The Parma Polyhedra Library User's Manual[*] (version 0.9)

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

based on previous work also by

Elisa Ricci

and

Sara Bonini
Andrea Pescetti
Angela Stazzone
Tatiana Zolo

March 12, 2006

[*]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"; MURST project "Abstract Interpretation: Design and Applications".

[†]bagnara@cs.unipr.it, Department of Mathematics, University of Parma, Italy.

[‡]hill@comp.leeds.ac.uk, School of Computing, University of Leeds, U.K.

[§]zaffanella@cs.unipr.it, Department of Mathematics, University of Parma, Italy.

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

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

Contents

1 General Information on the PPL

1.1 The Main Features

The Parma Polyhedra Library (PPL) is a modern C++ library for the manipulation of numerical information that can be represented by points in some n-dimensional vector space. For instance, one of the key domains the PPL supports is that of rational convex polyhedra (Section Convex Polyhedra). Such domains are 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 has interfaces to other programming languages: including C and a number of Prolog systems;
- it is free software: distributed under the terms of the GNU General Public License.

In addition to the basic domains, we also provide generic support for constructing new domains from pre-existing domains. The following domains and domain constructors are provided by the PPL:

- the domain of topologically closed, rational convex polyhedra;
- the domain of rational convex polyhedra that are not necessarily closed;
- the domain of topologically closed, rational bounded difference shapes;
- the powerset construction;
- the powerset construction, instantiated for convex polyhedra.

In the following sections we describe these domains and domain constructors together with their representations and operations that are available to the PPL user.

In the final section of this chapter (Section Using the Library), we provide some additional advice on the use of the library.

1.2 Convex Polyhedra

In this section we introduce convex polyhedra, as considered by the library, in more detail. For more information about the definitions and results stated here see [BRZH02b], [Fuk98], [NW88], and [Wil93].

1.2.1 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: $S_1 + S_2 = \{\, \boldsymbol{v}_1 + \boldsymbol{v}_2 \mid \boldsymbol{v}_1 \in S_1, \boldsymbol{v}_2 \in S_2 \,\}$.

1.2.2 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 $\boldsymbol{a} = \mathbf{0}$, the constraint $\langle \mathbf{0}, \boldsymbol{x} \rangle \bowtie b$ 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 $\varnothing$.

1.2.3 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 $\mathcal{P} = \varnothing$. 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 $\mathcal{P} = \varnothing$. 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 $\varnothing$ 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 $\mathbb{P}_n$.

Note:

In the following, we will usually specify operators on the domain $\mathbb{P}_n$ of NNC polyhedra. Unless an explicit distinction is made, these operators are provided with the same specification when applied to the domain $\mathbb{CP}_n$ of topologically closed polyhedra. The implementation maintains a clearer separation between the two domains of polyhedra (see Topologies and Topological-compatibility): while computing polyhedra in $\mathbb{P}_n$ may provide more precise results, polyhedra in $\mathbb{CP}_n$ can be represented and manipulated more efficiently. As a rule of thumb, if your application will only manipulate polyhedra that are topologically closed, then it should use the simpler domain $\mathbb{CP}_n$. Using NNC polyhedra is only recommended if you are going to actually benefit from the increased accuracy.

1.2.4 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.

1.3.1 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

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

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.

1.3.2 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.

1.3.3 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).

1.3.4 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

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

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

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

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 $\mathcal{P}$.

1.3.5 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 $C' \subseteq C$.

1.3.6 Double Description

Any NNC polyhedron $\mathcal{P}$ can be described by using a constraint system $\mathcal{C}$, a generator system $\mathcal{G}$, 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.

1.3.7 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 topological 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.

1.3.8 Space Dimensions and Dimension Compatibility

The *space dimension* of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ (resp., a C polyhedron $\mathcal{P} \in \mathbb{CP}_n$) 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.

1.3.9 Affine Independence and Affine Dimension

A finite set of points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \subseteq \mathbb{R}^n$ is *affinely independent* if, for all $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$, the system of equations

$$\sum_{i=1}^{k} \lambda_i \boldsymbol{x}_i = \boldsymbol{0}, \qquad \sum_{i=1}^{k} \lambda_i = 0$$

implies that, for each $i = 1, \ldots, k$, $\lambda_i = 0$.

The maximum number of affinely independent points in $\mathbb{R}^n$ is $n + 1$.

