficient_t**

  Opaque pointer to Coefficient .

- typedef ppl_Coefficient_tag const $*$ **ppl_const_Coefficient_t**

  Opaque pointer to const Coefficient .

- typedef ppl_LinExpression_tag $*$ **ppl_LinExpression_t**

  Opaque pointer to LinExpression .

- typedef ppl_LinExpression_tag const $*$ **ppl_const_LinExpression_t**

  Opaque pointer to const LinExpression .

- typedef ppl_Constraint_tag $*$ **ppl_Constraint_t**

Opaque pointer to Constraint .

- typedef ppl_Constraint_tag const * **ppl_const_Constraint_t**

  Opaque pointer to const Constraint .

- typedef ppl_ConSys_tag * **ppl_ConSys_t**

  Opaque pointer to ConSys .

- typedef ppl_ConSys_tag const * **ppl_const_ConSys_t**

  Opaque pointer to const ConSys .

- typedef ppl_ConSys_const_iterator_tag * **ppl_ConSys_const_iterator_t**

  Opaque pointer to ConSys_const_iterator .

- typedef ppl_ConSys_const_iterator_tag const * **ppl_const_ConSys_const_iterator_t**

  Opaque pointer to const ConSys_const_iterator .

- typedef ppl_Generator_tag * **ppl_Generator_t**

  Opaque pointer to Generator .

- typedef ppl_Generator_tag const * **ppl_const_Generator_t**

  Opaque pointer to const Generator .

- typedef ppl_GenSys_tag * **ppl_GenSys_t**

  Opaque pointer to GenSys .

- typedef ppl_GenSys_tag const * **ppl_const_GenSys_t**

  Opaque pointer to const GenSys .

- typedef ppl_GenSys_const_iterator_tag * **ppl_GenSys_const_iterator_t**

  Opaque pointer to GenSys_const_iterator .

- typedef ppl_GenSys_const_iterator_tag const * **ppl_const_GenSys_const_iterator_t**

  Opaque pointer to const GenSys_const_iterator .

- typedef ppl_Polyhedron_tag * **ppl_Polyhedron_t**

  Opaque pointer to Polyhedron .

- typedef ppl_Polyhedron_tag const * **ppl_const_Polyhedron_t**

  Opaque pointer to const Polyhedron .

Enumerations

- enum ppl_enum_error_code {

  PPL_ERROR_OUT_OF_MEMORY, PPL_ERROR_INVALID_ARGUMENT, PPL_ERROR_-INTERNAL_ERROR, PPL_ERROR_UNKNOWN_STANDARD_EXCEPTION,

  PPL_ERROR_UNEXPECTED_ERROR }

  Defines the error code that any function can return.

- enum ppl_enum_Constraint_Type {
  PPL_CONSTRAINT_TYPE_LESS_THAN, PPL_CONSTRAINT_TYPE_LESS_THAN_OR_-EQUAL, PPL_CONSTRAINT_TYPE_EQUAL, PPL_CONSTRAINT_TYPE_GREATER_THAN_-OR_EQUAL,
  PPL_CONSTRAINT_TYPE_GREATER_THAN }

  Describes the relations represented by a constraint.

- enum ppl_enum_Generator_Type { PPL_GENERATOR_TYPE_LINE, PPL_GENERATOR_TYPE_-RAY, PPL_GENERATOR_TYPE_POINT, PPL_GENERATOR_TYPE_CLOSURE_POINT }

  Describes the different kinds of generators.

Variables

- unsigned int **PPL_COMPLEXITY_CLASS_POLYNOMIAL**

  Code of the worst-case polynomial complexity class.

- unsigned int **PPL_COMPLEXITY_CLASS_SIMPLEX**

  Code of the worst-case exponential but typically polynomial complexity class.

- unsigned int **PPL_COMPLEXITY_CLASS_ANY**

  Code of the universal complexity class.

- unsigned int **PPL_POLY_CON_RELATION_IS_DISJOINT**

  Individual bit saying that the polyhedron and the set of points satisfying the constraint are disjoint.

- unsigned int **PPL_POLY_CON_RELATION_STRICTLY_INTERSECTS**

  Individual bit saying that the polyhedron intersects the set of points satisfying the constraint, but it is not included in it.

- unsigned int **PPL_POLY_CON_RELATION_IS_INCLUDED**

  Individual bit saying that the polyhedron is included in the set of points satisfying the constraint.

- unsigned int **PPL_POLY_CON_RELATION_SATURATES**

  Individual bit saying that the polyhedron is included in the set of points saturating the constraint.

- unsigned int **PPL_POLY_GEN_RELATION_SUBSUMES**

  Individual bit saying that adding the generator would not change the polyhedron.

6.3.1 Enumeration Type Documentation

6.3.1.1 enum ppl_enum_error_code

Defines the error code that any function can return.

Enumeration values:

PPL_ERROR_OUT_OF_MEMORY The virtual memory available to the process has been exhausted.

PPL_ERROR_INVALID_ARGUMENT A function has been invoked with an invalid argument.

PPL_ERROR_INTERNAL_ERROR An internal error that was diagnosed by the PPL itself. This indicates a bug in the PPL.

PPL_ERROR_UNKNOWN_STANDARD_EXCEPTION A standard exception has been raised by the C++ run-time environment. This indicates a bug in the PPL.

PPL_ERROR_UNEXPECTED_ERROR A totally unknown, totally unexpected error happened. This indicates a bug in the PPL.

6.3.1.2 enum ppl_enum_Constraint_Type

Describes the relations represented by a constraint.

Enumeration values:

PPL_CONSTRAINT_TYPE_LESS_THAN The constraint is of the form $e < 0$.

PPL_CONSTRAINT_TYPE_LESS_THAN_OR_EQUAL The constraint is of the form $e \leq 0$.

PPL_CONSTRAINT_TYPE_EQUAL The constraint is of the form $e = 0$.

PPL_CONSTRAINT_TYPE_GREATER_THAN_OR_EQUAL The constraint is of the form $e \geq 0$.

PPL_CONSTRAINT_TYPE_GREATER_THAN The constraint is of the form $e > 0$.

6.3.1.3 enum ppl_enum_Generator_Type

Describes the different kinds of generators.

Enumeration values:

PPL_GENERATOR_TYPE_LINE The generator is a line.

PPL_GENERATOR_TYPE_RAY The generator is a ray.

PPL_GENERATOR_TYPE_POINT The generator is a point.

PPL_GENERATOR_TYPE_CLOSURE_POINT The generator is a closure point.

6.3.2 Function Documentation

6.3.2.1 int ppl_set_error_handler (void($*$ *h*)(enum ppl_enum_error_code code, const char $*$description))

Installs the user-defined error handler pointed by `h`.

The error handler takes an error code and a textual description that gives further information about the actual error. The C string containing the textual description is read-only and its existence it not guaranteed after the handler has returned.

6.3.2.2 int ppl_new_C_Polyhedron_recycle_ConSys (ppl_Polyhedron_t $*$ *pph*, ppl_ConSys_t *cs*)

Builds a new closed polyhedron recycling the system of constraints `cs` and writes an handle for the newly created polyhedron at address `pph`. Since `cs` will be *the* system of constraints of the new polyhedron, the space dimension is also inherited.

Warning:

This function modifies the constraint system referenced by `cs`: upon return, no assumption can be made on its value.

6.3.2.3 int ppl_new_NNC_Polyhedron_recycle_ConSys (ppl_Polyhedron_t * *pph*, ppl_ConSys_t *cs*)

Builds a new NNC polyhedron recycling the system of constraints `cs` and writes an handle for the newly created polyhedron at address `pph`. Since `cs` will be *the* system of constraints of the new polyhedron, the space dimension is also inherited.

Warning:
This function modifies the constraint system referenced by `cs`: upon return, no assumption can be made on its value.

6.3.2.4 int ppl_new_C_Polyhedron_recycle_GenSys (ppl_Polyhedron_t * *pph*, ppl_GenSys_t *gs*)

Builds a new closed polyhedron recycling the system of generators `gs` and writes an handle for the newly created polyhedron at address `pph`. Since `gs` will be *the* system of generators of the new polyhedron, the space dimension is also inherited.

Warning:
This function modifies the generator system referenced by `gs`: upon return, no assumption can be made on its value.

6.3.2.5 int ppl_new_NNC_Polyhedron_recycle_GenSys (ppl_Polyhedron_t * *pph*, ppl_GenSys_t *gs*)

Builds a new NNC polyhedron recycling the system of generators `gs` and writes an handle for the newly created polyhedron at address `pph`. Since `gs` will be *the* system of generators of the new polyhedron, the space dimension is also inherited.

Warning:
This function modifies the generator system referenced by `gs`: upon return, no assumption can be made on its value.

6.3.2.6 int ppl_new_C_Polyhedron_from_bounding_box (ppl_Polyhedron_t * *pph*, ppl_dimension_type(* *space_dimension*)(void), int(* *is_empty*)(void), int(* *get_lower_bound*)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d), int(* *get_upper_bound*)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d))

Builds a new C polyhedron corresponding to an interval-based bounding box, writing a handle for the newly created polyhedron at address `pph`.

If an interval of the bounding box is provided with any finite but open bound, then the polyhedron is not built and the value `PPL_ERROR_INVALID_ARGUMENT` is returned. The bounding box is accessed by using the following functions, passed as arguments:

```
ppl_dimension_type space_dimension()
```

returns the dimension of the vector space enclosing the polyhedron represented by the bounding box.

```
int is_empty()
```

returns 0 if and only if the bounding box describes a non-empty set. The function `is_empty()` will always be called before the other functions. However, if `is_empty()` does not return 0, none of the functions below will be called.

```
int get_lower_bound(ppl_dimension_type k, int closed,
                    ppl_Coefficient_t n, ppl_Coefficient_t d)
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from below, simply return 0. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to 0 if the lower boundary of I is open and is set to a value different from zero otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the greatest lower bound of I. The fraction n/d is in canonical form if and only if n and d have no common factors and d is positive, $0/1$ being the unique representation for zero.

```
int get_upper_bound(ppl_dimension_type k, int closed,
                    ppl_Coefficient_t n, ppl_Coefficient_t d)
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from above, simply return 0. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to 0 if the upper boundary of I is open and is set to a value different from 0 otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the least upper bound of I.

6.3.2.7 int ppl_new_NNC_Polyhedron_from_bounding_box (ppl_Polyhedron_t ∗ *pph*, ppl_dimension_type(∗ *space_dimension*)(void), int(∗ *is_empty*)(void), int(∗ *get_lower_bound*)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d), int(∗ *get_upper_bound*)(ppl_dimension_type k, int closed, ppl_Coefficient_t n, ppl_Coefficient_t d))

Builds a new C polyhedron corresponding to an interval-based bounding box, writing a handle for the newly created polyhedron at address `pph`.

The bounding box is accessed by using the following functions, passed as arguments:

```
ppl_dimension_type space_dimension()
```

returns the dimension of the vector space enclosing the polyhedron represented by the bounding box.

```
int is_empty()
```

returns 0 if and only if the bounding box describes a non-empty set. The function `is_empty()` will always be called before the other functions. However, if `is_empty()` does not return 0, none of the functions below will be called.

```
int get_lower_bound(ppl_dimension_type k, int closed,
                    ppl_Coefficient_t n, ppl_Coefficient_t d)
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from below, simply return 0. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to 0 if the lower boundary of I is open and is set to a value different from zero otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the greatest lower bound of I. The fraction n/d is in canonical form if and only if n and d have no common factors and d is positive, $0/1$ being the unique representation for zero.

```
int get_upper_bound(ppl_dimension_type k, int closed,
                    ppl_Coefficient_t n, ppl_Coefficient_t d)
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from above, simply return 0. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to 0 if the upper boundary of I is open and is set to a value different from 0 otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the least upper bound of I.

6.3.2.8 int ppl_Polyhedron_relation_with_Constraint (ppl_const_Polyhedron_t *ph*, ppl_const_Constraint_t *c*)

Checks the relation between the polyhedron `ph` with the constraint `c`.

If successful, returns a non-negative integer that is obtained as the bitwise or of the bits (chosen among PPL_POLY_CON_RELATION_IS_DISJOINT PPL_POLY_CON_RELATION_STRICTLY_INTERSECTS, PPL_POLY_CON_RELATION_IS_INCLUDED, and PPL_POLY_CON_RELATION_SATURATES) that describe the relation between `ph` and `c`.

6.3.2.9 int ppl_Polyhedron_relation_with_Generator (ppl_const_Polyhedron_t *ph*, ppl_const_Generator_t *g*)

Checks the relation between the polyhedron `ph` with the generator `g`.

If successful, returns a non-negative integer that is obtained as the bitwise or of the bits (only PPL_POLY_GEN_RELATION_SUBSUMES, at present) that describe the relation between `ph` and `g`.

6.3.2.10 int ppl_Polyhedron_shrink_bounding_box (ppl_const_Polyhedron_t *ph*, unsigned int *complexity*, void(∗ *set_empty*)(void), void(∗ *raise_lower_bound*)(ppl_dimension_type k, int closed, ppl_const_Coefficient_t n, ppl_const_Coefficient_t d), void(∗ *lower_upper_bound*)(ppl_dimension_type k, int closed, ppl_const_Coefficient_t n, ppl_const_Coefficient_t d))

Use `ph` to shrink a generic, interval-based bounding box. The bounding box is abstractly provided by means of the parameters,.

Parameters:

complexity The code of the complexity class of the algorithm to be used. Must be one of PPL_COMPLEXITY_CLASS_POLYNOMIAL, PPL_COMPLEXITY_CLASS_SIMPLEX, or PPL_COMPLEXITY_CLASS_ANY.

ph The polyhedron that is used to shrink the bounding box.

set_empty a pointer to a void function with no arguments that causes the bounding box to become empty, i.e., to represent the empty set.

raise_lower_bound a pointer to a void function with arguments `(ppl_dimension_type k, int closed, ppl_const_Coefficient_t n, ppl_const_Coefficient_t d)` that intersects the interval corresponding to the `k`-th dimension with $[n/d, +\infty)$ if `closed` is non-zero, with $(n/d, +\infty)$ if `closed` is zero. The fraction n/d is in canonical form, that is, n and d have no common factors and d is positive, $0/1$ being the unique representation for zero.

lower_upper_bound a pointer to a void function with argument `(ppl_dimension_type k, int closed, ppl_const_Coefficient_t n, ppl_const_Coefficient_t d)` that intersects the interval corresponding to the `k`-th dimension with $(-\infty, n/d]$ if `closed` is non-zero, with $(-\infty, n/d)$ if `closed` is zero. The fraction n/d is in canonical form.

6.3.2.11 int ppl_Polyhedron_equals_Polyhedron (ppl_const_Polyhedron_t *x*, ppl_const_Polyhedron_t *y*)

Returns a positive integer if `x` and `y` are the same polyhedron; return 0 if they are different.

Note that `x` and `y` may be topology- and/or dimension-incompatible polyhedra: in those cases, the value 0 is returned.

6.3.2.12 int ppl_Polyhedron_add_constraints (ppl_Polyhedron_t *ph*, ppl_ConSys_t *cs*)

Adds the system of constraints `cs` to the system of constraints of `ph`.

Warning:

This function modifies the constraint system referenced by `cs`: upon return, no assumption can be made on its value.

6.3.2.13 int ppl_Polyhedron_add_constraints_and_minimize (ppl_Polyhedron_t *ph*, ppl_ConSys_t *cs*)

Adds the system of constraints `cs` to the system of constraints of `ph`. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return, `ph` is guaranteed to be minimized.

Warning:

This function modifies the constraint system referenced by `cs`: upon return, no assumption can be made on its value.

6.3.2.14 int ppl_Polyhedron_add_generators (ppl_Polyhedron_t *ph*, ppl_GenSys_t *gs*)

Adds the system of generators `gs` to the system of generators of `ph`.

Warning:

This function modifies the generator system referenced by `gs`: upon return, no assumption can be made on its value.

6.3.2.15 int ppl_Polyhedron_add_generators_and_minimize (ppl_Polyhedron_t *ph*, ppl_GenSys_t *gs*)

Adds the system of generators `gs` to the system of generators of `ph`. Returns a positive integer if the resulting polyhedron is non-empty; returns 0 if it is empty. Upon successful return, `ph` is guaranteed to be minimized.

Warning:

This function modifies the generator system referenced by `gs`: upon return, no assumption can be made on its value.

6.3.2.16 int ppl_Polyhedron_affine_image (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_LinExpression_t *le*, ppl_const_Coefficient_t *d*)

Transforms the polyhedron `ph`, assigning an affine expression to the specified variable.

Parameters:

ph The polyhedron that is transformed.

var The variable to which the affine expression is assigned.

le The numerator of the affine expression.

d The denominator of the affine expression.

6.3.2.17 int ppl_Polyhedron_affine_preimage (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_LinExpression_t *le*, ppl_const_Coefficient_t *d*)

Transforms the polyhedron `ph`, substituting an affine expression to the specified variable.

Parameters:

ph The polyhedron that is transformed.

var The variable to which the affine expression is substituted.

le The numerator of the affine expression.

d The denominator of the affine expression.

6.3.2.18 int ppl_Polyhedron_generalized_affine_image (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, enum ppl_enum_Constraint_Type *relsym*, ppl_const_LinExpression_t *le*, ppl_const_Coefficient_t *d*)

Assigns to `ph` the image of `ph` with respect to the generalized affine transfer function $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$, where $\bowtie$ is the relation symbol encoded by `relsym`.

Parameters:

ph The polyhedron that is transformed.

var The left hand side variable of the generalized affine transfer function.

relsym The relation symbol.

le The numerator of the right hand side affine expression.

d The denominator of the right hand side affine expression.

6.3.2.19 int ppl_Polyhedron_generalized_affine_image_lhs_rhs (ppl_Polyhedron_t *ph*, ppl_const_LinExpression_t *lhs*, enum ppl_enum_Constraint_Type *relsym*, ppl_const_LinExpression_t *rhs*)

Assigns to `ph` the image of `ph` with respect to the generalized affine transfer function $\mathrm{lhs}' \bowtie \mathrm{rhs}$, where $\bowtie$ is the relation symbol encoded by `relsym`.

Parameters:

ph The polyhedron that is transformed.

lhs The left hand side affine expression.

relsym The relation symbol.

rhs The right hand side affine expression.

6.3.2.20 int ppl_Polyhedron_map_dimensions (ppl_Polyhedron_t *ph*, ppl_dimension_type *maps*[ ], size_t *n*)

Remaps the dimensions of the vector space according to a partial function. This function is specified by means of the `maps` array, which has `n` entries.

The partial function is defined on dimension `i` if `i` $<$ `n` and `maps[i] != ppl_not_a_dimension`; otherwise it is undefined on dimension `i`. If the function is defined on dimension `i`, then dimension `i` is mapped onto dimension `maps[i]`.

The result is undefined if `maps` does not encode a partial function with the properties described in the specification of the mapping operator.

6.4 Prolog Language Interface

6.4.1 Introduction

The Parma Polyhedra Library comes equipped with a Prolog interface. Despite the lack of standardization of Prolog's foreign language interfaces, the PPL Prolog interface supports several Prolog systems and, to the extent this is possible, provides a uniform view of the library from each such systems.

The system-independent features of the library are described in Section System-Independent Features. Section Compilation and Installation explains how the various incarnations of the Prolog interface are compiled and installed. Section System-Dependent Features illustrates the system-dependent features of the interface for all the supported systems.

6.4.2 System-Independent Features

The Prolog interface provides access to the PPL polyhedra. A general introduction to convex polyhedra, their representation in the PPL and the operations provided by the PPL is given in Sections A Library for Convex Polyhedra, An Introduction to Convex Polyhedra, Representations of Convex Polyhedra and Operations on Convex Polyhedra of this manual. Here we just describe those aspects that are specific to the Prolog interface.

6.4.2.1 Overview First, here is a list of notes with general information and advice on the use of the interface.

- A PPL polyhedron can only be accessed by means of a Prolog term called a *handle*. Note, however, that the data structure of a handle, is implementation-dependent, system-dependent and version-dependent, and, for this reason, deliberately left unspecified. What we do guarantee is that the handle requires very little memory.

- A Prolog term can be bound to a valid handle by using:

```
ppl_new_Polyhedron_from_dimension/3,
ppl_new_Polyhedron_empty_from_dimension/3,
ppl_new_Polyhedron_from_Polyhedron/4,
ppl_new_Polyhedron_from_constraints/3,
ppl_new_Polyhedron_from_generators/3.
ppl_new_Polyhedron_from_bounding_box/3.
```

  These predicates will create or copy a PPL polyhedron and construct a valid handle for referencing it. The first argument (in the case of `ppl_new_Polyhedron_from_Polyhedron/4`, the first and third arguments) denotes the topology and can be either `c` or `nnc` indicating a C or NNC polyhedron, respectively. The third argument (in the case of `ppl_new_Polyhedron_from_Polyhedron/4`, the fourth argument) is a Prolog term that is unified with a new valid handle for accessing this polyhedron.

- As soon as a PPL polyhedron is no longer required, the memory occupied by it should be released using the PPL predicate `ppl_delete_Polyhedron/1`. To understand why this is important, consider a Prolog program and a variable that is bound to a Herbrand term. When the variable dies (goes out of scope) or is uninstantiated (on backtracking) the term it is bound to is amenable to garbage collection. But this only applies for the standard domain of the language: Herbrand terms. In Prolog+PPL, when a variable bound to a handle for a PPL Polyhedron dies or is uninstantiated, the handle can be garbage-collected, but the polyhedra to which the handle refers will not be released. Once a handle has been used as an argument in `ppl_delete_Polyhedron/1`, it becomes invalid.

- For a PPL polyhedron with space dimension `k`, the identifiers used for the PPL variables must lie between 0 and $k-1$ and correspond to the indices of the associated Cartesian axes. When using the predicates that combine PPL polyhedra or add constraints or generators to a representation of a PPL polyhedron, the polyhedra referenced and any constraints or generators in the call should follow all the space dimension-compatibility rules stated in Section Representations of Convex Polyhedra.

- As explained above, a polyhedron has a fixed topology C or NNC, that is determined at the time of its initialization. All subsequent operations on the polyhedron must respect all the topological compatibility rules stated in Section Representations of Convex Polyhedra.

- The predicates `ppl_initialize/0` and `ppl_finalize/0` initialize and finalize, respectively, the Prolog interface. Thus the only interface predicates callable after `ppl_finalize/0` are `ppl_finalize/0` itself (this further call has no effect) and `ppl_initialize/0`, after which the interface's services are usable again. Some Prolog systems allow the specification of initialization and deinitialization functions in their foreign language interfaces. The corresponding incarnations of the PPL-Prolog interface have been written so that `ppl_initialize/0` and/or `ppl_finalize/0` are called automatically. Section System-Dependent Features will detail in which cases initialization and finalization is automatically performed or is left to the Prolog programmer's responsibility. However, for portable applications, it is best to invoke `ppl_initialize/0` and `ppl_finalize/0` explicitly: since they can be called multiple times without problems, this will result in enhanced portability at a cost that is, by all means, negligible.

6.4.2.2 PPL Predicate List Here is a list of all the PPL predicates provided by the Prolog interface.

`ppl_initialize`

`ppl_finalize`

`ppl_set_timeout_exception_atom(+Atom)`

`ppl_set_timeout(+Integer)`

`ppl_reset_timeout`

`ppl_new_Polyhedron_from_dimension(+Topology, +Integer, -Handle)`

`ppl_new_Polyhedron_empty_from_dimension(+Topology, +Integer, -Handle)`

`ppl_new_Polyhedron_from_Polyhedron(+Topology_1, +Handle_1, +Topology_2, -Handle_2)`

`ppl_new_Polyhedron_from_constraints(+Topology, +Constraint_System, -Handle)`

`ppl_new_Polyhedron_from_generators(+Topology, +Generator_System, -Handle)`

`ppl_new_Polyhedron_from_bounding_box(+Topology, +Box, -Handle)`

`ppl_Polyhedron_swap(+Handle1, +Handle2)`

`ppl_delete_Polyhedron(+Handle)`

`ppl_Polyhedron_space_dimension(+Handle, -Integer)`

`ppl_Polyhedron_get_constraints(+Handle, -Constraint_System)`

`ppl_Polyhedron_get_minimized_constraints(+Handle, -Constraint_System)`

`ppl_Polyhedron_get_generators(+Handle, -Generator_System)`

`ppl_Polyhedron_get_minimized_generators(+Handle, -Generator_System)` ppl_Polyhedron_relation_with_constraint(+Handle, +Constraint, -Relation)

ppl_Polyhedron_relation_with_generator(+Handle, +Generator, -Relation)

ppl_Polyhedron_get_bounding_box(+Handle, +Complexity, -Box)

ppl_Polyhedron_is_empty(+Handle)

ppl_Polyhedron_is_universe(+Handle)

ppl_Polyhedron_is_bounded(+Handle)

ppl_Polyhedron_bounds_from_above(+Handle, +LinExpr)

ppl_Polyhedron_bounds_from_below(+Handle, +LinExpr)

ppl_Polyhedron_is_topologically_closed(+Handle)

ppl_Polyhedron_contains_Polyhedron(+Handle_1, +Handle_2)

ppl_Polyhedron_strictly_contains_Polyhedron(+Handle_1, +Handle_2)

ppl_Polyhedron_is_disjoint_from_Polyhedron(+Handle_1, +Handle_2)

ppl_Polyhedron_equals_Polyhedron(+Handle_1, +Handle_2)

ppl_Polyhedron_OK(+Handle)

ppl_Polyhedron_add_constraint(+Handle, +Constraint)

ppl_Polyhedron_add_constraint_and_minimize(+Handle, +Constraint)

ppl_Polyhedron_add_generator(+Handle, +Generator)

ppl_Polyhedron_add_generator_and_minimize(+Handle, +Generator)

ppl_Polyhedron_add_constraints(+Handle, +Constraint_System)

ppl_Polyhedron_add_constraints_and_minimize(+Handle, +Constraint_System)

ppl_Polyhedron_add_generators(+Handle, +Generator_System)

ppl_Polyhedron_add_generators_and_minimize(+Handle, +Generator_System)

ppl_Polyhedron_intersection_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_intersection_assign_and_minimize(+Handle_1, +Handle_2)

ppl_Polyhedron_poly_hull_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_poly_hull_assign_and_minimize(+Handle_1, +Handle_2)

ppl_Polyhedron_poly_difference_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_affine_image(+Handle, +PPL_Var, +LinExpr, +Integer)

ppl_Polyhedron_affine_preimage(+Handle, +PPL_Var, +LinExpr, +Integer)

ppl_Polyhedron_generalized_affine_image(+Handle, +PPL_Var, +Relation_Symbol, +LinExpr, +Integer)

ppl_Polyhedron_generalized_affine_image_lhs_rhs(+Handle, +LinExpr1, +Relation_Symbol, +LinExpr2)

ppl_Polyhedron_time_elapse_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_BHRZ03_widening_assign_with_token(+Handle_1, +Handle_2, ?Integer)

ppl_Polyhedron_BHRZ03_widening_assign(+Handle_1, +Handle_2) `ppl_Polyhedron_limited_BHRZ03_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System, ?Integer)`

`ppl_Polyhedron_limited_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)`

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System, ?Integer)`

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)`

`ppl_Polyhedron_H79_widening_assign_with_token(+Handle_1, +Handle_2, ?Integer)`

`ppl_Polyhedron_H79_widening_assign(+Handle_1, +Handle_2)`

`ppl_Polyhedron_limited_H79_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System, ?Integer)`

`ppl_Polyhedron_limited_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)`

`ppl_Polyhedron_bounded_H79_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System)`

`ppl_Polyhedron_bounded_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System, ?Integer)`

`ppl_Polyhedron_topological_closure_assign(+Handle)`

`ppl_Polyhedron_add_dimensions_and_embed(+Handle, +Integer)`

`ppl_Polyhedron_add_dimensions_and_project(+Handle, +Integer)`

`ppl_Polyhedron_concatenate_assign(+Handle1, +Handle2)`

`ppl_Polyhedron_remove_dimensions(+Handle, +List_of_PPL_Vars)`

`ppl_Polyhedron_remove_higher_dimensions(+Handle, +Integer))`

`ppl_Polyhedron_map_dimensions(+Handle, +P_Func))`

6.4.2.3 PPL Predicate Specifications

The PPL predicates provided by the Prolog interface are specified below. The specification uses the following grammar rules:

```
Topology      --> c | nnc

VarId         --> number | + number       variable identifier

PPL_Var       --> '$VAR'(VarId)           PPL variable

LinExpr       --> PPL_Var                 PPL variable
             | number
             | + LinExpr                  unary plus
             | - LinExpr                  unary minus
             | LinExpr + LinExpr          addition
             | LinExpr - LinExpr          subtraction
             | number * LinExpr           multiplication
             | LinExpr * number           multiplication

Relation_Symbol
              --> =                       equals
             | =<                         less than or equal
             | >=                         greater than or equal
```

```
           | <                         strictly less than
           | >                         strictly greater than

Denominator --> number
           | + number | - number       number must be non-zero

Constraint  --> LinExpr Relation_Symbol LinExpr
                                       constraint

Constraint_System                      list of constraints
            --> []
           | [Constraint | Constraint_System]

Generator   --> point(LinExpr)         point
           | point(LinExpr, Denominator)
                                       point
           | closure_point(LinExpr)    closure point
           | closure_point(LinExpr, Denominator)
                                       closure point
             (the point or closure point is defined by LinExpr/Denominator.)
           | ray(LinExpr)              ray
           | line(LinExpr)             line

Generator_System                       list of generators
            --> []
           | [Generator | Generator_System]

Atom        --> Prolog atom

Relation    --> is_disjoint            between a constraint and a polyhedron
           | strictly_intersects       between a constraint and a polyhedron
           | is_included               between a constraint and a polyhedron
           | saturates                 between a constraint and a polyhedron
           | subsumes                  between a generator and a polyhedron

Relation_List                          list of relations
            --> []
           | [Relation | Relation_List]


Complexity  --> polynomial | simplex | any

Rational_Numerator
            --> number | + number | - number

Rational_Denominator
            --> number                 number must be non-zero

Rational    --> Rational_Numerator     rational number
           | Rational_Numerator/Rational_Denominator

Bound       --> c(Rational)            closed rational limit
           | o(Rational)               open rational limit
           | o(pinf)                   unbounded in the positive direction
           | o(minf)                   unbounded in the negative direction

Interval    --> i(Bound, Bound)        rational interval

Box         --> []                     list of intervals
           | [Interval | Box]

Vars_Pair   --> PPLVar - PPLVar        map relation

P_Func      --> []                     list of map relations
           | [Vars_Pair | P_Func].
```

Below is a short description of each of the interface predicates. For full definitions of terminology used here, see Sections A Library for Convex Polyhedra, An Introduction to Convex Polyhedra, Representations of Convex Polyhedra and Operations on Convex Polyhedra of this manual.

`ppl_initialize` Initializes the PPL interface. Multiple calls to `ppl_initialize` does no harm.

`ppl_finalize` Finalizes the PPL interface. Once this is executed, the next call to an interface predicate must either be to `ppl_initialize` or to `ppl_finalize`. Multiple calls to `ppl_finalize` does no harm.

`ppl_set_timeout_exception_atom(+Atom)` Sets the atom to be thrown by timeout exceptions to `Atom`. The default value is `time_out`.

`ppl_timeout_exception_atom(?Atom)` The atom to be thrown by timeout exceptions is unified with `Atom`.

`ppl_set_timeout(+Integer)` Computations taking exponential time will be interrupted some time after `Integer` ms after that call. If the computation is interrupted that way, the current timeout exception atom will be thrown. `Integer` must be strictly greater than zero.

`ppl_reset_timeout` Resets the timeout time so that the computation is not interrupted.

`ppl_new_Polyhedron_from_dimension(+Topology, +Integer, -Handle)` Creates a new universe C or NNC polyhedron $\mathcal{P}$, depending on the value of `Topology`, with `Integer` dimensions. `Handle` is unified with the handle for $\mathcal{P}$. Thus the query

```
?- ppl_new_Polyhedron_from_dimension(c, 3, X).
```

creates the C polyhedron defining the 3-dimensional vector space $\mathbb{R}^3$ with `X` bound to a valid handle for accessing it.

`ppl_new_Polyhedron_empty_from_dimension(+Topology, +Integer, -Handle)` Creates a new empty C or NNC polyhedron $\mathcal{P}$, depending on the value of `Topology`, with `Integer` dimensions. `Handle` is unified with the handle for $\mathcal{P}$. Thus the query

```
?- ppl_new_Polyhedron_empty_from_dimension(nnc, 3, X).
```

creates an empty NNC polyhedron embedded in $\mathbb{R}^3$ with `X` bound to a valid handle for accessing it.

`ppl_new_Polyhedron_from_Polyhedron(+Topology_1, +Handle_1, +Topology_2, -Handle_2)` If `Handle_1` refers to a C or NNC polyhedron $\mathcal{P}_1$ (depending on the value of `Topology_1`), then this creates a copy $\mathcal{P}_2$ of $\mathcal{P}_1$ with topology C or NNC, depending on the value of `Topology_2`. `Handle_2` is unified with the handle for $\mathcal{P}_2$. Thus the query

```
?- ppl_new_Polyhedron_empty_from_dimension(nnc, 3, X),
   ppl_new_Polyhedron_from_Polyhedron(c, X, nnc, Y).
```

creates an empty C polyhedron embedded in $\mathbb{R}^3$ referenced by `X` and then makes a copy, converting the topology to an NNC polyhedron. with `Y` bound to a valid handle for accessing it.

When using `ppl_new_Polyhedron_from_Polyhedron/2`, when the source polyhedron is NNC and the copy is C, care must be taken that the source polyhedron referenced by `Handle1` is topologically closed.

`ppl_new_Polyhedron_from_constraints(+Topology, +Constraint_System, -Handle)` Creates a polyhedron $\mathcal{P}$ represented by `Constraint_System` with topology C or NNC, depending on the value of `Topology`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_Polyhedron_from_generators(+Topology, +Generator_System, -Handle)` Creates a polyhedron $\mathcal{P}$ represented by `Generator_System` with topology C or NNC, depending on the value of `Topology`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_Polyhedron_from_bounding_box(+Topology, +Box, -Handle)` Creates a polyhedron $\mathcal{P}$ represented by `Box` with topology C or NNC, depending on the value of `Topology`, and `Handle` is unified with the handle for $\mathcal{P}$. A bound of the form `o(Rational)` can be included in an interval in `Box` only if `Topology` is `nnc`.

`ppl_Polyhedron_swap(+Handle1, +Handle2)` Swaps the polyhedron referenced by `Handle1` with the one referenced by `Handle2`. The polyhedra $\mathcal{P}$ and $\mathcal{Q}$ must have the same topology.

`ppl_delete_Polyhedron(+Handle)` Deletes the polyhedron referenced by `Handle`. After execution, `Handle` is no longer a valid handle for a PPL polyhedron.

`ppl_Polyhedron_space_dimension(+Handle, -Integer)` Unifies the space dimension of the polyhedron referenced by `Handle` with `Integer`.

`ppl_Polyhedron_get_constraints(+Handle, -Constraint_System)` Unifies `Constraint_System` with a list of the constraints in the constraints system representing the polyhedron referenced by `Handle`.

`ppl_Polyhedron_get_minimized_constraints(+Handle, -Constraint_System)` Unifies `Constraint_System` with a minimized list of the constraints in the constraints system representing the polyhedron referenced by `Handle`.

`ppl_Polyhedron_get_generators(+Handle, -Generator_System)` Unifies `Generator_System` with a list of the generators in the generators system representing the polyhedron referenced by `Handle`.

`ppl_Polyhedron_get_minimized_generators(+Handle, -Generator_System)` Unifies `Generator_System` with a minimized list of the generators in the generators system representing the polyhedron referenced by `Handle`.

`ppl_Polyhedron_relation_with_constraint(+Handle, +Constraint, -Relation_List)` Unifies `Relation_List` with the list of relations the polyhedron referenced by `Handle` has with `Constraint`. The possible relations are listed in the grammar rules above; their meaning is given in Section Operations on Convex Polyhedra.

`ppl_Polyhedron_relation_with_generator(+Handle, +Generator, -Relation_List)` Unifies `Relation_List` with the list of relations the polyhedron referenced by `Handle` has with `Generator`. The possible relations are listed in the grammar rules above; their meaning is given in Section Operations on Convex Polyhedra.

`ppl_Polyhedron_get_bounding_box(+Handle, +Complexity, -Box)` Succeeds if and only if the bounding box of the polyhedron referenced by `Handle` unifies with the box defined by `Box`. E.g.,

```
?- A = '$VAR'(0), B = '$VAR'(1),
   ppl_new_Polyhedron_from_constraints(nnc, [B > 0, 4*A =< 2], X),
   ppl_Polyhedron_get_bounding_box(X, any, Box).

Box = [i(o(minf), c(1/2)), i(o(0), o(pinf))].
```

Note that the rational numbers in `Box` are in canonical form. E.g., the following will fail:

```
?- A = '$VAR'(0), B = '$VAR'(1),
   ppl_new_Polyhedron_from_constraints(nnc, [B > 0, 4*A =< 2], X),
   ppl_Polyhedron_get_bounding_box(X, any, Box),
   Box = [i(o(minf), c(2/4)), i(o(0), o(pinf))].
```

The complexity class `Complexity` determining the algorithm to be used has the following meaning:

- `polynomial` allows code of the worst-case polynomial complexity class;
- `simplex` allows code of the worst-case exponential but typically polynomial complexity class;
- `any` allows code of the universal complexity class.