A *non-empty* NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ has *affine dimension* $k \in \mathbb{N}$, denoted by $\dim(\mathcal{P}) = k$, if the maximum number of affinely independent points in $\mathcal{P}$ is $k + 1$.

We remark that the above definition only applies to polyhedra that are not empty, so that $0 \leq \dim(\mathcal{P}) \leq n$. By convention, the affine dimension of an empty polyhedron is 0 (even though the "natural" generalization of the definition above would imply that the affine dimension of an empty polyhedron is -1).

Note:

The affine dimension $k \leq n$ of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ must not be confused with the space dimension n of $\mathcal{P}$, which is the dimension of the enclosing vector space $\mathbb{R}^n$. In particular, we can have $\dim(\mathcal{P}) \neq \dim(\mathcal{Q})$ even though $\mathcal{P}$ and $\mathcal{Q}$ are dimension-compatible; and vice versa, $\mathcal{P}$ and $\mathcal{Q}$ may be dimension-incompatible polyhedra even though $\dim(\mathcal{P}) = \dim(\mathcal{Q})$.

1.3.10 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.

1.4.1 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 $\mathbb{CP}_n$.

1.4.2 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 smallest convex polyhedron containing 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.

1.4.3 Concatenating Polyhedra

Viewing a polyhedron as a set of tuples (its points), it is sometimes useful to consider the set of tuples obtained by concatenating an ordered pair of polyhedra. Formally, the *concatenation* of the polyhedra $\mathcal{P} \in \mathbb{P}_n$ and $\mathcal{Q} \in \mathbb{P}_m$ (taken in this order) is the polyhedron $\mathcal{R} \in \mathbb{P}_{n+m}$ such that

$$\mathcal{R} \stackrel{\mathrm{def}}{=} \Bigl\{ (x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})^{\mathrm{T}} \in \mathbb{R}^{n+m} \Bigm| (x_0, \ldots, x_{n-1})^{\mathrm{T}} \in \mathcal{P}, (y_0, \ldots, y_{m-1})^{\mathrm{T}} \in \mathcal{Q} \Bigr\}.$$

Another way of seeing it is as follows: first embed polyhedron $\mathcal{P}$ into a vector space of dimension $n + m$ and then add a suitably renamed-apart version of the constraints defining $\mathcal{Q}$.

1.4.4 Adding New Dimensions to the Vector Space

The library provides two operators for adding a number i of space dimensions to an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$, therefore transforming it into a new NNC polyhedron $\mathcal{Q} \in \mathbb{P}_{n+i}$. In both cases, the added dimensions of the vector space are those having the highest indices.

The operator `add_space_dimensions_and_embed` *embeds* the polyhedron $\mathcal{P}$ into the new vector space of dimension $i + n$ and returns 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 space 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 `add_space_dimensions_and_project` *projects* the polyhedron $\mathcal{P}$ into the new vector space of dimension $i + n$ and returns 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 space dimension, the result will be the polyhedron

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

1.4.5 Removing Dimensions from the Vector Space

The library provides two operators for removing space dimensions from an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$, therefore transforming it into a new NNC polyhedron $\mathcal{Q} \in \mathbb{P}_m$ where $m \leq n$.

Given a set of variables, the operator `remove_space_dimensions` removes all the space dimensions specified by the variables in the 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

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

Given a space dimension m less than or equal to that of the polyhedron, the operator `remove_higher_space_dimensions` removes the space dimensions having indices 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

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

1.4.6 Mapping the Dimensions of the Vector Space

The operator `map_space_dimensions` provided by the library maps the dimensions of the vector space $\mathbb{R}^n$ according to a partial injective function $\rho \colon \{0, \ldots, n-1\} \rightarrowtail \mathbb{N}$ such that $\rho\bigl(\{0, \ldots, n-1\}\bigr) = \{0, \ldots, m-1\}$ 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

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

1.4.7 Expanding One Dimension of the Vector Space to Multiple Dimensions

The operator `expand_space_dimension` provided by the library adds m new space dimensions to a polyhedron $\mathcal{P} \in \mathbb{P}_n$, with $n > 0$, so that dimensions n, $n+1$, …, $n+m-1$ of the result $\mathcal{Q}$ are exact copies of the i-th space dimension of $\mathcal{P}$. More formally,

$$\mathcal{Q} \stackrel{\mathrm{def}}{=} \left\{ \boldsymbol{u} \in \mathbb{R}^{n+m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v}, \boldsymbol{w} \in \mathcal{P} \;.\; u_i = v_i \\ \qquad \wedge\, \forall j = n, n+1, \ldots, n+m-1 : u_j = w_i \\ \qquad \wedge\, \forall k = 0, \ldots, n-1 : k \neq i \implies u_k = v_k = w_k \end{array} \right\}.$$

This operation has been proposed in [GDMDRS04].