`ppl_Polyhedron_is_empty(+Handle)` Succeeds if and only if the polyhedron referenced by `Handle` is empty.

`ppl_Polyhedron_is_universe(+Handle)` Succeeds if and only if the polyhedron referenced by `Handle` is the universe.

`ppl_Polyhedron_is_bounded(+Handle)` Succeeds if and only if the polyhedron referenced by `Handle` is bounded.

`ppl_Polyhedron_bounds_from_above(+Handle, +LinExpr)` Succeeds if and only if `LinExpr` is bounded from above in the polyhedron referenced by `Handle`.

`ppl_Polyhedron_bounds_from_below(+Handle, +LinExpr)` Succeeds if and only if `LinExpr` is bounded from below in the polyhedron referenced by `Handle`.

`ppl_Polyhedron_is_topologically_closed(+Handle)` Succeeds if and only if the polyhedron referenced by `Handle` is topologically closed.

`ppl_Polyhedron_contains_Polyhedron(+Handle_1, +Handle_2)` Succeeds if and only if the polyhedron referenced by `Handle_1` is included in or equal to the polyhedron referenced by `Handle_2`.

`ppl_Polyhedron_strictly_contains_Polyhedron(+Handle_1, +Handle_2)` Succeeds if and only if the polyhedron referenced by `Handle_1` is included in but not equal to the polyhedron referenced by `Handle_2`.

`ppl_Polyhedron_is_disjoint_from_Polyhedron(+Handle_1, +Handle_2)` Succeeds if and only if the polyhedron referenced by `Handle_1` is disjoint from the polyhedron referenced by `Handle_2`.

`ppl_Polyhedron_equals_Polyhedron(+Handle_1, +Handle_2)` Succeeds if and only if the polyhedron referenced by `Handle_1` is equal to the polyhedron referenced by `Handle_2`.

`ppl_Polyhedron_OK(+Handle)` Succeeds only if the polyhedron referenced by `Handle` is well formed, i.e., if it satisfies all its implementation invariants. Useful for debugging purposes.

`ppl_Polyhedron_add_constraint(+Handle, +Constraint)`

`ppl_Polyhedron_add_constraint_and_minimize(+Handle, +Constraint)` Updates the polyhedron referenced by `Handle` to one obtained by adding `Constraint` to its constraint system. Thus, the query

```
?- ppl_new_Polyhedron_from_dimension(c, 3, X),
   A = '$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_add_constraint(X, 4*A + B - 2*C >= 5).
```

will update the polyhedron with handle `X` to consist of the set of points in the vector space $\mathbb{R}^3$ satisfying the constraint $4x + y - 2z >= 5$.

Note that `ppl_Polyhedron_add_constraint_and_minimize/2` will fail if, after adding the constraint, the polyhedron is empty.

`ppl_Polyhedron_add_generator(+Handle, +Generator)`

`ppl_Polyhedron_add_generator_and_minimize(+Handle, +Generator)` Updates the polyhedron referenced by `Handle` to one obtained by adding `Generator` to its generator system. Thus, after the query

```
?- ppl_new_Polyhedron_from_dimension(c, 3, X),
   A = '$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_add_generator(X, point(-100*A - 5*B, 8)).
```

will update the polyhedron with handle `X` to be the single point $(-12.5, -0.625, 0)^{\mathrm{T}}$ in the vector space $\mathbb{R}^3$.

`ppl_Polyhedron_add_constraints(+Handle, +Constraint_System)` Updates the polyhedron referenced by `Handle` to one obtained by adding to its constraint system the constraints in `Constraint_System`. E.g.,

```
| ?- ppl_new_Polyhedron_from_dimension(c, 2, X),
     A = '$VAR'(0), B = '$VAR'(1),
     ppl_Polyhedron_add_constraints(X, [4*A + B >= 3, A = 1]),
     ppl_Polyhedron_get_constraints(X, CS).

CS = [4*A+1*B>=3,1*A=1] ?
```

The updated polyhedron referenced by `Handle` can be empty and a query will succeed even when `Constraint_System` is unsatisfiable.

`ppl_Polyhedron_add_constraints_and_minimize(+Handle, +Constraint_System)` Updates the polyhedron referenced by `Handle` to one obtained by adding to its constraint system the constraints in `Constraint_System`. E.g.,

```
?- ppl_new_Polyhedron_from_dimension(c, 2, X),
   A = '$VAR'(0), B = '$VAR'(1),
   ppl_Polyhedron_add_constraints_and_minimize(X, [4*A + B >= 3, A = 1]),
   ppl_Polyhedron_get_constraints(X, CS).

CS = [1*B>= -1,1*A=1]
```

This will fail if, after adding the constraints, the polyhedron is empty. E.g., the following will fail,

```
?- A = '$VAR'(0), B = '$VAR'(1),
   ppl_new_Polyhedron_from_dimension(c, 2, X),
   ppl_Polyhedron_add_constraints_and_minimize(X,
     [4*A + B >= 3, A = 0, B =< 0]),
   ppl_Polyhedron_get_constraints(X, CS).
```

`ppl_Polyhedron_add_generators(+Handle, +Generator_System)` Updates the polyhedron referenced by `Handle` to one obtained by adding to its generator system the generators in `Generator_System`.

If the system of generators representing a polyhedron is non-empty, then it must include a point (see the paragraph on generator representation in Section Representations of Convex Polyhedra). Thus care must be taken to ensure that, before calling this predicate, either the polyhedron referenced by `Handle` is non-empty or that whenever `Generator_System` is non-empty the first element defines a point. E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 3, X),
   A='$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_add_generators(X,
     [point(1*A + 1*B + 1*C, 1), ray(1*A), ray(2*A)]),
   ppl_Polyhedron_get_generators(X, GS).

GS = [ray(2*A), point(1*A+1*B+1*C), ray(1*A)]
```

`ppl_Polyhedron_add_generators_and_minimize(+Handle, +Generator_System)` Updates the polyhedron referenced by `Handle` to one obtained by adding to its generator system the generators in `Generator_System`.

Unlike the predicate `ppl_add_generators`, the order of the generators in `Generator_System` is not important. E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 3, X),
   A='$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_add_generators_and_minimize(X,
     [ray(1*A), ray(2*A), point(1*A + 1*B + 1*C, 1)]),
   ppl_Polyhedron_get_generators(X, GS).

GS = [point(1*A+1*B+1*C), ray(1*A)]
```

`ppl_Polyhedron_intersection_assign(+Handle_1, +Handle_2)`

`ppl_Polyhedron_intersection_assign_and_minimize(+Handle_1, +Handle_2)` Assigns to the polyhedron referenced by `Handle_1` its intersection with the polyhedra referenced by `Handle_2`.

`ppl_Polyhedron_poly_hull_assign(+Handle_1, +Handle_2)`

`ppl_Polyhedron_poly_hull_assign_and_minimize(+Handle_1, +Handle_2)` Assigns to the polyhedron referenced by `Handle_1` its poly-hull with the polyhedra referenced by `Handle_2`.

`ppl_Polyhedron_poly_difference_assign(+Handle_1, +Handle_2)` Assigns to the polyhedron referenced by `Handle_1` its poly-difference with the polyhedron referenced by `Handle_2`.

`ppl_Polyhedron_affine_image(+Handle, +PPL_Var, +LinExpr, +Integer)` Transforms the polyhedron referenced by `Handle` assigning the affine expression `LinExpr`/`Integer` to `PPL_Var`.

`ppl_Polyhedron_affine_preimage(+Handle, +PPL_Var, +LinExpr, +Integer)` This is the inverse transformation to that for `ppl_affine_image`.

`ppl_Polyhedron_generalized_affine_image(+Handle, +PPL_Var, +Relation_-Symbol +LinExpr, +Integer)` Transforms the polyhedron referenced by `Handle` assigning the generalized affine image with respect to the transfer function `PPL_Var Relation_Symbol LinExpr`/`Integer`.

`ppl_Polyhedron_generalized_affine_image_lhs_rhs(+Handle, +LinExpr1, +Relation_Symbol +LinExpr2)` Transforms the polyhedron referenced by `Handle` assigning the generalized affine image with respect to the transfer function `LinExpr1 Relation_Symbol LinExpr2`.

`ppl_Polyhedron_time_elapse_assign(+Handle_1, +Handle_2)` Assigns to the polyhedron $\mathcal{P}$ referenced by `Handle_1` the time-elapse ($\mathcal{P} \nearrow \mathcal{Q}$) with the polyhedra $\mathcal{Q}$ referenced by `Handle_-2`.

`ppl_Polyhedron_BHRZ03_widening_assign_with_token(+Handle_1, +Handle_2, ?Integer)` The polyhedra referenced by `Handle_1` and `Handle_2` are unaltered. The token `Integer` is 0 if a BHRZ03 widening would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_BHRZ03_widening_assign(+Handle_1, +Handle_2)` Assigns to the polyhedron referenced by `Handle_1` its BHRZ03-widening with the polyhedra referenced by `Handle_2`.

`ppl_Polyhedron_limited_BHRZ03_extrapolation_assign_with_token( +Handle_1, +Handle_2, +Constraint_System, ?Integer)` The polyhedra referenced by `Handle_1` and `Handle_2` are unaltered. The token `Integer` is 0 if a BHRZ03-widening with the polyhedra referenced by `Handle_2`, improved by enforcing those constraints in `Constraint_System` would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_limited_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` Assigns to the polyhedron $\mathcal{P}$ referenced by `Handle_1` the result of its BHRZ03-widening with the polyhedra referenced by `Handle_2`, improved by enforcing those constraints in `Constraint_System`.

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign_with_token( +Handle_1, +Handle_2, +Constraint_System, ?Integer)` The polyhedron $\mathcal{P}_1$ and $\mathcal{P}_2$ referenced by `Handle_1` and `Handle_2`, respectively are unaltered. The token `Integer` is 0 if a BHRZ03-widening with $\mathcal{P}_2$, improved by enforcing all the constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by all the points of $\mathcal{P}_1$ together with the constraints in `Constraint_System` would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` Assigns to the polyhedron $\mathcal{P}$ referenced by `Handle_1` the result of its BHRZ03-widening with the polyhedra referenced by `Handle_2` improved by enforcing all the constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by all the points of $\mathcal{P}$ together with the constraints in `Constraint_System`.

`ppl_Polyhedron_H79_widening_assign_with_token(+Handle_1, +Handle_2, ?Integer)` The polyhedra referenced by `Handle_1` and `Handle_2` are unaltered. The token `Integer` is 0 if an H79 widening would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_H79_widening_assign(+Handle_1, +Handle_2)` Assigns to the polyhedron referenced by `Handle_1` its H79-widening with the polyhedra referenced by `Handle_2`.

`ppl_Polyhedron_limited_H79_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System, ?Integer)` The polyhedra referenced by `Handle_1` and `Handle_2` are unaltered. The token `Integer` is 0 if a H79-widening with the polyhedra referenced by `Handle_2`, improved by enforcing those constraints in `Constraint_System` would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_limited_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` Assigns to the polyhedron $\mathcal{P}$ referenced by `Handle_1` its H79-widening with the polyhedra referenced by `Handle_2`, improved by enforcing those constraints in `Constraint_System`.

`ppl_Polyhedron_bounded_H79_extrapolation_assign_with_token(+Handle_1, +Handle_2, +Constraint_System, ?Integer)` The polyhedron $\mathcal{P}_1$ and $\mathcal{P}_2$ referenced by `Handle_1` and `Handle_2`, respectively are unaltered. The token `Integer` is 0 if a H79-widening with $\mathcal{P}_2$, improved by enforcing all the constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by all the points of $\mathcal{P}_1$ together with the constraints in `Constraint_System` would have changed the polyhedron referenced by `Handle_1` and is 1 otherwise.

`ppl_Polyhedron_bounded_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` Assigns to the polyhedron $\mathcal{P}$ referenced by `Handle_1` the result of its H79-widening with the polyhedra referenced by `Handle_2` improved by enforcing all the constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by all the points of $\mathcal{P}$ together with the constraints in `Constraint_System`.

`ppl_Polyhedron_topological_closure_assign(+Handle)` Assigns to the polyhedron referenced by `Handle` its topological closure.

`ppl_Polyhedron_add_dimensions_and_embed(+Handle, +Integer)` Embeds the polyhedron referenced by `Handle` in a space that is enlarged by `Integer` dimensions, E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 0, X),
   ppl_Polyhedron_add_dimensions_and_embed(X, 2),
   ppl_Polyhedron_get_constraints(X, CS),
   ppl_Polyhedron_get_generators(X, GS).

CS = [],
GS = [point(0),line(1*A),line(1*B)]
```

`ppl_Polyhedron_add_dimensions_and_project(+Handle, +Integer)` Projects the polyhedron referenced by `Handle` onto a space that is enlarged by `Integer` dimensions, E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 0, X),
   ppl_Polyhedron_add_dimensions_and_project(X, 2),
   ppl_Polyhedron_get_constraints(X, CS),
   ppl_Polyhedron_get_generators(X, GS).

CS = [1*A = 0, 1*B = 0],
GS = [point(0)]
```

`ppl_Polyhedron_concatenate_assign(+Handle1, +Handle2)` Updates the polyhedron $\mathcal{P}_1$ referenced by `Handle1` by first embedding $\mathcal{P}_1$ in a new space enlarged by the space dimensions of the polyhedron $\mathcal{P}_2$ referenced by `Handle2`, and then adds to its system of constraints a renamed-apart version of the constraints of $\mathcal{P}_2$.

E.g.,

```
?- ppl_new_Polyhedron_from_dimension(nnc, 2, X),
   A = '$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   D = '$VAR'(3), E = '$VAR'(4),
   ppl_new_Polyhedron_from_constraints(nnc, [A > 1, B >= 0, C >= 0], Y),
   ppl_Polyhedron_concatenate_assign(X, Y),
   ppl_Polyhedron_get_constraints(X, CS).

CS = [1*C > 1, 1*D >= 0, 1*E >= 0]
```

`ppl_Polyhedron_remove_dimensions(+Handle, +List_of_PPL_Vars)` Removes the dimensions given by the identifiers of the PPL variables in list `List_of_PPL_Vars` from the polyhedron referenced by `Handle`. The identifiers for the remaining PPL variables are renumbered so that they are consecutive and the maximum index is less than the number of dimensions. E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 3, X),
   A='$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_remove_dimensions(X, [B]),
   ppl_Polyhedron_space_dimension(X, K),
   ppl_Polyhedron_get_generators(X, GS).

K = 2,
GS = [point(0),line(1*A),line(1*B),line(0)]
```

`ppl_Polyhedron_remove_higher_dimensions(+Handle, +Integer))` Projects the the polyhedron referenced to by `Handle` onto the first `Integer` dimension. E.g.,

```
?- ppl_new_Polyhedron_empty_from_dimension(c, 5, X),
   ppl_Polyhedron_remove_higher_dimensions(X, 3),
   ppl_Polyhedron_space_dimension(X, K).
```

`ppl_Polyhedron_map_dimensions(+Handle, +P_Func))` Maps the dimensions of the polyhedron referenced by `Handle` using the partial function defined by `P_Func`. The result is undefined if `P_Func` does not encode a partial function with the properties described in the specification of the mapping operator.

6.4.3 Compilation and Installation

When the Parma Polyhedra Library is configured, it tests for the existence of each supported Prolog system. If a supported Prolog system is correctly installed in a standard location, things are arranged so that the corresponding interface is built and installed.

In the sequel, `prefix` is the prefix under which you have installed the library (typically `/usr` or `/usr/local`).

As an option, the Prolog interface can track the creation and disposal of polyhedra. In fact, differently from native Prolog data, PPL polyhedra must be explicitly disposed and forgetting to do so is a very common mistake. To enable this option, configure the library adding `-DPROLOG_TRACK_ALLOCATION` to the options passed to the C++ compiler. Your configure command would then look like

```
path/to/configure --with-cxxflags="-DPROLOG_TRACK_ALLOCATION" ...
```

6.4.4 System-Dependent Features

CIAO Prolog Support for CIAO Prolog is under development and will be available in a future release. Only Ciao Prolog 1.9 #44 or later is supported.

GNU Prolog The GNU Prolog interface to the PPL library is available both as "PPL enhanced" GNU Prolog interpreter and as a library that can be linked to GNU Prolog programs. Only GNU Prolog version 1.2.12 or later is supported.

Notice that GNU Prolog version 1.2.12 suffers from a serious limitation as far as foreign code is concerned. In order to be safe you must configure GNU Prolog with the

--disable-ebp option (note that this has a negative effect on performance). See http://www.cs.unipr.it/pipermail/ppl-devel/2002-June/001777.html, http://www.cs.unipr.it/pipermail/ppl-devel/2002-June/001780.html, http://www.cs.unipr.it/pipermail/ppl-devel/2002-June/001788.html and http://www.cs.unipr.it/pipermail/ppl-devel/2002-June/001789.html for more information.

We have experienced other serious problems with the GNU Prolog interface, up to and including GNU Prolog version 1.2.16: see http://www.cs.unipr.it/pipermail/ppl-devel/2002-October/002657.html for more information.

The ppl_gprolog Executable If an appropriate version of GNU Prolog is installed on the machine on which you compiled the library, the command make install will install the executable ppl_gprolog in the directory prefix/bin. The ppl_gprolog executable is simply the GNU Prolog interpreter with the Parma Polyhedra library linked in. The only thing you should do to use the library is to call ppl_initialize/0 before any other PPL predicate and to call ppl_finalize/0 when you are done with the library.

Linking the Library To GNU Prolog Programs In order to allow linking GNU Prolog programs to the PPL, the following files are installed in the directory prefix/lib/ppl: ppl_gprolog.pl contains the required foreign declarations; libppl_gprolog.* contain the executable code for the GNU Prolog interface in various formats (static library, shared library, libtool library). If your GNU Prolog program is constituted by, say, source1.pl and source2.pl and you want to create the executable myprog, your compilation command may look like

```
gplc -o myprog prefix/lib/ppl/ppl_gprolog.pl source1.pl source2.pl \
  -L '-Lprefix/lib/ppl -lppl_gprolog -Lprefix/lib -lppl -lgmpxx -lgmp -lstdc++'
```

SICStus Prolog The SICStus Prolog interface to the PPL library is available both as a statically linked module or as a dynamically linked one. Only SICStus Prolog version 3.9.0 or later is supported.

The Statically Linked ppl_sicstus Executable If an appropriate version of SICStus Prolog is installed on the machine on which you compiled the library, the command make install will install the executable ppl_sicstus in the directory prefix/bin. The ppl_sicstus executable is simply the SICStus Prolog system with the Parma Polyhedra library statically linked. The only thing you should do to use the library is to load prefix/lib/ppl/ppl_sicstus.pl.

Loading the SICStus Interface Dynamically In order to dynamically load the library from SICStus Prolog you should simply load prefix/lib/ppl/ppl_sicstus.pl. Notice that, for dynamic linking to work, you should have configured the library with the --enable-shared option.

SWI-Prolog The SWI-Prolog interface of the library is available both as a statically linked module or as a dynamically linked one. Only SWI-Prolog version 5.0 or later is supported.

The ppl_pl Executable If an appropriate version of SWI-Prolog is installed on the machine on which you compiled the library, the command make install will install the executable ppl_pl in the directory prefix/bin. The ppl_pl executable is simply the SWI-Prolog shell with the Parma Polyhedra library statically linked: from within ppl_pl all the services of the library are available without further action.

Loading the SWI-Prolog Interface Dynamically In order to dynamically load the library from SWI-Prolog you should simply load `prefix/lib/ppl/ppl_swiprolog.pl`. This will invoke `ppl_initialize/0` automatically but, at least for SWI-Prolog versions up to 5.0.7, it is the programmer's responsibility to call `ppl_finalize/0`. Alternatively, you can load the library directly with

```
:- load_foreign_library('prefix/lib/ppl/libppl_swiprolog').
```

This will call `ppl_initialize/0` automatically. Analogously,

```
:- unload_foreign_library('prefix/lib/ppl/libppl_swiprolog').
```

will, as part of the unload process, invoke `ppl_finalize/0`.

Notice that, for dynamic linking to work, you should have configured the library with the `--enable-shared` option.

XSB The XSB Prolog interface to the PPL library is available as a dynamically linked module. Only CVS versions of XSB from August 2002 onward are supported. See http://www.cs.unipr.it/pipermail/ppl-devel/2002-July/002201.html for information about a bug in XSB 2.5 that has bitten several people.

In order to dynamically load the library from XSB you should load the `ppl_xsb` module and import the predicates you need. For things to work, you may have to copy the files `prefix/lib/ppl/ppl_xsb.O` and `prefix/lib/ppl/ppl_xsb.so` in your current directory or in one of the XSB library directories.

YAP The YAP Prolog interface to the PPL library is available as a dynamically linked module. Only YAP version 4.4 or later is supported.

In order to dynamically load the library from YAP you should simply load `prefix/lib/ppl/ppl_yap.pl`. This will invoke `ppl_initialize/0` automatically; it is the programmer's responsibility to call `ppl_finalize/0` when the PPL library is no longer needed. Notice that, for dynamic linking to work, you should have configured the library with the `--enable-shared` option.

6.5 PPL License Pages

6.5.1 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other

Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The ”Program”, below, refers to any such program or work, and a ”work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term ”modification”.) Each licensee is addressed as ”you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

- **b)** You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- **c)** If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- **a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- **b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- **c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and ”any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM ”AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the ”copyright” line and a pointer to where the full notice is found.

```
 one line to give the program's name and an idea of what it does.
Copyright (C)  yyyy   name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C)  year  name of author
```

```
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type ‘show w’.  This is free software, and you are welcome
to redistribute it under certain conditions; type ‘show c’
for details.
```

The hypothetical commands `‘show w’` and `‘show c’` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `‘show w’` and `‘show c’`; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ”copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program ‘Gnomovision’
(which makes passes at compilers) written
by James Hacker.

 signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

7 PPL Namespace Documentation

7.1 Parma_Polyhedra_Library Namespace Reference

The entire library is confined into this namespace.

Compounds

- class Variable

  A dimension of the space.

- struct Variable::Compare

  Binary predicate defining the total ordering on variables.

- class LinExpression

  A linear expression.

- class Constraint

  A linear equality or inequality.

- class Generator

  A line, ray, point or closure point.

- class Poly_Con_Relation

  The relation between a polyhedron and a constraint.

- class Poly_Gen_Relation

  The relation between a polyhedron and a generator.

- class Polyhedron

  The base class for convex polyhedra.

- class C_Polyhedron

  A closed convex polyhedron.

- class NNC_Polyhedron

  A not necessarily closed convex polyhedron.

- class Determinate

  Wrap a polyhedron class into a determinate constraint system interface.

- class PowerSet

  The powerset construction on constraint systems.

Typedefs

- typedef mpz_class **Integer**

  See the GMP's manual available at `http://swox.com/gmp/` .

- typedef std::set< Variable, Variable::Compare > **Variables_Set**

  An std::set containing variables in increasing order of dimension index.

Functions

- const char ∗ **version** ()

  Returns a character string containing the PPL version.

- const char ∗ **banner** ()

  Returns a character string containing information about the PPL version, the licensing, the lack of any warranty whatsoever, the C++ compiler used to build the library, where to report bugs and where to look for further information.

- template<typename PH> std::pair< PH, PowerSet< Determinate< NNC_Polyhedron > > > linear_partition (const PH &p, const PH &q)

  Partitions `q` *with respect to* `p`.

7.1.1 Detailed Description

The entire library is confined into this namespace.

7.1.2 Function Documentation

7.1.2.1 template<typename PH> std::pair< PH, PowerSet< Determinate< NNC_Polyhedron > > > Parma_Polyhedra_Library::linear_partition (const PH & *p*, const PH & *q*)

Partitions `q` with respect to `p`.

Let `p` and `q` be two polyhedra. The function returns an object `r` of type `std::pair<PH, PowerSet<Determinate<NNC_Polyhedron> > >` such that

- `r.first` is the intersection of `p` and `q`;
- `r.second` has the property that all its elements are not empty, pairwise disjoint, and disjoint from `p`;
- the union of `r.first` with all the elements of `r.second` gives `q` (i.e., `r` is the representation of a partition of `q`).

7.2 Parma_Polyhedra_Library::IO_Operators Namespace Reference

All input/output operators are confined into this namespace.

7.2.1 Detailed Description

All input/output operators are confined into this namespace.

This is done so that the library's input/output operators do not interfere with those the user might want to define. In fact, it is highly unlikely that any pre-defined I/O operator will suit the needs of a client application. On the other hand, those applications for which the PPL I/O operator are enough can easily obtain access to them. For example, a directive like

```
using namespace Parma_Polyhedra_Library::IO_Operators;
```

would suffice for most uses. In more complex situations, such as

```
const ConSys& cs = ...;
copy(cs.begin(), cs.end(),
     ostream_iterator<Constraint>(cout, "\n"));
```

the Parma_Polyhedra_Library namespace must be suitably extended. This can be done as follows:

```
namespace Parma_Polyhedra_Library {
  // Import all the output operators into the main PPL namespace.
  using IO_Operators::operator<<;
}
```

7.3 std Namespace Reference

The standard C++ namespace.

7.3.1 Detailed Description

The standard C++ namespace.

The Parma Polyhedra Library conforms to the C++ standard and, in particular, as far as reserved names are concerned (17.4.3.1, [lib.reserved.names]). The PPL, however, defines several template specializations for the standard library templates swap() and iter_swap() (25.2.2, [lib.alg.swap]).

8 PPL Class Documentation

8.1 C_Polyhedron Class Reference

A closed convex polyhedron.

Inherits Polyhedron.

Public Member Functions

- C_Polyhedron (dimension_type num_dimensions=0, Degenerate_Kind kind=UNIVERSE)

  Builds either the universe or the empty C polyhedron.

- C_Polyhedron (const ConSys &cs)

  Builds a C polyhedron from a system of constraints.

- C_Polyhedron (ConSys &cs)

  Builds a C polyhedron recycling a system of constraints.

- C_Polyhedron (const GenSys &gs)

  Builds a C polyhedron from a system of generators.

- C_Polyhedron (GenSys &gs)

  Builds a C polyhedron recycling a system of generators.

- C_Polyhedron (const NNC_Polyhedron &y)

  Builds a C polyhedron from the NNC polyhedron `y`.

- template<typename Box> C_Polyhedron (const Box &box, From_Bounding_Box dummy)

  Builds a C polyhedron out of a generic, interval-based bounding box.

- **C_Polyhedron** (const C_Polyhedron &y)

  Ordinary copy-constructor.

- C_Polyhedron & **operator=** (const C_Polyhedron &y)

  The assignment operator. (`*this` *and* `y` *can be dimension-incompatible.).*

- **~C_Polyhedron** ()

  Destructor.

8.1.1 Detailed Description

A closed convex polyhedron.

An object of the class C_Polyhedron represents a *topologically closed* convex polyhedron in the vector space $\mathbb{R}^n$.

When building a closed polyhedron starting from a system of constraints, an exception is thrown if the system contains a *strict inequality* constraint. Similarly, an exception is thrown when building a closed polyhedron starting from a system of generators containing a *closure point*.

Note:

Such an exception will be obtained even if the system of constraints (resp., generators) actually defines a topologically closed subset of the vector space, i.e., even if all the strict inequalities (resp., closure points) in the system happen to be redundant with respect to the system obtained by removing all the strict inequality constraints (resp., all the closure points). In contrast, when building a closed polyhedron starting from an object of the class NNC_Polyhedron, the precise topological closure test will be performed.

8.1.2 Constructor & Destructor Documentation

8.1.2.1 C_Polyhedron::C_Polyhedron (dimension_type *num_dimensions* = 0, Degenerate_Kind *kind* = UNIVERSE) `[explicit]`

Builds either the universe or the empty C polyhedron.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the C polyhedron.

kind Specifies whether a universe or an empty C polyhedron should be built.

Both parameters are optional: by default, a 0-dimension space universe C polyhedron is built.

8.1.2.2 C_Polyhedron::C_Polyhedron (const ConSys & *cs*)

Builds a C polyhedron from a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of constraints contains strict inequalities.

8.1.2.3 C_Polyhedron::C_Polyhedron (ConSys & *cs*)

Builds a C polyhedron recycling a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of constraints contains strict inequalities.

8.1.2.4 C_Polyhedron::C_Polyhedron (const GenSys & *gs*)

Builds a C polyhedron from a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

gs The system of generators defining the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of generators is not empty but has no points, or if it contains closure points.

8.1.2.5 C_Polyhedron::C_Polyhedron (GenSys & *gs*)

Builds a C polyhedron recycling a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

gs The system of generators defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of generators is not empty but has no points, or if it contains closure points.

8.1.2.6 C_Polyhedron::C_Polyhedron (const NNC_Polyhedron & *y*) `[explicit]`

Builds a C polyhedron from the NNC polyhedron `y`.

Exceptions:

std::invalid_argument thrown if the polyhedron `y` is not topologically closed.

8.1.2.7 template<typename Box> C_Polyhedron::C_Polyhedron (const Box & *box*, From_Bounding_Box *dummy*)

Builds a C polyhedron out of a generic, interval-based bounding box.

For a description of the methods that should be provided by the template class Box, see the documentation of the protected method: template <typename Box> Polyhedron::Polyhedron(Topology topol, const Box& box);

Parameters:

box The bounding box representing the polyhedron to be built.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions:

std::invalid_argument thrown if `box` has intervals that are not topologically closed (i.e., having some finite but open bounds).

8.2 Constraint Class Reference

A linear equality or inequality.

Public Types

- enum Type { EQUALITY, NONSTRICT_INEQUALITY, STRICT_INEQUALITY }

  The constraint type.

Public Member Functions

- **Constraint** (const Constraint &c)

  Ordinary copy-constructor.

- ∼**Constraint** ()

  Destructor.

- Constraint & **operator=** (const Constraint &c)

  Assignment operator.

- dimension_type **space_dimension** () const

  Returns the dimension of the vector space enclosing ∗`this`.

- Type **type** () const

  Returns the constraint type of ∗`this`.

- bool **is_equality** () const

  Returns `true` *if and only if* ∗`this` *is an equality constraint.*

- bool **is_inequality** () const

  Returns `true` *if and only if* ∗`this` *is an inequality constraint (either strict or non-strict).*

- bool **is_nonstrict_inequality** () const

  Returns `true` *if and only if* ∗`this` *is a non-strict inequality constraint.*

- bool **is_strict_inequality** () const

  Returns `true` *if and only if* ∗`this` *is a strict inequality constraint.*

- const Integer & coefficient (Variable v) const

  Returns the coefficient of `v` *in* ∗`this`.

- const Integer & **inhomogeneous_term** () const

  Returns the inhomogeneous term of ∗`this`.

- bool **OK** () const

  Checks if all the invariants are satisfied.

Static Public Member Functions

- const Constraint & **zero_dim_false** ()

  The unsatisfiable (zero-dimension space) constraint $0 = 1$.

- const Constraint & **zero_dim_positivity** ()

  The true (zero-dimension space) constraint $0 \leq 1$, *also known as* positivity constraint.

Related Functions

(Note that these are not member functions.)

- std::ostream & **operator<<** (std::ostream &s, const Constraint &c)

  Output operator.

- Constraint **operator==** (const LinExpression &e1, const LinExpression &e2)

  Returns the constraint `e1 = e2`.

- Constraint **operator==** (const LinExpression &e, const Integer &n)

  Returns the constraint `e = n`.

- Constraint **operator==** (const Integer &n, const LinExpression &e)

  Returns the constraint `n = e`.

- Constraint **operator<=** (const LinExpression &e1, const LinExpression &e2)

  Returns the constraint `e1 <= e2`.

- Constraint **operator<=** (const LinExpression &e, const Integer &n)

  Returns the constraint `e <= n`.

- Constraint **operator<=** (const Integer &n, const LinExpression &e)

  Returns the constraint `n <= e`.

- Constraint **operator>=** (const LinExpression &e1, const LinExpression &e2)

  Returns the constraint `e1 >= e2`.

- Constraint **operator>=** (const LinExpression &e, const Integer &n)

  Returns the constraint `e >= n`.

- Constraint **operator>=** (const Integer &n, const LinExpression &e)

  Returns the constraint `n >= e`.

- Constraint **operator<** (const LinExpression &e1, const LinExpression &e2)

  Returns the constraint `e1 < e2`.

- Constraint **operator<** (const LinExpression &e, const Integer &n)

  Returns the constraint `e < n`.

- Constraint **operator<** (const Integer &n, const LinExpression &e)

Returns the constraint `n` $<$ `e`.

- Constraint **operator**$>$ (const LinExpression &e1, const LinExpression &e2)

  Returns the constraint `e1` $>$ `e2`.

- Constraint **operator**$>$ (const LinExpression &e, const Integer &n)

  Returns the constraint `e` $>$ `n`.

- Constraint **operator**$>$ (const Integer &n, const LinExpression &e)

  Returns the constraint `n` $>$ `e`.

- void **swap** (Parma_Polyhedra_Library::Constraint &x, Parma_Polyhedra_Library::Constraint &y)

  Specializes `std::swap`.

8.2.1 Detailed Description

A linear equality or inequality.

An object of the class Constraint is either:

- an equality: $\sum_{i=0}^{n-1} a_i x_i + b = 0$;
- a non-strict inequality: $\sum_{i=0}^{n-1} a_i x_i + b \geq 0$; or
- a strict inequality: $\sum_{i=0}^{n-1} a_i x_i + b > 0$;

where n is the dimension of the space, a_i is the integer coefficient of variable x_i and b is the integer inhomogeneous term.

How to build a constraint

Constraints are typically built by applying a relation symbol to a pair of linear expressions. Available relation symbols are equality (`==`), non-strict inequalities (`>=` and `<=`) and strict inequalities (`<` and `>`). The space-dimension of a constraint is defined as the maximum space-dimension of the arguments of its constructor.

In the following examples it is assumed that variables `x`, `y` and `z` are defined as follows:

```
Variable x(0);
Variable y(1);
Variable z(2);
```

Example 1

The following code builds the equality constraint $3x + 5y - z = 0$, having space-dimension 3:

```
Constraint eq_c(3*x + 5*y - z == 0);
```

The following code builds the (non-strict) inequality constraint $4x \geq 2y - 13$, having space-dimension 2:

```
Constraint ineq_c(4*x >= 2*y - 13);
```

The corresponding strict inequality constraint $4x > 2y - 13$ is obtained as follows:

```
Constraint strict_ineq_c(4*x > 2*y - 13);
```

An unsatisfiable constraint on the zero-dimension space $\mathbb{R}^0$ can be specified as follows:

```
Constraint false_c = Constraint::zero_dim_false();
```

Equivalent, but more involved ways are the following:

```
Constraint false_c1(LinExpression::zero() == 1);
Constraint false_c2(LinExpression::zero() >= 1);
Constraint false_c3(LinExpression::zero() > 0);
```

In contrast, the following code defines an unsatisfiable constraint having space-dimension 3:

```
Constraint false_c(0*z == 1);
```

How to inspect a constraint

Several methods are provided to examine a constraint and extract all the encoded information: its space-dimension, its type (equality, non-strict inequality, strict inequality) and the value of its integer coefficients.

Example 2

The following code shows how it is possible to access each single coefficient of a constraint. Given an inequality constraint (in this case $x - 5y + 3z <= 4$), we construct a new constraint corresponding to its complement (thus, in this case we want to obtain the strict inequality constraint $x - 5y + 3z > 4$).