1.4.8 Folding Multiple Dimensions of the Vector Space into One Dimension

The operator `fold_space_dimensions` provided by the library, given a polyhedron $\mathcal{P} \in \mathbb{P}_n$, with $n > 0$, folds a set of space dimensions $J = \{j_0, \ldots, j_{m-1}\}$, with $m < n$ and $j < n$ for each $j \in J$, into space dimension $i < n$, where $i \notin J$. The result is given by

$$\mathcal{Q} \stackrel{\mathrm{def}}{=} \biguplus_{d=0}^{m} \mathcal{Q}_d$$

where

$$\mathcal{Q}_m \stackrel{\mathrm{def}}{=} \left\{ \boldsymbol{u} \in \mathbb{R}^{n-m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v} \in \mathcal{P} \;.\; u_{i'} = v_i \\ \qquad \wedge\, \forall k = 0, \ldots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\}$$

and, for $d = 0$, …, $m-1$,

$$\mathcal{Q}_d \stackrel{\mathrm{def}}{=} \left\{ \boldsymbol{u} \in \mathbb{R}^{n-m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v} \in \mathcal{P} \;.\; u_{i'} = v_{j_d} \\ \qquad \wedge\, \forall k = 0, \ldots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\},$$

and, finally, for $k = 0$, …, $n-1$,

$$k' \stackrel{\mathrm{def}}{=} k - \#\{\, j \in J \mid k > j \,\},$$

($\#\, S$ denotes the cardinality of the finite set S).

This operation has been proposed in [GDMDRS04].

1.4.9 Images and Preimages of Affine Transfer Relations

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

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

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

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

If $n = m$, then the relation ϕ is said to be *space dimension preserving*.

The relation $\phi \subseteq \mathbb{R}^n \times \mathbb{R}^m$ is said to be an *affine relation* if there exists $\ell \in \mathbb{N}$ such that

$$\forall \boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^m : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff \bigwedge_{i=1}^{\ell} \bigl(\langle \boldsymbol{c}_i, \boldsymbol{w} \rangle \bowtie_i \langle \boldsymbol{a}_i, \boldsymbol{v} \rangle + b_i\bigr),$$

where $\boldsymbol{a}_i \in \mathbb{R}^n$, $\boldsymbol{c}_i \in \mathbb{R}^m$, $b_i \in \mathbb{R}$ and $\bowtie_i \in \{<, \leq, =, \geq, >\}$, for each $i = 1, \ldots, \ell$.

As a special case, the relation $\phi \subseteq \mathbb{R}^n \times \mathbb{R}^m$ is an *affine function* if and only 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,

$$\forall \boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^m : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff \boldsymbol{w} = A\boldsymbol{v} + \boldsymbol{b}.$$

The set $\mathbb{P}_n$ of NNC polyhedra is closed under the application of images and preimages of any space dimension preserving affine relation. The same property holds for the set $\mathbb{CP}_n$ of closed polyhedra, provided the affine relation makes no use of the strict relation symbols $<$ and $>$. Images and preimages of affine relations can be used to model several kinds of transition relations, including deterministic assignments of affine expressions, (affinely constrained) nondeterministic assignments and affine conditional guards.

A space dimension preserving relation $\phi \subseteq \mathbb{R}^n \times \mathbb{R}^n$ can be specified by means of a shorthand notation:

- the vector $\boldsymbol{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$ of *unprimed* variables is used to represent the space dimensions of the domain of ϕ;
- the vector $\boldsymbol{x}' = (x'_0, \ldots, x'_{n-1})^{\mathrm{T}}$ of *primed* variables is used to represent the space dimensions of the range of ϕ;
- any primed variable that "does not occur" in the shorthand specification is meant to be *unaffected* by the relation; namely, for each index $i \in \{0, \ldots, n-1\}$, if in the syntactic specification of the relation the primed variable x'_i only occurs (if ever) with coefficient 0, then it is assumed that the specification also contains the constraint $x'_i = x_i$.