```
Constraint c1(x - 5*y + 3*z <= 4);
cout << "Constraint c1: " << c1 << endl;
if (c1.is_equality())
  cout << "Constraint c1 is not an inequality." << endl;
else {
  LinExpression e;
  for (int i = c1.space_dimension() - 1; i >= 0; i--)
    e += c1.coefficient(Variable(i)) * Variable(i);
  e += c1.inhomogeneous_term();
  Constraint c2 = c1.is_strict_inequality() ? (e <= 0) : (e < 0);
  cout << "Complement c2: " << c2 << endl;
}
```

The actual output is the following:

```
Constraint c1: -A + 5*B - 3*C >= -4
Complement c2: A - 5*B + 3*C > 4
```

Note that, in general, the particular output obtained can be syntactically different from the (semantically equivalent) constraint considered.

8.2.2 Member Enumeration Documentation

8.2.2.1 enum Parma_Polyhedra_Library::Constraint::Type

The constraint type.

Enumeration values:

EQUALITY The constraint is an equality.

NONSTRICT_INEQUALITY The constraint is a non-strict inequality.

STRICT_INEQUALITY The constraint is a strict inequality.

8.2.3 Member Function Documentation

8.2.3.1 const Integer& Constraint::coefficient (Variable *v*) const

Returns the coefficient of `v` in `*this`.

Exceptions:

std::invalid_argument thrown if the index of `v` is greater than or equal to the space-dimension of `*this`.

8.3 Determinate< PH > Class Template Reference

Wrap a polyhedron class into a determinate constraint system interface.

Public Member Functions

- dimension_type **space_dimension** () const
  Returns the dimension of the vector space enclosing `*this`.
- const ConSys & **constraints** () const
  Returns the system of constraints.
- const ConSys & **minimized_constraints** () const
  Returns the system of constraints, with no redundant constraint.
- const GenSys & **generators** () const
  Returns the system of generators.
- const GenSys & **minimized_generators** () const
  Returns the system of generators, with no redundant generator.
- void add_constraint (const Constraint &c)
  Intersects `*this` *with (a copy of) constraint* `c`.
- void add_constraints (ConSys &cs)
  Intersects `*this` *with the constraints in* `cs`.
- void **add_dimensions_and_embed** (dimension_type m)
  Adds `m` *new dimensions and embeds the old polyhedron into the new space.*
- void **add_dimensions_and_project** (dimension_type m)
  Adds `m` *new dimensions to the polyhedron and does not embed it in the new space.*
- void remove_dimensions (const Variables_Set &to_be_removed)
  Removes all the specified dimensions.
- void remove_higher_dimensions (dimension_type new_dimension)
  Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.
- void H79_widening_assign (const Determinate &y)

Assigns to $*$`this` *the result of computing the* *H79-widening* *between* $*$`this` *and* `y`.

- void limited_H79_extrapolation_assign (const Determinate &y, ConSys &cs)

  Limits the *H79-widening* *computation between* $*$`this` *and* `y` *by enforcing constraints* `cs` *and assigns the result to* $*$`this`.

- bool **OK** () const

  Checks if all the invariants are satisfied.

Friends

- bool operator== (const Determinate< PH > &x, const Determinate< PH > &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same polyhedron.*

- bool operator!= (const Determinate< PH > &x, const Determinate< PH > &y)

  Returns `true` *if and only if* `x` *and* `y` *are different polyhedra.*

- bool lcompare (const Determinate &x, const Determinate &y)

Related Functions

(Note that these are not member functions.)

- Determinate< PH > operator+ (const Determinate< PH > &x, const Determinate< PH > &y)
- Determinate< PH > operator $*$ (const Determinate< PH > &x, const Determinate< PH > &y)
- std::ostream & operator<< (std::ostream &, const Determinate< PH > &)

8.3.1 Detailed Description

template<typename PH> class Determinate< PH >

Wrap a polyhedron class into a determinate constraint system interface.

8.3.2 Member Function Documentation

8.3.2.1 template<typename PH> void Determinate< PH >::add_constraint (const Constraint & *c*)

Intersects $*$`this` with (a copy of) constraint `c`.

Exceptions:

std::invalid_argument thrown if $*$`this` and constraint `c` are topology-incompatible or dimension-incompatible.

8.3.2.2 template<typename PH> void Determinate< PH >::add_constraints (ConSys & *cs*)

Intersects ∗this with the constraints in cs.

Parameters:

cs The constraints to intersect with. This parameter is not declared const because it can be modified.

Exceptions:

std::invalid_argument thrown if ∗this and cs are topology-incompatible or dimension-incompatible.

8.3.2.3 template<typename PH> void Determinate< PH >::remove_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions.

Parameters:

to_be_removed The set of Variable objects corresponding to the dimensions to be removed.

Exceptions:

std::invalid_argument thrown if ∗this is dimension-incompatible with one of the Variable objects contained in to_be_removed.

8.3.2.4 template<typename PH> void Determinate< PH >::remove_higher_dimensions (dimension_type *new_dimension*)

Removes the higher dimensions so that the resulting space will have dimension new_dimension.

Exceptions:

std::invalid_argument thrown if new_dimensions is greater than the space dimension of ∗this.

8.3.2.5 template<typename PH> void Determinate< PH >::H79_widening_assign (const Determinate< PH > & *y*)

Assigns to ∗this the result of computing the H79-widening between ∗this and y.

Parameters:

y A polyhedron that *must* be contained in ∗this.

Exceptions:

std::invalid_argument thrown if ∗this and y are topology-incompatible or dimension-incompatible.

8.3.2.6 template<typename PH> void Determinate< PH >::limited_H79_extrapolation_assign (const Determinate< PH > & *y*, ConSys & *cs*)

Limits the H79-widening computation between ∗this and y by enforcing constraints cs and assigns the result to ∗this.

Parameters:

y A polyhedron that *must* be contained in ∗this.

cs The system of constraints that limits the widened polyhedron. It is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if `*this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.3.3 Friends And Related Function Documentation

8.3.3.1 template<typename PH> bool operator== (const Determinate< PH > & *x*, const Determinate< PH > & *y*) `[friend]`

Returns `true` if and only if `x` and `y` are the same polyhedron.

<PH>

Exceptions:

std::invalid_argument thrown if `x` and `y` are topology-incompatible or dimension-incompatible.

8.3.3.2 template<typename PH> bool operator!= (const Determinate< PH > & *x*, const Determinate< PH > & *y*) `[friend]`

Returns `true` if and only if `x` and `y` are different polyhedra.

<PH>

Exceptions:

std::invalid_argument thrown if `x` and `y` are topology-incompatible or dimension-incompatible.

8.3.3.3 template<typename PH> bool lcompare (const Determinate< PH > & *x*, const Determinate< PH > & *y*) `[friend]`

<PH>

8.3.3.4 template<typename PH> Determinate< PH > operator+ (const Determinate< PH > & *x*, const Determinate< PH > & *y*) `[related]`

<PH>

8.3.3.5 template<typename PH> Determinate< PH > operator * (const Determinate< PH > & *x*, const Determinate< PH > & *y*) `[related]`

<PH>

8.3.3.6 template<typename PH> std::ostream & operator<< (std::ostream &, const Determinate< PH > &) `[related]`

<PH>

8.4 Generator Class Reference

A line, ray, point or closure point.

Public Types

- enum Type { LINE, RAY, POINT, CLOSURE_POINT }

  The generator type.

Public Member Functions

- **Generator** (const Generator &g)

  Ordinary copy-constructor.

- ∼**Generator** ()

  Destructor.

- Generator & **operator=** (const Generator &g)

  Assignment operator.

- dimension_type **space_dimension** () const

  Returns the dimension of the vector space enclosing `*this`.

- Type **type** () const

  Returns the generator type of `*this`.

- bool **is_line** () const

  Returns `true` *if and only if* `*this` *is a line.*

- bool **is_ray** () const

  Returns `true` *if and only if* `*this` *is a ray.*

- bool **is_point** () const

  Returns `true` *if and only if* `*this` *is a point.*

- bool **is_closure_point** () const

  Returns `true` *if and only if* `*this` *is a closure point.*

- const Integer & coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- const Integer & divisor () const

  If `*this` *is either a point or a closure point, returns its divisor.*

- bool **OK** () const

  Checks if all the invariants are satisfied.

Static Public Member Functions

- Generator line (const LinExpression &e)

  Shorthand for Generator Generator::line(const LinExpression& e).

- Generator ray (const LinExpression &e)

  Shorthand for Generator Generator::ray(const LinExpression& e).

- Generator point (const LinExpression &e=LinExpression::zero(), const Integer &d=Integer_one())

  Shorthand for Generator Generator::point(const LinExpression& e, const Integer& d).

- Generator closure_point (const LinExpression &e=LinExpression::zero(), const Integer &d=Integer_one())

  Shorthand for Generator Generator::closure_point(const LinExpression& e, const Integer& d).

- const Generator & **zero_dim_point** ()

  Returns the origin of the zero-dimensional space $\mathbb{R}^0$.

- const Generator & **zero_dim_closure_point** ()

  Returns, as a closure point, the origin of the zero-dimensional space $\mathbb{R}^0$.

Related Functions

(Note that these are not member functions.)

- std::ostream & **operator**<< (std::ostream &s, const Generator &g)

  Output operator.

- void **swap** (Parma_Polyhedra_Library::Generator &x, Parma_Polyhedra_Library::Generator &y)

  Specializes `std::swap`.

8.4.1 Detailed Description

A line, ray, point or closure point.

An object of the class Generator is one of the following:

- a line $\boldsymbol{l} = (a_0, \ldots, a_{n-1})^{\mathrm{T}}$;
- a ray $\boldsymbol{r} = (a_0, \ldots, a_{n-1})^{\mathrm{T}}$;
- a point $\boldsymbol{p} = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$;
- a closure point $\boldsymbol{c} = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$;

where n is the dimension of the space and, for points and closure points, $d > 0$ is the divisor.

A note on terminology.

As observed in Section Representations of Convex Polyhedra, there are cases when, in order to represent a polyhedron $\mathcal{P}$ using the generator system $\mathcal{G} = (L, R, P, C)$, we need to include in the finite set P even points of $\mathcal{P}$ that are *not* vertices of $\mathcal{P}$. This situation is even more frequent when working with NNC polyhedra and it is the reason why we prefer to use the word 'point' where other libraries use the word 'vertex'.

How to build a generator.

Each type of generator is built by applying the corresponding function (`line`, `ray`, `point` or `closure_point`) to a linear expression, representing a direction in the space; the space-dimension of the generator is defined as the space-dimension of the corresponding linear expression. Linear expressions used to define a generator should be homogeneous (any constant term will be simply ignored). When defining points and closure points, an optional Integer argument can be used as a common *divisor* for all the coefficients occurring in the provided linear expression; the default value for this argument is 1.

In all the following examples it is assumed that variables `x`, `y` and `z` are defined as follows:

```
Variable x(0);
Variable y(1);
Variable z(2);
```

Example 1

The following code builds a line with direction $x - y - z$ and having space-dimension 3:

```
Generator l = line(x - y - z);
```

As mentioned above, the constant term of the linear expression is not relevant. Thus, the following code has the same effect:

```
Generator l = line(x - y - z + 15);
```

By definition, the origin of the space is not a line, so that the following code throws an exception:

```
Generator l = line(0*x);
```

Example 2

The following code builds a ray with the same direction as the line in Example 1:

```
Generator r = ray(x - y - z);
```

As is the case for lines, when specifying a ray the constant term of the linear expression is not relevant; also, an exception is thrown when trying to build a ray from the origin of the space.

Example 3

The following code builds the point $\boldsymbol{p} = (1, 0, 2)^{\mathrm{T}} \in \mathbb{R}^3$:

```
Generator p = point(1*x + 0*y + 2*z);
```

The same effect can be obtained by using the following code:

```
Generator p = point(x + 2*z);
```

Similarly, the origin $\mathbf{0} \in \mathbb{R}^3$ can be defined using either one of the following lines of code:

```
Generator origin3 = point(0*x + 0*y + 0*z);
Generator origin3_alt = point(0*z);
```

Note however that the following code would have defined a different point, namely $\mathbf{0} \in \mathbb{R}^2$:

```
Generator origin2 = point(0*y);
```

The following two lines of code both define the only point having space-dimension zero, namely $\mathbf{0} \in \mathbb{R}^0$. In the second case we exploit the fact that the first argument of the function `point` is optional.

```
Generator origin0 = Generator::zero_dim_point();
Generator origin0_alt = point();
```

Example 4

The point $\boldsymbol{p}$ specified in Example 3 above can also be obtained with the following code, where we provide a non-default value for the second argument of the function `point` (the divisor):

```
Generator p = point(2*x + 0*y + 4*z, 2);
```

Obviously, the divisor can be usefully exploited to specify points having some non-integer (but rational) coordinates. For instance, the point $\boldsymbol{q} = (-1.5, 3.2, 2.1)^{\mathrm{T}} \in \mathbb{R}^3$ can be specified by the following code:

```
Generator q = point(-15*x + 32*y + 21*z, 10);
```

If a zero divisor is provided, an exception is thrown.

Example 5

Closures points are specified in the same way we defined points, but invoking their specific constructor function. For instance, the closure point $\boldsymbol{c} = (1, 0, 2)^{\mathrm{T}} \in \mathbb{R}^3$ is defined by

```
Generator c = closure_point(1*x + 0*y + 2*z);
```

For the particular case of the (only) closure point having space-dimension zero, we can use any of the following:

```
Generator closure_origin0 = Generator::zero_dim_closure_point();
Generator closure_origin0_alt = closure_point();
```

How to inspect a generator

Several methods are provided to examine a generator and extract all the encoded information: its space-dimension, its type and the value of its integer coefficients.

Example 6

The following code shows how it is possible to access each single coefficient of a generator. If `g1` is a point having coordinates $(a_0, \ldots, a_{n-1})^{\mathrm{T}}$, we construct the closure point `g2` having coordinates $(a_0, 2a_1, \ldots, (i+1)a_i, \ldots, na_{n-1})^{\mathrm{T}}$.

```
if (g1.is_point()) {
  cout << "Point g1: " << g1 << endl;
  LinExpression e;
  for (int i = g1.space_dimension() - 1; i >= 0; i--)
    e += (i + 1) * g1.coefficient(Variable(i)) * Variable(i);
  Generator g2 = closure_point(e, g1.divisor());
  cout << "Closure point g2: " << g2 << endl;
}
else
  cout << "Generator g1 is not a point." << endl;
```

Therefore, for the point

```
Generator g1 = point(2*x - y + 3*z, 2);
```

we would obtain the following output:

```
Point g1: p((2*A - B + 3*C)/2)
Closure point g2: cp((2*A - 2*B + 9*C)/2)
```

When working with (closure) points, be careful not to confuse the notion of *coefficient* with the notion of *coordinate*: these are equivalent only when the divisor of the (closure) point is 1.

8.4.2 Member Enumeration Documentation

8.4.2.1 enum Parma_Polyhedra_Library::Generator::Type

The generator type.

Enumeration values:

LINE The generator is a line.

RAY The generator is a ray.

POINT The generator is a point.

CLOSURE_POINT The generator is a closure point.

8.4.3 Member Function Documentation

8.4.3.1 Generator line (const LinExpression & *e*) `[static]`

Shorthand for Generator Generator::line(const LinExpression& e).

Exceptions:

std::invalid_argument thrown if the homogeneous part of `e` represents the origin of the vector space.

8.4.3.2 Generator ray (const LinExpression & *e*) `[static]`

Shorthand for Generator Generator::ray(const LinExpression& e).

Exceptions:

std::invalid_argument thrown if the homogeneous part of `e` represents the origin of the vector space.

8.4.3.3 Generator point (const LinExpression & *e* = LinExpression::zero(), const Integer & *d* = Integer_one()) `[static]`

Shorthand for Generator Generator::point(const LinExpression& e, const Integer& d).

Both `e` and `d` are optional arguments, with default values LinExpression::zero() and Integer_one(), respectively.

Exceptions:

std::invalid_argument thrown if `d` is zero.

8.4.3.4 Generator closure_point (const LinExpression & *e* = LinExpression::zero(), const Integer & *d* = Integer_one()) `[static]`

Shorthand for Generator Generator::closure_point(const LinExpression& e, const Integer& d).

Both `e` and `d` are optional arguments, with default values LinExpression::zero() and Integer_one(), respectively.

Exceptions:

std::invalid_argument thrown if `d` is zero.

8.4.3.5 const Integer& Generator::coefficient (Variable *v*) const

Returns the coefficient of `v` in ∗`this`.

Exceptions:

std::invalid_argument thrown if the index of `v` is greater than or equal to the space-dimension of ∗`this`.

8.4.3.6 const Integer& Generator::divisor () const

If ∗`this` is either a point or a closure point, returns its divisor.

Exceptions:

std::invalid_argument thrown if ∗`this` is neither a point nor a closure point.

8.5 LinExpression Class Reference

A linear expression.

Public Member Functions

- **LinExpression** ()

  Default constructor: returns a copy of LinExpression::zero().

- **LinExpression** (const LinExpression &e)

  Ordinary copy-constructor.

- virtual ∼**LinExpression** ()

  Destructor.

- **LinExpression** (const Integer &n)

  Builds the linear expression corresponding to the inhomogeneous term `n`.

- **LinExpression** (const Variable v)

  Builds the linear expression corresponding to the variable `v`.

- LinExpression (const Constraint &c)

  Builds the linear expression corresponding to constraint `c`.

- LinExpression (const Generator &g)

  Builds the linear expression corresponding to generator `g` *(for points and closure points, the divisor is not copied).*

- dimension_type **space_dimension** () const

  Returns the dimension of the vector space enclosing ∗`this`.

- const Integer & **coefficient** (Variable v) const

  Returns the coefficient of `v` *in* ∗`this`.

- const Integer & **inhomogeneous_term** () const

  Returns the inhomogeneous term of ∗`this`.

Static Public Member Functions

- const LinExpression & **zero** ()
  Returns the (zero-dimension space) constant 0.

Related Functions

(Note that these are not member functions.)

- LinExpression **operator+** (const LinExpression &e1, const LinExpression &e2)
  Returns the linear expression `e1 + e2`.
- LinExpression **operator+** (const Integer &n, const LinExpression &e)
  Returns the linear expression `n + e`.
- LinExpression **operator+** (const LinExpression &e, const Integer &n)
  Returns the linear expression `e + n`.
- LinExpression **operator+** (const LinExpression &e)
  Returns the linear expression `e`.
- LinExpression **operator-** (const LinExpression &e)
  Returns the linear expression `- e`.
- LinExpression **operator-** (const LinExpression &e1, const LinExpression &e2)
  Returns the linear expression `e1 - e2`.
- LinExpression **operator-** (const Integer &n, const LinExpression &e)
  Returns the linear expression `n - e`.
- LinExpression **operator-** (const LinExpression &e, const Integer &n)
  Returns the linear expression `e - n`.
- LinExpression **operator** ∗ (const Integer &n, const LinExpression &e)
  Returns the linear expression `n ∗ e`.
- LinExpression **operator** ∗ (const LinExpression &e, const Integer &n)
  Returns the linear expression `e ∗ n`.
- LinExpression & **operator+=** (LinExpression &e1, const LinExpression &e2)
  Returns the linear expression `e1 + e2` *and assigns it to* `e1`.
- LinExpression & **operator+=** (LinExpression &e, const Variable v)
  Returns the linear expression `e + v` *and assigns it to* `e`.
- LinExpression & **operator+=** (LinExpression &e, const Integer &n)
  Returns the linear expression `e + n` *and assigns it to* `e`.
- LinExpression & **operator-=** (LinExpression &e1, const LinExpression &e2)

Returns the linear expression `e1 - e2` *and assigns it to* `e1`.

- LinExpression & **operator-=** (LinExpression &e, const Variable v)

  Returns the linear expression `e - v` *and assigns it to* `e`.

- LinExpression & **operator-=** (LinExpression &e, const Integer &n)

  Returns the linear expression `e - n` *and assigns it to* `e`.

- LinExpression & **operator** $*$= (LinExpression &e, const Integer &n)

  Returns the linear expression `n` $*$ `e` *and assigns it to* `e`.

- void **swap** (Parma_Polyhedra_Library::LinExpression &x, Parma_Polyhedra_Library::LinExpression &y)

  Specializes `std::swap`.

8.5.1 Detailed Description

A linear expression.

An object of the class LinExpression represents the linear expression

$$\sum_{i=0}^{n-1} a_i x_i + b$$

where n is the dimension of the space, each a_i is the integer coefficient of the `i` -th variable x_i and b is the integer for the inhomogeneous term.

How to build a linear expression.

Linear expressions are the basic blocks for defining both constraints (i.e., linear equalities or inequalities) and generators (i.e., lines, rays, points and closure points). A full set of functions is defined to provide a convenient interface for building complex linear expressions starting from simpler ones and from objects of the classes Variable and Integer: available operators include unary negation, binary addition and subtraction, as well as multiplication by an Integer. The space-dimension of a linear expression is defined as the maximum space-dimension of the arguments used to build it: in particular, the space-dimension of a Variable `x` is defined as `x.id()+1`, whereas all the objects of the class Integer have space-dimension zero.

Example

The following code builds the linear expression $4x - 2y - z + 14$, having space-dimension 3:

```
LinExpression e = 4*x - 2*y - z + 14;
```

Another way to build the same linear expression is:

```
LinExpression e1 = 4*x;
LinExpression e2 = 2*y;
LinExpression e3 = z;
LinExpression e = LinExpression(14);
e += e1 - e2 - e3;
```

Note that `e1`, `e2` and `e3` have space-dimension 1, 2 and 3, respectively; also, in the fourth line of code, `e` is created with space-dimension zero and then extended to space-dimension 3.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 LinExpression::LinExpression (const Constraint & *c*) `[explicit]`

Builds the linear expression corresponding to constraint `c`.

Given the constraint $c = \bigl(\sum_{i=0}^{n-1} a_i x_i + b \bowtie 0\bigr)$, where $\bowtie \in \{=, \geq, >\}$, builds the linear expression $\sum_{i=0}^{n-1} a_i x_i + b$. If `c` is an inequality (resp., equality) constraint, then the built linear expression is unique up to a positive (resp., non-zero) factor.

8.5.2.2 LinExpression::LinExpression (const Generator & *g*) `[explicit]`

Builds the linear expression corresponding to generator `g` (for points and closure points, the divisor is not copied).

Given the generator $g = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$ (where, for lines and rays, we have $d = 1$), builds the linear expression $\sum_{i=0}^{n-1} a_i x_i$. The inhomogeneous term of the linear expression will always be 0. If `g` is a ray, point or closure point (resp., a line), then the linear expression is unique up to a positive (resp., non-zero) factor.

8.6 NNC_Polyhedron Class Reference

A not necessarily closed convex polyhedron.

Inherits Polyhedron.

Public Member Functions

- NNC_Polyhedron (dimension_type num_dimensions=0, Degenerate_Kind kind=UNIVERSE)

  Builds either the universe or the empty NNC polyhedron.

- NNC_Polyhedron (const ConSys &cs)

  Builds an NNC polyhedron from a system of constraints.

- NNC_Polyhedron (ConSys &cs)

  Builds an NNC polyhedron recycling a system of constraints.

- NNC_Polyhedron (const GenSys &gs)

  Builds an NNC polyhedron from a system of generators.

- NNC_Polyhedron (GenSys &gs)

  Builds an NNC polyhedron recycling a system of generators.

- **NNC_Polyhedron** (const C_Polyhedron &y)

  Builds an NNC polyhedron from the C polyhedron `y`.

- template<typename Box> NNC_Polyhedron (const Box &box, From_Bounding_Box dummy)

  Builds an NNC polyhedron out of a generic, interval-based bounding box.

- **NNC_Polyhedron** (const NNC_Polyhedron &y)

  Ordinary copy-constructor.

- NNC_Polyhedron & **operator=** (const NNC_Polyhedron &y)

  The assignment operator. (`*this` *and* `y` *can be dimension-incompatible.).*

- ~**NNC_Polyhedron** ()

  Destructor.

8.6.1 Detailed Description

A not necessarily closed convex polyhedron.

An object of the class NNC_Polyhedron represents a *not necessarily closed* (NNC) convex polyhedron in the vector space $\mathbb{R}^n$.

Note:

Since NNC polyhedra are a generalization of closed polyhedra, any object of the class C_Polyhedron can be (explicitly) converted into an object of the class NNC_Polyhedron. The reason for defining two different classes is that objects of the class C_Polyhedron are characterized by a more efficient implementation, requiring less time and memory resources.

8.6.2 Constructor & Destructor Documentation

8.6.2.1 NNC_Polyhedron::NNC_Polyhedron (dimension_type *num_dimensions* = 0, Degenerate_Kind *kind* = UNIVERSE) `[explicit]`

Builds either the universe or the empty NNC polyhedron.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the NNC polyhedron.

kind Specifies whether a universe or an empty NNC polyhedron should be built.

Both parameters are optional: by default, a 0-dimension space universe NNC polyhedron is built.

8.6.2.2 NNC_Polyhedron::NNC_Polyhedron (const ConSys & *cs*)

Builds an NNC polyhedron from a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

8.6.2.3 NNC_Polyhedron::NNC_Polyhedron (ConSys & *cs*)

Builds an NNC polyhedron recycling a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

8.6.2.4 NNC_Polyhedron::NNC_Polyhedron (const GenSys & *gs*)

Builds an NNC polyhedron from a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

gs The system of generators defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of generators is not empty but has no points.

8.6.2.5 NNC_Polyhedron::NNC_Polyhedron (GenSys & *gs*)

Builds an NNC polyhedron recycling a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

gs The system of generators defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if the system of generators is not empty but has no points.

8.6.2.6 template<typename Box> NNC_Polyhedron::NNC_Polyhedron (const Box & *box*, From_-Bounding_Box *dummy*)

Builds an NNC polyhedron out of a generic, interval-based bounding box.

For a description of the methods that should be provided by the template class Box, see the documentation of the protected method: template <typename Box> Polyhedron::Polyhedron(Topology topol, const Box& box);

Parameters:

box The bounding box representing the polyhedron to be built.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

8.7 Poly_Con_Relation Class Reference

The relation between a polyhedron and a constraint.

Public Member Functions

- bool **implies** (const Poly_Con_Relation &y) const

  True if and only if `*this` *implies* `y`.

- bool **OK** () const

  Checks if all the invariants are satisfied.

Static Public Member Functions

- Poly_Con_Relation **nothing** ()
  The assertion that says nothing.
- Poly_Con_Relation **is_disjoint** ()
  The polyhedron and the set of points satisfying the constraint are disjoint.
- Poly_Con_Relation **strictly_intersects** ()
  The polyhedron intersects the set of points satisfying the constraint, but it is not included in it.
- Poly_Con_Relation **is_included** ()
  The polyhedron is included in the set of points satisfying the constraint.
- Poly_Con_Relation **saturates** ()
  The polyhedron is included in the set of points saturating the constraint.

Related Functions

(Note that these are not member functions.)

- bool **operator==** (const Poly_Con_Relation &x, const Poly_Con_Relation &y)
  True if and only if `x` *and* `y` *are logically equivalent.*
- bool **operator!=** (const Poly_Con_Relation &x, const Poly_Con_Relation &y)
  True if and only if `x` *and* `y` *are not logically equivalent.*
- Poly_Con_Relation **operator &&** (const Poly_Con_Relation &x, const Poly_Con_Relation &y)
  Yields the logical conjunction of `x` *and* `y`.
- Poly_Con_Relation **operator-** (const Poly_Con_Relation &x, const Poly_Con_Relation &y)
  Yields the assertion with all the conjuncts of `x` *that are not in* `y`.
- std::ostream & **operator**<< (std::ostream &s, const Poly_Con_Relation &r)
  Output operator.

8.7.1 Detailed Description

The relation between a polyhedron and a constraint.

This class implements conjunctions of assertions on the relation between a polyhedron and a constraint.

8.8 Poly_Gen_Relation Class Reference

The relation between a polyhedron and a generator.

Public Member Functions

- bool **implies** (const Poly_Gen_Relation &y) const
  True if and only if `*this` *implies* `y`.
- bool **OK** () const
  Checks if all the invariants are satisfied.

Static Public Member Functions

- Poly_Gen_Relation **nothing** ()
  The assertion that says nothing.
- Poly_Gen_Relation **subsumes** ()
  Adding the generator would not change the polyhedron.

Related Functions

(Note that these are not member functions.)

- bool **operator==** (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)
  True if and only if `x` *and* `y` *are logically equivalent.*
- bool **operator!=** (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)
  True if and only if `x` *and* `y` *are not logically equivalent.*
- Poly_Gen_Relation **operator &&** (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)
  Yields the logical conjunction of `x` *and* `y`.
- Poly_Gen_Relation **operator-** (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)
  Yields the assertion with all the conjuncts of `x` *that are not in* `y`.
- std::ostream & **operator**<< (std::ostream &s, const Poly_Gen_Relation &r)
  Output operator.

8.8.1 Detailed Description

The relation between a polyhedron and a generator.

This class implements conjunctions of assertions on the relation between a polyhedron and a generator.

8.9 Polyhedron Class Reference

The base class for convex polyhedra.

Inherited by C_Polyhedron, and NNC_Polyhedron.

Public Types

- enum Degenerate_Kind { UNIVERSE, EMPTY }

  Kinds of degenerate polyhedra.

Public Member Functions

Member Functions that Do Not Modify the Polyhedron

- dimension_type **space_dimension** () const

  Returns the dimension of the vector space enclosing `*this`.

- const ConSys & **constraints** () const

  Returns the system of constraints.

- const ConSys & **minimized_constraints** () const

  Returns the system of constraints, with no redundant constraint.

- const GenSys & **generators** () const

  Returns the system of generators.

- const GenSys & **minimized_generators** () const

  Returns the system of generators, with no redundant generator.

- Poly_Con_Relation relation_with (const Constraint &c) const

  Returns the relations holding between the polyhedron `*this` *and the constraint* `c`.

- Poly_Gen_Relation relation_with (const Generator &g) const

  Returns the relations holding between the polyhedron `*this` *and the generator* `g`.

- bool **is_empty** () const

  Returns `true` *if and only if* `*this` *is an empty polyhedron.*

- bool **is_universe** () const

  Returns `true` *if and only if* `*this` *is a universe polyhedron.*

- bool **is_topologically_closed** () const

  Returns `true` *if and only if* `*this` *is a topologically closed subset of the vector space.*

- bool is_disjoint_from (const Polyhedron &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are disjoint.*

- bool **is_bounded** () const

  Returns `true` *if and only if* `*this` *is a bounded polyhedron.*

- bool bounds_from_above (const LinExpression &expr) const

  Returns `true` *if and only if* `expr` *is bounded from above in* `*this`.

- bool bounds_from_below (const LinExpression &expr) const

  Returns `true` *if and only if* `expr` *is bounded from below in* `*this`.

- bool contains (const Polyhedron &y) const

Returns `true` *if and only if* `*this` *contains* `y`.

- bool strictly_contains (const Polyhedron &y) const
  Returns `true` *if and only if* `*this` *strictly contains* `y`.

- template<typename Box> void shrink_bounding_box (Box &box, Complexity_Class complexity=ANY) const
  Uses `*this` *to shrink a generic, interval-based bounding box.*

- bool OK (bool check_not_empty=false) const
  Checks if all the invariants are satisfied.

Space-Dimension Preserving Member Functions that May Modify the Polyhedron

- void add_constraint (const Constraint &c)
  Adds a copy of constraint `c` *to the system of constraints of* `*this` *(without minimizing the result).*

- bool add_constraint_and_minimize (const Constraint &c)
  Adds a copy of constraint `c` *to the system of constraints of* `*this`*, minimizing the result.*

- void add_generator (const Generator &g)
  Adds a copy of generator `g` *to the system of generators of* `*this` *(without minimizing the result).*

- bool add_generator_and_minimize (const Generator &g)
  Adds a copy of generator `g` *to the system of generators of* `*this`*, minimizing the result.*

- void add_constraints (ConSys &cs)
  Adds the constraints in `cs` *to the system of constraints of* `*this`*, minimizing the result.*

- bool add_constraints_and_minimize (ConSys &cs)
  Adds the constraints in `cs` *to the system of constraints of* `*this` *(without minimizing the result).*

- void add_generators (GenSys &gs)
  Adds the generators in `gs` *to the system of generators of* `*this` *(without minimizing the result).*

- bool add_generators_and_minimize (GenSys &gs)
  Adds the generators in `gs` *to the system of generators of* `*this`*, minimizing the result.*

- void intersection_assign (const Polyhedron &y)
  Assigns to `*this` *the intersection of* `*this` *and* `y`*. The result is not guaranteed to be minimized.*

- bool intersection_assign_and_minimize (const Polyhedron &y)
  Assigns to `*this` *the intersection of* `*this` *and* `y`*, minimizing the result.*

- void poly_hull_assign (const Polyhedron &y)
  Assigns to `*this` *the poly-hull* `*this` *and* `y`*. The result is not guaranteed to be minimized.*

- bool poly_hull_assign_and_minimize (const Polyhedron &y)
  Assigns to `*this` *the poly-hull of* `*this` *and* `y`*, minimizing the result.*

- void poly_difference_assign (const Polyhedron &y)
  Assigns to `*this` *the poly-difference of* `*this` *and* `y`*. The result is not guaranteed to be minimized.*

- void affine_image (Variable var, const LinExpression &expr, const Integer &denominator=Integer_one())

Assigns to `*this` *the affine image of* `*this` *under the function mapping variable* `var` *into the affine expression specified by* `expr` *and* `denominator`.

- void affine_preimage (Variable var, const LinExpression &expr, const Integer &denominator=Integer_one())

  Assigns to `*this` *the affine preimage of* `*this` *under the function mapping variable* `var` *into the affine expression specified by* `expr` *and* `denominator`.

- void generalized_affine_image (Variable var, const Relation_Symbol relsym, const LinExpression &expr, const Integer &denominator=Integer_one())

  Assigns to `*this` *the image of* `*this` *with respect to the generalized affine transfer function* $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$*, where* $\bowtie$ *is the relation symbol encoded by* `relsym`.

- void generalized_affine_image (const LinExpression &lhs, const Relation_Symbol relsym, const LinExpression &rhs)

  Assigns to `*this` *the image of* `*this` *with respect to the generalized affine transfer function* $\mathrm{lhs}' \bowtie \mathrm{rhs}$*, where* $\bowtie$ *is the relation symbol encoded by* `relsym`.

- void time_elapse_assign (const Polyhedron &y)

  Assigns to `*this` *the result of computing the time-elapse between* `*this` *and* `y`.

- void **topological_closure_assign** ()

  Assigns to `*this` *its topological closure.*

- void BHRZ03_widening_assign (const Polyhedron &y, unsigned *tp=0)

  Assigns to `*this` *the result of computing the BHRZ03-widening between* `*this` *and* `y`.

- void limited_BHRZ03_extrapolation_assign (const Polyhedron &y, const ConSys &cs, unsigned *tp=0)

  Improves the result of the BHRZ03-widening computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`.

- void bounded_BHRZ03_extrapolation_assign (const Polyhedron &y, const ConSys &cs, unsigned *tp=0)

  Improves the result of the BHRZ03-widening computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`*, plus all the constraints of the form* $\pm x \leq r$ *and* $\pm x < r$*, with* $r \in \mathbb{Q}$*, that are satisfied by all the points of* `*this`.

- void H79_widening_assign (const Polyhedron &y, unsigned *tp=0)

  Assigns to `*this` *the result of computing the H79-widening between* `*this` *and* `y`.

- void limited_H79_extrapolation_assign (const Polyhedron &y, const ConSys &cs, unsigned *tp=0)

  Improves the result of the H79-widening computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`.

- void bounded_H79_extrapolation_assign (const Polyhedron &y, const ConSys &cs, unsigned *tp=0)

  Improves the result of the H79-widening computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`*, plus all the constraints of the form* $\pm x \leq r$ *and* $\pm x < r$*, with* $r \in \mathbb{Q}$*, that are satisfied by all the points of* `*this`.

Member Functions that May Modify the Dimension of the Vector Space

- void add_dimensions_and_embed (dimension_type m)

Adds `m` *new dimensions and embeds the old polyhedron into the new space.*

- void add_dimensions_and_project (dimension_type m)
  Adds `m` *new dimensions to the polyhedron and does not embed it in the new space.*
- void concatenate_assign (const Polyhedron &y)
  Seeing a polyhedron as a set of tuples (its points), assigns to `*this` *all the tuples that can be obtained by concatenating, in the order given, a tuple of* `*this` *with a tuple of* `y`.
- void remove_dimensions (const Variables_Set &to_be_removed)
  Removes all the specified dimensions.
- void remove_higher_dimensions (dimension_type new_dimension)
  Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.
- template<typename PartialFunction> void map_dimensions (const PartialFunction &pfunc)
  Remaps the dimensions of the vector space according to a partial function.

Miscellaneous Member Functions

- ~**Polyhedron** ()
  Destructor.
- void swap (Polyhedron &y)
  Swaps `*this` *with polyhedron* `y`. *(*`*this` *and* `y` *can be dimension-incompatible.).*

Protected Member Functions

- Polyhedron (Topology topol, dimension_type num_dimensions, Degenerate_Kind kind)
  Builds a polyhedron having the specified properties.
- **Polyhedron** (const Polyhedron &y)
  Ordinary copy-constructor.
- Polyhedron (Topology topol, const ConSys &cs)
  Builds a polyhedron from a system of constraints.
- Polyhedron (Topology topol, ConSys &cs)
  Builds a polyhedron recycling a system of constraints.
- Polyhedron (Topology topol, const GenSys &gs)
  Builds a polyhedron from a system of generators.
- Polyhedron (Topology topol, GenSys &gs)
  Builds a polyhedron recycling a system of generators.
- template<typename Box> Polyhedron (Topology topol, const Box &box)
  Builds a polyhedron out of a generic, interval-based bounding box.
- Polyhedron & **operator=** (const Polyhedron &y)
  The assignment operator. (`*this` *and* `y` *can be dimension-incompatible.).*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Polyhedron &ph)

  Output operator.

- bool operator== (const Polyhedron &x, const Polyhedron &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same polyhedron.*

- bool operator!= (const Polyhedron &x, const Polyhedron &y)

  Returns `true` *if and only if* `x` *and* `y` *are different polyhedra.*

- void **swap** (Parma_Polyhedra_Library::Polyhedron &x, Parma_Polyhedra_Library::Polyhedron &y)

  Specializes `std::swap`.

8.9.1 Detailed Description

The base class for convex polyhedra.

An object of the class Polyhedron represents a convex polyhedron in the vector space $\mathbb{R}^n$.

A polyhedron can be specified as either a finite system of constraints or a finite system of generators (see Section Representations of Convex Polyhedra) and it is always possible to obtain either representation. That is, if we know the system of constraints, we can obtain from this the system of generators that define the same polyhedron and vice versa. These systems can contain redundant members: in this case we say that they are not in the minimal form. Most operators on polyhedra are provided with two implementations: one of these, denoted <`operator-name`>`_and_minimize`, also enforces the minimization of the representations, and returns the Boolean value `false` whenever the resulting polyhedron turns out to be empty.

Two key attributes of any polyhedron are its topological kind (recording whether it is a C_Polyhedron or an NNC_Polyhedron object) and its space dimension (the dimension $n \in \mathbb{N}$ of the enclosing vector space):

- all polyhedra, the empty ones included, are endowed with a specific topology and space dimension;
- most operations working on a polyhedron and another object (i.e., another polyhedron, a constraint or generator, a set of variables, etc.) will throw an exception if the polyhedron and the object are not both topology-compatible and dimension-compatible (see Section Representations of Convex Polyhedra);
- there is no way to change the topology of a polyhedron; rather, there are constructors of the two derived classes that builds a new polyhedron having a topology when provided with the corresponding polyhedron of the other topology;
- the only ways to change the space dimension of a polyhedron are:
  - *explicit* calls to operators provided for that purpose;
  - standard copy, assignment and swap operators.

Note that four different polyhedra can be defined on the zero-dimension space: the empty polyhedron, either closed or NNC, and the universe polyhedron R^0, again either closed or NNC.

In all the examples it is assumed that variables `x` and `y` are defined (where they are used) as follows:

```
Variable x(0);
Variable y(1);
```

Example 1

The following code builds a polyhedron corresponding to a square in $\mathbb{R}^2$, given as a system of constraints:

```
ConSys cs;
cs.add_constraint(x >= 0);
cs.add_constraint(x <= 3);
cs.add_constraint(y >= 0);
cs.add_constraint(y <= 3);
C_Polyhedron ph(cs);
```

The following code builds the same polyhedron as above, but starting from a system of generators specifying the four vertices of the square:

```
GenSys gs;
gs.add_generator(point(0*x + 0*y));
gs.add_generator(point(0*x + 3*y));
gs.add_generator(point(3*x + 0*y));
gs.add_generator(point(3*x + 3*y));
C_Polyhedron ph(gs);
```

Example 2

The following code builds an unbounded polyhedron corresponding to a half-strip in $\mathbb{R}^2$, given as a system of constraints:

```
ConSys cs;
cs.add_constraint(x >= 0);
cs.add_constraint(x - y <= 0);
cs.add_constraint(x - y + 1 >= 0);
C_Polyhedron ph(cs);
```

The following code builds the same polyhedron as above, but starting from the system of generators specifying the two vertices of the polyhedron and one ray:

```
GenSys gs;
gs.add_generator(point(0*x + 0*y));
gs.add_generator(point(0*x + y));
gs.add_generator(ray(x - y));
C_Polyhedron ph(gs);
```

Example 3

The following code builds the polyhedron corresponding to an half-plane by adding a single constraint to the universe polyhedron in $\mathbb{R}^2$:

```
C_Polyhedron ph(2);
ph.add_constraint(y >= 0);
```

The following code builds the same polyhedron as above, but starting from the empty polyhedron in the space $\mathbb{R}^2$ and inserting the appropriate generators (a point, a ray and a line).

```
C_Polyhedron ph(2, Polyhedron::EMPTY);
ph.add_generator(point(0*x + 0*y));
ph.add_generator(ray(y));
ph.add_generator(line(x));
```

Note that, although the above polyhedron has no vertices, we must add one point, because otherwise the result of the Minkowsky's sum would be an empty polyhedron. To avoid subtle errors related to the minimization process, it is required that the first generator inserted in an empty polyhedron is a point (otherwise, an exception is thrown).

Example 4

The following code shows the use of the function `add_dimensions_and_embed`:

```
C_Polyhedron ph(1);
ph.add_constraint(x == 2);
ph.add_dimensions_and_embed(1);
```

We build the universe polyhedron in the 1-dimension space $\mathbb{R}$. Then we add a single equality constraint, thus obtaining the polyhedron corresponding to the singleton set $\{2\} \subseteq \mathbb{R}$. After the last line of code, the resulting polyhedron is

$$\left\{ (2, y)^{\mathrm{T}} \in \mathbb{R}^2 \;\middle|\; y \in \mathbb{R} \right\}.$$

Example 5

The following code shows the use of the function `add_dimensions_and_project`:

```
C_Polyhedron ph(1);
ph.add_constraint(x == 2);
ph.add_dimensions_and_project(1);
```

The first two lines of code are the same as in Example 4 for `add_dimensions_and_embed`. After the last line of code, the resulting polyhedron is the singleton set $\left\{(2, 0)^{\mathrm{T}}\right\} \subseteq \mathbb{R}^2$.

Example 6

The following code shows the use of the function `affine_image`:

```
C_Polyhedron ph(2, Polyhedron::EMPTY);
ph.add_generator(point(0*x + 0*y));
ph.add_generator(point(0*x + 3*y));
ph.add_generator(point(3*x + 0*y));
ph.add_generator(point(3*x + 3*y));
LinExpression coeff = x + 4;
ph.affine_image(x, coeff);
```

In this example the starting polyhedron is a square in $\mathbb{R}^2$, the considered variable is x and the affine expression is $x + 4$. The resulting polyhedron is the same square translated to the right. Moreover, if the affine transformation for the same variable `x` is $x + y$:

```
LinExpression coeff = x + y;
```

the resulting polyhedron is a parallelogram with the height equal to the side of the square and the oblique sides parallel to the line $x - y$. Instead, if we do not use an invertible transformation for the same variable; for example, the affine expression y:

```
LinExpression coeff = y;
```

the resulting polyhedron is a diagonal of the square.

Example 7

The following code shows the use of the function `affine_preimage`:

```
C_Polyhedron ph(2);
ph.add_constraint(x >= 0);
ph.add_constraint(x <= 3);
ph.add_constraint(y >= 0);
ph.add_constraint(y <= 3);
LinExpression coeff = x + 4;
ph.affine_preimage(x, coeff);
```

In this example the starting polyhedron, `var` and the affine expression and the denominator are the same as in Example 6, while the resulting polyhedron is again the same square, but translated to the left. Moreover, if the affine transformation for `x` is $x + y$

```
LinExpression coeff = x + y;
```

the resulting polyhedron is a parallelogram with the height equal to the side of the square and the oblique sides parallel to the line $x + y$. Instead, if we do not use an invertible transformation for the same variable `x`, for example, the affine expression y:

```
LinExpression coeff = y;
```

the resulting polyhedron is a line that corresponds to the y axis.

Example 8

For this example we use also the variables:

```
Variable z(2);
Variable w(3);
```

The following code shows the use of the function `remove_dimensions`:

```
GenSys gs;
gs.add_generator(point(3*x + y +0*z + 2*w));
C_Polyhedron ph(gs);
set<Variable> to_be_removed;
to_be_removed.insert(y);
to_be_removed.insert(z);
ph.remove_dimensions(to_be_removed);
```

The starting polyhedron is the singleton set $\{(3, 1, 0, 2)^{\mathrm{T}}\} \subseteq \mathbb{R}^4$, while the resulting polyhedron is $\{(3, 2)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$. Be careful when removing dimensions *incrementally*: since dimensions are automatically renamed after each application of the `remove_dimensions` operator, unexpected results can be obtained. For instance, by using the following code we would obtain a different result:

```
set<Variable> to_be_removed1;
to_be_removed1.insert(y);
ph.remove_dimensions(to_be_removed1);
set<Variable> to_be_removed2;
to_be_removed2.insert(z);
ph.remove_dimensions(to_be_removed2);
```

In this case, the result is the polyhedron $\{(3, 0)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$: when removing the set of dimensions `to_be_removed2` we are actually removing variable w of the original polyhedron. For the same reason, the operator `remove_dimensions` is not idempotent: removing twice the same set of dimensions is never a no-op.

8.9.2 Member Enumeration Documentation

8.9.2.1 enum Parma_Polyhedra_Library::Polyhedron::Degenerate_Kind

Kinds of degenerate polyhedra.

Enumeration values:

UNIVERSE The universe polyhedron, i.e., the whole vector space.

EMPTY The empty polyhedron, i.e., the empty set.

8.9.3 Constructor & Destructor Documentation

8.9.3.1 Polyhedron::Polyhedron (Topology *topol*, dimension_type *num_dimensions*, Degenerate_Kind *kind*) `[protected]`

Builds a polyhedron having the specified properties.

Parameters:

topol The topology of the polyhedron;

num_dimensions The number of dimensions of the vector space enclosing the polyhedron;

kind Specifies whether the universe or the empty polyhedron has to be built.

8.9.3.2 Polyhedron::Polyhedron (Topology *topol*, const ConSys & *cs*) `[protected]`

Builds a polyhedron from a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

topol The topology of the polyhedron;

cs The system of constraints defining the polyhedron.

Exceptions:

std::invalid_argument thrown if the topology of `cs` is incompatible with `topology`.

8.9.3.3 Polyhedron::Polyhedron (Topology *topol*, ConSys & *cs*) `[protected]`

Builds a polyhedron recycling a system of constraints.

The polyhedron inherits the space dimension of the constraint system.

Parameters:

topol The topology of the polyhedron;

cs The system of constraints defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if the topology of `cs` is incompatible with `topology`.

8.9.3.4 Polyhedron::Polyhedron (Topology *topol*, const GenSys & *gs*) `[protected]`

Builds a polyhedron from a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

topol The topology of the polyhedron;

gs The system of generators defining the polyhedron.

Exceptions:

std::invalid_argument thrown if if the topology of `gs` is incompatible with `topol`, or if the system of generators is not empty but has no points.

8.9.3.5 Polyhedron::Polyhedron (Topology *topol*, GenSys & *gs*) [protected]

Builds a polyhedron recycling a system of generators.

The polyhedron inherits the space dimension of the generator system.

Parameters:

topol The topology of the polyhedron;

gs The system of generators defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

Exceptions:

std::invalid_argument thrown if if the topology of `gs` is incompatible with `topol`, or if the system of generators is not empty but has no points.

8.9.3.6 template<typename Box> Polyhedron::Polyhedron (Topology *topol*, const Box & *box*) [protected]

Builds a polyhedron out of a generic, interval-based bounding box.

Parameters:

topol The topology of the polyhedron;

box The bounding box representing the polyhedron to be built.

Exceptions:

std::invalid_argument thrown if `box` has intervals that are incompatible with `topol`.

The template class Box must provide the following methods.

```
dimension_type space_dimension() const
```

returns the dimension of the vector space enclosing the polyhedron represented by the bounding box.

```
bool is_empty() const
```

returns `true` if and only if the bounding box describes the empty set. The `is_empty()` method will always be called before the methods below. However, if `is_empty()` returns `true`, none of the functions below will be called.

```
bool get_lower_bound(dimension_type k, bool closed,
                     Integer& n, Integer& d) const
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from below, simply return `false`. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to `true` if the the lower boundary of I is closed and is set to `false` otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the greatest lower bound of I. The fraction n/d is in canonical form if and only if n and d have no common factors and d is positive, $0/1$ being the unique representation for zero.

```
bool get_upper_bound(dimension_type k, bool closed,
                     Integer& n, Integer& d) const
```

Let I the interval corresponding to the `k`-th dimension. If I is not bounded from above, simply return `false`. Otherwise, set `closed`, `n` and `d` as follows: `closed` is set to `true` if the the upper boundary of I is closed and is set to `false` otherwise; `n` and `d` are assigned the integers n and d such that the canonical fraction n/d corresponds to the least upper bound of I.

8.9.4 Member Function Documentation

8.9.4.1 Poly_Con_Relation Polyhedron::relation_with (const Constraint & *c*) const

Returns the relations holding between the polyhedron `*this` and the constraint `c`.

Exceptions:

std::invalid_argument thrown if `*this` and constraint `c` are dimension-incompatible.

8.9.4.2 Poly_Gen_Relation Polyhedron::relation_with (const Generator & *g*) const

Returns the relations holding between the polyhedron `*this` and the generator `g`.

Exceptions:

std::invalid_argument thrown if `*this` and generator `g` are dimension-incompatible.

8.9.4.3 bool Polyhedron::is_disjoint_from (const Polyhedron & *y*) const

Returns `true` if and only if `*this` and `y` are disjoint.

Exceptions:

std::invalid_argument thrown if `x` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.4 bool Polyhedron::bounds_from_above (const LinExpression & *expr*) const

Returns `true` if and only if `expr` is bounded from above in `*this`.

Exceptions:

std::invalid_argument thrown if `expr` and `*this` are dimension-incompatible.

8.9.4.5 bool Polyhedron::bounds_from_below (const LinExpression & *expr*) const

Returns `true` if and only if `expr` is bounded from below in `*this`.

Exceptions:

std::invalid_argument thrown if `expr` and `*this` are dimension-incompatible.

8.9.4.6 bool Polyhedron::contains (const Polyhedron & *y*) const

Returns `true` if and only if `*this` contains `y`.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.7 bool Polyhedron::strictly_contains (const Polyhedron & *y*) const

Returns `true` if and only if `*this` strictly contains `y`.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.8 template<typename Box> void Polyhedron::shrink_bounding_box (Box & *box*, Complexity_Class *complexity* = ANY) const

Uses $*$`this` to shrink a generic, interval-based bounding box.

Parameters:

box The bounding box to be shrunk.

complexity The complexity class of the algorithm to be used.

The template class Box must provide the following methods, whose return value, if any, is simply ignored.

```
set_empty()
```

causes the box to become empty, i.e., to represent the empty set.

```
raise_lower_bound(dimension_type k, bool closed,
                  const Integer& n, const Integer& d)
```

intersects the interval corresponding to the `k`-th dimension with $[n/d, +\infty)$ if `closed` is `true`, with $(n/d, +\infty)$ if `closed` is `false`.

```
lower_upper_bound(dimension_type k, bool closed,
                  const Integer& n, const Integer& d)
```

intersects the interval corresponding to the `k`-th dimension with $(-\infty, n/d]$ if `closed` is `true`, with $(-\infty, n/d)$ if `closed` is `false`.

The function `raise_lower_bound(k, closed, n, d)` will be called at most once for each possible value for `k` and for all such calls the fraction n/d will be in canonical form, that is, n and d have no common factors and d is positive, $0/1$ being the unique representation for zero. The same guarantee is offered for the function `lower_upper_bound(k, closed, n, d)`.

8.9.4.9 bool Polyhedron::OK (bool *check_not_empty* = false) const

Checks if all the invariants are satisfied.

Parameters:

check_not_empty `true` if and only if, in addition to checking the invariants, $*$`this` must be checked to be not empty.

Returns:

`true` if and only if $*$`this` satisfies all the invariants and either `check_not_empty` is `false` or $*$`this` is not empty.

The check is performed so as to intrude as little as possible. If the library has been compiled with run-time assertions enabled, error messages are written on `std::cerr` in case invariants are violated. This is useful for the purpose of debugging the library.

8.9.4.10 void Polyhedron::add_constraint (const Constraint & *c*)

Adds a copy of constraint `c` to the system of constraints of $*$`this` (without minimizing the result).

Exceptions:

std::invalid_argument thrown if $*$`this` and constraint `c` are topology-incompatible or dimension-incompatible.

8.9.4.11 bool Polyhedron::add_constraint_and_minimize (const Constraint & *c*)

Adds a copy of constraint `c` to the system of constraints of `*this`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument thrown if `*this` and constraint `c` are topology-incompatible or dimension-incompatible.

8.9.4.12 void Polyhedron::add_generator (const Generator & *g*)

Adds a copy of generator `g` to the system of generators of `*this` (without minimizing the result).

Exceptions:

std::invalid_argument thrown if `*this` and generator `g` are topology-incompatible or dimension-incompatible, or if `*this` is an empty polyhedron and `g` is not a point.

8.9.4.13 bool Polyhedron::add_generator_and_minimize (const Generator & *g*)

Adds a copy of generator `g` to the system of generators of `*this`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument thrown if `*this` and generator `g` are topology-incompatible or dimension-incompatible, or if `*this` is an empty polyhedron and `g` is not a point.

8.9.4.14 void Polyhedron::add_constraints (ConSys & *cs*)

Adds the constraints in `cs` to the system of constraints of `*this`, minimizing the result.

Parameters:

cs The constraints that will be added to the current system of constraints. This parameter is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.15 bool Polyhedron::add_constraints_and_minimize (ConSys & *cs*)

Adds the constraints in `cs` to the system of constraints of `*this` (without minimizing the result).

Returns:

`false` if and only if the result is empty.

Parameters:

cs The constraints that will be added to the current system of constraints. This parameter is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if ∗`this` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.16 void Polyhedron::add_generators (GenSys & *gs*)

Adds the generators in `gs` to the system of generators of ∗`this` (without minimizing the result).

Parameters:

gs The generators that will be added to the current system of generators. This parameter is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if ∗`this` and `gs` are topology-incompatible or dimension-incompatible, or if ∗`this` is empty and the system of generators `gs` is not empty, but has no points.

8.9.4.17 bool Polyhedron::add_generators_and_minimize (GenSys & *gs*)

Adds the generators in `gs` to the system of generators of ∗`this`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

gs The generators that will be added to the current system of generators. The parameter is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if ∗`this` and `gs` are topology-incompatible or dimension-incompatible, or if ∗`this` is empty and the the system of generators `gs` is not empty, but has no points.

8.9.4.18 void Polyhedron::intersection_assign (const Polyhedron & *y*)

Assigns to ∗`this` the intersection of ∗`this` and `y`. The result is not guaranteed to be minimized.