As an example, assuming $\phi \subseteq \mathbb{R}^3 \times \mathbb{R}^3$, the notation $x'_0 - x'_2 \geq 2x_0 - x_1$, where the primed variable x'_1 does not occur, is meant to specify the affine relation defined by

$$\forall \boldsymbol{v} \in \mathbb{R}^3, \boldsymbol{w} \in \mathbb{R}^3 : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff (w_0 + w_2 \geq 2v_0 - v_1) \wedge (w_1 = v_1).$$

The same relation is specified by $x'_0 + 0 \cdot x'_1 - x'_2 \geq 2x_0 - x_1$, since x'_1 occurs with coefficient 0.

The library allows for the computation of images and preimages of polyhedra under restricted subclasses of space dimension preserving affine relations, as described in the following.

1.4.10 Single-Update Affine Functions.

Given a primed variable x'_k and an unprimed affine expression $\langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$, the *affine function* $\phi = \bigl(x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b\bigr) \colon \mathbb{R}^n \to \mathbb{R}^n$ is defined by

$$\forall \boldsymbol{v} \in \mathbb{R}^n : \phi(\boldsymbol{v}) = A\boldsymbol{v} + \boldsymbol{b},$$

where

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

and the a_i (resp., b) occur in the $(k+1)$st row in A (resp., position in $\boldsymbol{b}$). Thus function ϕ maps any vector $(v_0, \dots, v_{n-1})^{\mathrm{T}}$ to

$$\Bigl(v_0, \dots, \bigl(\textstyle\sum_{i=0}^{n-1} a_i v_i + b\bigr), \dots, v_{n-1}\Bigr)^{\mathrm{T}}.$$

The *affine image* operator computes the affine image of a polyhedron $\mathcal{P}$ under $x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$. For instance, suppose the polyhedron $\mathcal{P}$ to be transformed is the square in $\mathbb{R}^2$ generated by the set of points $\bigl\{(0,0)^{\mathrm{T}}, (0,3)^{\mathrm{T}}, (3,0)^{\mathrm{T}}, (3,3)^{\mathrm{T}}\bigr\}$. Then, if the primed variable is x_0 and the affine expression is $x_0 + 2x_1 + 4$ (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 $\bigl\{(4,0)^{\mathrm{T}}, (10,3)^{\mathrm{T}}, (7,0)^{\mathrm{T}}, (13,3)^{\mathrm{T}}\bigr\}$ with height equal to the side of the square and oblique sides parallel to the line $x_0 - 2x_1$. If the primed variable is as before (i.e., $k = 0$) but the affine 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 a polyhedron $\mathcal{P}$ under $x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$. For instance, suppose now that we apply the affine preimage operator as given in the first example using primed variable x_0 and affine 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 primed variable x_0 and affine expression x_1 to $\mathcal{P}_2$, then the resulting polyhedron is the stripe obtained by adding the line $(1,0)^{\mathrm{T}}$ to polyhedron $\mathcal{P}_2$.

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

1.4.11 Single-Update Bounded Affine Relations.

Given a primed variable x'_k and two unprimed affine expressions $\mathrm{lb} = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$ and $\mathrm{ub} = \langle \boldsymbol{c}, \boldsymbol{x} \rangle + d$, the *bounded affine relation* $\phi = (\mathrm{lb} \leq x'_k \leq \mathrm{ub})$ is defined as

$$\forall \boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^n : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff \bigl(\langle \boldsymbol{a}, \boldsymbol{v} \rangle + b \leq w_k \leq \langle \boldsymbol{c}, \boldsymbol{v} \rangle + d\bigr) \wedge \Bigl(\bigwedge_{0 \leq i < n, i \neq k} w_i = v_i\Bigr).$$

1.4.12 Generalized Affine Relations.

Similarly, the *generalized affine relation* $\phi = (\mathrm{lhs}' \bowtie \mathrm{rhs})$, where $\mathrm{lhs} = \langle \boldsymbol{c}, \boldsymbol{x} \rangle + d$ and $\mathrm{rhs} = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$ are affine expressions and $\bowtie \in \{<, \leq, =, \geq, >\}$ is a relation symbol, is defined as

$$\forall \boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^n : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff \bigl(\langle \boldsymbol{c}, \boldsymbol{w} \rangle + d \bowtie \langle \boldsymbol{a}, \boldsymbol{v} \rangle + b\bigr) \wedge \Bigl(\bigwedge_{0 \leq i < n, c_i = 0} w_i = v_i\Bigr).$$

When $\mathrm{lhs} = x_k$ and $\bowtie \in \{=\}$, then the above affine relation becomes equivalent to the single-update affine function $x'_k = \mathrm{rhs}$ (hence the name given to this operator). It is worth stressing that the notation is not symmetric, because the variables occurring in expression lhs are interpreted as primed variables, whereas those occurring in rhs are unprimed; for instance, the transfer relations $\mathrm{lhs}' \leq \mathrm{rhs}$ and $\mathrm{rhs}' \geq \mathrm{lhs}$ are not equivalent in general.