Exceptions:

std::invalid_argument thrown if ∗`this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.19 bool Polyhedron::intersection_assign_and_minimize (const Polyhedron & *y*)

Assigns to ∗`this` the intersection of ∗`this` and `y`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument thrown if ∗`this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.20 void Polyhedron::poly_hull_assign (const Polyhedron & *y*)

Assigns to `*this` the poly-hull `*this` and `y`. The result is not guaranteed to be minimized.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.21 bool Polyhedron::poly_hull_assign_and_minimize (const Polyhedron & *y*)

Assigns to `*this` the poly-hull of `*this` and `y`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.22 void Polyhedron::poly_difference_assign (const Polyhedron & *y*)

Assigns to `*this` the poly-difference of `*this` and `y`. The result is not guaranteed to be minimized.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.23 void Polyhedron::affine_image (Variable *var*, const LinExpression & *expr*, const Integer & *denominator* = Integer_one())

Assigns to `*this` the affine image of `*this` under the function mapping variable `var` into the affine expression specified by `expr` and `denominator`.

Parameters:

var The variable to which the affine expression is assigned.

expr The numerator of the affine expression.

denominator The denominator of the affine expression (optional argument with default value 1.)

Exceptions:

std::invalid_argument thrown if `denominator` is zero or if `expr` and `*this` are dimension-incompatible or if `var` is not a dimension of `*this`.

8.9.4.24 void Polyhedron::affine_preimage (Variable *var*, const LinExpression & *expr*, const Integer & *denominator* = Integer_one())

Assigns to `*this` the affine preimage of `*this` under the function mapping variable `var` into the affine expression specified by `expr` and `denominator`.

Parameters:

var The variable to which the affine expression is substituted.

expr The numerator of the affine expression.

denominator The denominator of the affine expression (optional argument with default value 1.)

Exceptions:

std::invalid_argument thrown if `denominator` is zero or if `expr` and `*this` are dimension-incompatible or if `var` is not a dimension of `*this`.

8.9.4.25 void Polyhedron::generalized_affine_image (Variable *var*, const Relation_Symbol *relsym*, const LinExpression & *expr*, const Integer & *denominator* = Integer_one())

Assigns to `*this` the image of `*this` with respect to the generalized affine transfer function $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$, where $\bowtie$ is the relation symbol encoded by `relsym`.

Parameters:

var The left hand side variable of the generalized affine transfer function.

relsym The relation symbol.

expr The numerator of the right hand side affine expression.

denominator The denominator of the right hand side affine expression (optional argument with default value 1.)

Exceptions:

std::invalid_argument thrown if `denominator` is zero or if `expr` and `*this` are dimension-incompatible or if `var` is not a dimension of `*this` or if `*this` is a C_Polyhedron and `relsym` is a strict relation symbol.

8.9.4.26 void Polyhedron::generalized_affine_image (const LinExpression & *lhs*, const Relation_-Symbol *relsym*, const LinExpression & *rhs*)

Assigns to `*this` the image of `*this` with respect to the generalized affine transfer function $\mathrm{lhs}' \bowtie \mathrm{rhs}$, where $\bowtie$ is the relation symbol encoded by `relsym`.

Parameters:

lhs The left hand side affine expression.

relsym The relation symbol.

rhs The right hand side affine expression.

Exceptions:

std::invalid_argument thrown if `*this` is dimension-incompatible with `lhs` or `rhs` or if `*this` is a C_Polyhedron and `relsym` is a strict relation symbol.

8.9.4.27 void Polyhedron::time_elapse_assign (const Polyhedron & *y*)

Assigns to `*this` the result of computing the time-elapse between `*this` and `y`.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.28 void Polyhedron::BHRZ03_widening_assign (const Polyhedron & *y*, unsigned * *tp* = 0)

Assigns to `*this` the result of computing the BHRZ03-widening between `*this` and `y`.

Parameters:

y A polyhedron that *must* be contained in `*this`.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.29 void Polyhedron::limited_BHRZ03_extrapolation_assign (const Polyhedron & *y*, const ConSys & *cs*, unsigned $*$ *tp* = 0)

Improves the result of the BHRZ03-widening computation by also enforcing those constraints in `cs` that are satisfied by all the points of $*$`this`.

Parameters:

y A polyhedron that *must* be contained in $*$`this`.

cs The system of constraints used to improve the widened polyhedron.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if $*$`this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.30 void Polyhedron::bounded_BHRZ03_extrapolation_assign (const Polyhedron & *y*, const ConSys & *cs*, unsigned $*$ *tp* = 0)

Improves the result of the BHRZ03-widening computation by also enforcing those constraints in `cs` that are satisfied by all the points of $*$`this`, plus all the constraints of the form $\pm x \leq r$ and $\pm x < r$, with $r \in \mathbb{Q}$, that are satisfied by all the points of $*$`this`.

Parameters:

y A polyhedron that *must* be contained in $*$`this`.

cs The system of constraints used to improve the widened polyhedron.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if $*$`this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.31 void Polyhedron::H79_widening_assign (const Polyhedron & *y*, unsigned $*$ *tp* = 0)

Assigns to $*$`this` the result of computing the H79-widening between $*$`this` and `y`.

Parameters:

y A polyhedron that *must* be contained in $*$`this`.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if $*$`this` and `y` are topology-incompatible or dimension-incompatible.

8.9.4.32 void Polyhedron::limited_H79_extrapolation_assign (const Polyhedron & *y*, const ConSys & *cs*, unsigned ∗ *tp* = 0)

Improves the result of the H79-widening computation by also enforcing those constraints in `cs` that are satisfied by all the points of ∗`this`.

Parameters:

y A polyhedron that *must* be contained in ∗`this`.

cs The system of constraints used to improve the widened polyhedron.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if ∗`this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.33 void Polyhedron::bounded_H79_extrapolation_assign (const Polyhedron & *y*, const ConSys & *cs*, unsigned ∗ *tp* = 0)

Improves the result of the H79-widening computation by also enforcing those constraints in `cs` that are satisfied by all the points of ∗`this`, plus all the constraints of the form $\pm x \leq r$ and $\pm x < r$, with $r \in \mathbb{Q}$, that are satisfied by all the points of ∗`this`.

Parameters:

y A polyhedron that *must* be contained in ∗`this`.

cs The system of constraints used to improve the widened polyhedron.

tp An optional pointer to an unsigned variable storing the number of available tokens (to be used when applying the widening with tokens delay technique).

Exceptions:

std::invalid_argument thrown if ∗`this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.9.4.34 void Polyhedron::add_dimensions_and_embed (dimension_type *m*)

Adds `m` new dimensions and embeds the old polyhedron into the new space.

Parameters:

m The number of dimensions to add.

The new dimensions will be those having the highest indexes in the new polyhedron, which is characterized by a system of constraints in which the variables running through the new dimensions are not constrained. For instance, when starting from the polyhedron $\mathcal{P} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the polyhedron

$$\bigl\{\, (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x, y)^{\mathrm{T}} \in \mathcal{P} \,\bigr\}.$$

8.9.4.35 void Polyhedron::add_dimensions_and_project (dimension_type *m*)

Adds `m` new dimensions to the polyhedron and does not embed it in the new space.

Parameters:

m The number of dimensions to add.

The new dimensions will be those having the highest indexes in the new polyhedron, which is characterized by a system of constraints in which the variables running through the new dimensions are all constrained to be equal to 0. For instance, when starting from the polyhedron $\mathcal{P} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the polyhedron

$$\left\{\, (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \;\middle|\; (x, y)^{\mathrm{T}} \in \mathcal{P} \,\right\}.$$

8.9.4.36 void Polyhedron::concatenate_assign (const Polyhedron & *y*)

Seeing a polyhedron as a set of tuples (its points), assigns to `*this` all the tuples that can be obtained by concatenating, in the order given, a tuple of `*this` with a tuple of `y`.

Let $P \subseteq \mathbb{R}^n$ and $Q \subseteq \mathbb{R}^m$ be the polyhedra represented, on entry, by `*this` and `y`, respectively. Upon successful completion, `*this` will represent the polyhedron $R \subseteq \mathbb{R}^{n+m}$ such that

$$R \stackrel{\mathrm{def}}{=} \left\{\, (x_1, \ldots, x_n, y_1, \ldots, y_m)^{\mathrm{T}} \;\middle|\; (x_1, \ldots, x_n)^{\mathrm{T}} \in P, (y_1, \ldots, y_m)^{\mathrm{T}} \in Q \,\right\}.$$

Another way of seeing it is as follows: first increases the space dimension of `*this` by adding `y.space_dimension()` new dimensions; then adds to the system of constraints of `*this` a renamed-apart version of the constraints of `y`.

Exceptions:

std::invalid_argument thrown if `*this` and `y` are topology-incompatible.

8.9.4.37 void Polyhedron::remove_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions.

Parameters:

to_be_removed The set of Variable objects corresponding to the dimensions to be removed.

Exceptions:

std::invalid_argument thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `to_be_removed`.

8.9.4.38 void Polyhedron::remove_higher_dimensions (dimension_type *new_dimension*)

Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.

Exceptions:

std::invalid_argument thrown if `new_dimensions` is greater than the space dimension of `*this`.

8.9.4.39 template<typename PartialFunction> void Polyhedron::map_dimensions (const PartialFunction & *pfunc*)

Remaps the dimensions of the vector space according to a partial function.

Parameters:

pfunc The partial function specifying the destiny of each dimension.

The template class PartialFunction must provide the following methods.

```
bool has_empty_codomain() const
```

returns `true` if and only if the represented partial function has an empty codomain (i.e., it is always undefined). The `has_empty_codomain()` method will always be called before the methods below. However, if `has_empty_codomain()` returns `true`, none of the functions below will be called.

```
dimension_type max_in_codomain() const
```

returns the maximum value that belongs to the codomain of the partial function.

```
bool maps(dimension_type i, dimension_type& j) const
```

Let f be the represented function and k be the value of `i`. If f is defined in k, then $f(k)$ is assigned to `j` and `true` is returned. If f is undefined in k, then `false` is returned.

The result is undefined if `pfunc` does not encode a partial function with the properties described in the specification of the mapping operator.

8.9.4.40 void Polyhedron::swap (Polyhedron & *y*)

Swaps $*$`this` with polyhedron `y`. ($*$`this` and `y` can be dimension-incompatible.).

Exceptions:

std::invalid_argument thrown if `x` and `y` are topology-incompatible.

8.9.5 Friends And Related Function Documentation

8.9.5.1 std::ostream & operator<< (std::ostream & *s*, const Polyhedron & *ph*) `[related]`

Output operator.

Writes a textual representation of `ph` on `s`: `false` is written if `ph` is an empty polyhedron; `true` is written if `ph` is a universe polyhedron; a minimized system of constraints defining `ph` is written otherwise, all constraints in one row separated by ", ".

8.9.5.2 bool operator== (const Polyhedron & *x*, const Polyhedron & *y*) `[related]`

Returns `true` if and only if `x` and `y` are the same polyhedron.

Note that `x` and `y` may be topology- and/or dimension-incompatible polyhedra: in those cases, the value `false` is returned.

8.9.5.3 bool operator!= (const Polyhedron & *x*, const Polyhedron & *y*) `[related]`

Returns `true` if and only if `x` and `y` are different polyhedra.

Note that `x` and `y` may be topology- and/or dimension-incompatible polyhedra: in those cases, the value `true` is returned.

8.10 PowerSet< CS > Class Template Reference

The powerset construction on constraint systems.

Public Member Functions

- PowerSet (dimension_type num_dimensions=0, bool universe=true)

  Builds a universe (top) or empty (bottom) PowerSet.

- **PowerSet** (const ConSys &cs)

  Creates a PowerSet with the same information contents as `cs`.

- PowerSet & **inject** (const CS &c)

  Injects `c` *into* ∗`this`.

- void **upper_bound_assign** (const PowerSet &y)

  Assigns to ∗`this` *an upper bound of* ∗`this` *and* `y`.

- void **meet_assign** (const PowerSet &y)

  Assigns to ∗`this` *the meet of* ∗`this` *and* `y`.

- bool **definitely_entails** (const PowerSet &y) const

  Returns `true` *if* ∗`this` *definitely entails* `y`. *Returns* `false` *if* ∗`this` *may not entail* `y` *(i.e., if* ∗`this` *does not entail* `y` *or if entailment could not be decided).*

- dimension_type **space_dimension** () const

  Returns the dimension of the vector space enclosing ∗`this`.

- void add_constraint (const Constraint &c)

  Intersects ∗`this` *with (a copy of) constraint* `c`.

- void add_constraints (ConSys &cs)

  Intersects ∗`this` *with the constraints in* `cs`.

- void **add_dimensions_and_embed** (dimension_type m)

  Adds `m` *new dimensions and embeds the old polyhedron into the new space.*

- void **add_dimensions_and_project** (dimension_type m)

  Adds `m` *new dimensions to the polyhedron and does not embed it in the new space.*

- void remove_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified dimensions.

- void remove_higher_dimensions (dimension_type new_dimension)

Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.

- void H79_extrapolation_assign (const PowerSet &y)

  Assigns to `*this` *the result of computing the H79-widening between* `*this` *and* `y`.

- void limited_H79_extrapolation_assign (const PowerSet &y, const ConSys &cs)

  Limits the H79-widening computation between `*this` *and* `y` *by enforcing constraints* `cs` *and assigns the result to* `*this`.

- bool **OK** () const

  Checks if all the invariants are satisfied.

Friends

- CS project (const PowerSet &x)
- int lcompare (const PowerSet &x, const PowerSet &y)

Related Functions

(Note that these are not member functions.)

- PowerSet< CS > operator+ (const PowerSet< CS > &, const PowerSet< CS > &)
- PowerSet< CS > operator * (const PowerSet< CS > &, const PowerSet< CS > &)
- bool operator== (const PowerSet< CS > &x, const PowerSet< CS > &y)
- std::ostream & operator<< (std::ostream &, const PowerSet< CS > &)

8.10.1 Detailed Description

template<typename CS> class PowerSet< CS >

The powerset construction on constraint systems.

8.10.2 Constructor & Destructor Documentation

8.10.2.1 template<typename CS> PowerSet< CS >::PowerSet (dimension_type *num_dimensions* = 0, bool *universe* = true) [explicit]

Builds a universe (top) or empty (bottom) PowerSet.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the powerset.

universe If `true`, a universe PowerSet is built; an empty PowerSet is built otherwise.

8.10.3 Member Function Documentation

8.10.3.1 template<typename CS> void PowerSet< CS >::add_constraint (const Constraint & *c*)

Intersects `*this` with (a copy of) constraint `c`.

Exceptions:

std::invalid_argument thrown if ∗`this` and constraint `c` are topology-incompatible or dimension-incompatible.

8.10.3.2 template<typename CS> void PowerSet< CS >::add_constraints (ConSys & *cs*)

Intersects ∗`this` with the constraints in `cs`.

Parameters:

cs The constraints to intersect with. This parameter is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if ∗`this` and `cs` are topology-incompatible or dimension-incompatible.

8.10.3.3 template<typename CS> void PowerSet< CS >::remove_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions.

Parameters:

to_be_removed The set of Variable objects corresponding to the dimensions to be removed.

Exceptions:

std::invalid_argument thrown if ∗`this` is dimension-incompatible with one of the Variable objects contained in `to_be_removed`.

8.10.3.4 template<typename CS> void PowerSet< CS >::remove_higher_dimensions (dimension_type *new_dimension*)

Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.

Exceptions:

std::invalid_argument thrown if `new_dimensions` is greater than the space dimension of ∗`this`.

8.10.3.5 template<typename CS> void PowerSet< CS >::H79_extrapolation_assign (const PowerSet< CS > & *y*)

Assigns to ∗`this` the result of computing the H79-widening between ∗`this` and `y`.

Parameters:

y A polyhedron that *must* be contained in ∗`this`.

Exceptions:

std::invalid_argument thrown if ∗`this` and `y` are topology-incompatible or dimension-incompatible.

8.10.3.6 template<typename CS> void PowerSet< CS >::limited_H79_extrapolation_assign (const PowerSet< CS > & *y*, const ConSys & *cs*)

Limits the H79-widening computation between `*this` and `y` by enforcing constraints `cs` and assigns the result to `*this`.

Parameters:

y A polyhedron that *must* be contained in `*this`.

cs The system of constraints that limits the widened polyhedron. It is not declared `const` because it can be modified.

Exceptions:

std::invalid_argument thrown if `*this`, `y` and `cs` are topology-incompatible or dimension-incompatible.

8.10.4 Friends And Related Function Documentation

8.10.4.1 template<typename CS> CS project (const PowerSet< CS > & *x*) `[friend]`

<CS>

8.10.4.2 template<typename CS> int lcompare (const PowerSet< CS > & *x*, const PowerSet< CS > & *y*) `[friend]`

<CS>

8.10.4.3 template<typename CS> PowerSet< CS > operator+ (const PowerSet< CS > &, const PowerSet< CS > &) `[related]`

<CS>

8.10.4.4 template<typename CS> PowerSet< CS > operator * (const PowerSet< CS > &, const PowerSet< CS > &) `[related]`

<CS>

8.10.4.5 template<typename CS> bool operator== (const PowerSet< CS > & *x*, const PowerSet< CS > & *y*) `[related]`

<CS>

8.10.4.6 template<typename CS> std::ostream & operator<< (std::ostream &, const PowerSet< CS > &) `[related]`

<CS>

8.11 Variable Class Reference

A dimension of the space.

Public Types

- typedef void **Output_Function_Type** (std::ostream &s, Variable v)

  Type of output functions.

Public Member Functions

- **Variable** (dimension_type i)

  Builds the variable corresponding to the Cartesian axis of index `i`.

- dimension_type **id** () const

  Returns the index of the Cartesian axis associated to the variable.

Static Public Member Functions

- void **set_output_function** (Output_Function_Type ∗p)

  Set the output function to be used for printing Variable objects.

- Output_Function_Type ∗ **get_output_function** ()

  Returns the pointer to the current output function.

Related Functions

(Note that these are not member functions.)

- std::ostream & **operator**<< (std::ostream &s, Variable v)

  Output operator.

- bool **less** (Variable v, Variable w)

  Defines a total ordering on variables.

8.11.1 Detailed Description

A dimension of the space.

An object of the class Variable represents a dimension of the space, that is one of the Cartesian axes. Variables are used as base blocks in order to build more complex linear expressions. Each variable is identified by a non-negative integer, representing the index of the corresponding Cartesian axis (the first axis has index 0).

Note that the "meaning" of an object of the class Variable is completely specified by the integer index provided to its constructor: be careful not to be mislead by C++ language variable names. For instance, in the following example the linear expressions `e1` and `e2` are equivalent, since the two variables `x` and `z` denote the same Cartesian axis.

```
Variable x(0);
Variable y(1);
Variable z(0);
LinExpression e1 = x + y;
LinExpression e2 = y + z;
```

8.12 Compare Struct Reference

Binary predicate defining the total ordering on variables.

Public Member Functions

- bool **operator()** (Variable x, Variable y) const

  Returns `true` *if and only if* `x` *comes before* `y`.

8.12.1 Detailed Description

Binary predicate defining the total ordering on variables.

Index