1.4.13 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

$$\bigl\{\, \boldsymbol{p} + \lambda \boldsymbol{q} \in \mathbb{R}^n \bigm| \boldsymbol{p} \in \mathcal{P}, \boldsymbol{q} \in \mathcal{Q}, \lambda \in \mathbb{R}_+ \,\bigr\}.$$

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.

1.4.14 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 $\mathcal{P}$.

1.4.15 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 $\boldsymbol{e}_i = (0, \ldots, 1, \ldots, 0)^{\mathrm{T}}$ 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.

1.4.16 Widening Operators

The library provides two widening operators for the domain of 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]. Note that in the computation of the H79-widening $\mathcal{P} \mathbin{\nabla} \mathcal{Q}$ of two polyhedra $\mathcal{P}, \mathcal{Q} \in \mathbb{CP}_n$ it is required as a precondition that $\mathcal{P} \subseteq \mathcal{Q}$ (the same assumption was implicitly present in the cited papers).

The second widening operator, that we call *BHRZ03-widening*, is an instance of the specification provided in [BHRZ03a]. This operator also requires as a precondition that $\mathcal{P} \subseteq \mathcal{Q}$ 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 NNC polyhedra. 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.

Note:

As is the case for the other operators on polyhedra, the implementation overwrites one of the two polyhedra arguments with the result of the widening application. To avoid trivial misunderstandings, it is worth stressing that if polyhedra $\mathcal{P}$ and $\mathcal{Q}$ (where $\mathcal{P} \subseteq \mathcal{Q}$) are identified by program variables `p` and `q`, respectively, then the call `q.H79_widening_assign(p)` will assign the polyhedron $\mathcal{P} \mathbin{\nabla} \mathcal{Q}$ to variable `q`. Namely, it is the bigger polyhedron $\mathcal{Q}$ which is overwritten by the result of the widening. The smaller polyhedron is not modified, so as to lead to an easier coding of the usual convergence test ($\mathcal{P} \supseteq \mathcal{P} \mathbin{\nabla} \mathcal{Q}$ can be coded as `p.contains(q)`). Note that, in the above context, a call such as `p.H79_widening_assign(q)` is likely to result in undefined behavior, since the precondition $\mathcal{Q} \subseteq \mathcal{P}$ will be missed (unless it happens that $\mathcal{P} = \mathcal{Q}$). The same observation holds for all flavors of widenings and extrapolation operators that are implemented in the library and for all the foreign language interfaces.

1.4.17 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* [BHRZ03a]. 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.

1.4.18 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.

1.5 Bounded Difference Shapes

For each vector $\boldsymbol{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, and for each relation symbol $\bowtie \in \{=, \geq\}$, the linear constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b$ is said to be a *bounded difference* if there exist two indices $i, j \in \{0, \dots, n-1\}$ such that:

- $a_i, a_j \in \{-1, 0, 1\}$ and $a_i \neq a_j$;
- $a_k = 0$, for all $k \notin \{i, j\}$.

A convex polyhedron $\mathcal{P} \in \mathbb{CP}_n$ is said to be a *bounded difference shape* (BDS, for short) if and only if either $\mathcal{P}$ can be expressed as the intersection of a finite number of bounded difference constraints or $n = 0$ and $\mathcal{P} = \varnothing$.

By construction, a BDS is always topologically closed. Under the usual set inclusion ordering, the set of all BDSs on the vector space $\mathbb{R}^n$ is a lattice having the empty set $\varnothing$ and the universe $\mathbb{R}^n$ as the smallest and the biggest elements, respectively. In theoretical terms, it is a meet sub-lattice of $\mathbb{CP}_n$, meaning that the intersection of a finite set of BDSs is still a BDS; on the other hand, in general the poly-hull of two BDSs is not a BDS. The smallest BDS containing a finite set of BDSs is said to be their *bds-hull*.

The PPL provides support for computations on the domain of rational bounded difference shapes that, in selected contexts, can achieve a better precision/efficiency ratio with respect to the corresponding computations on a domain of convex polyhedra. As far as the representation of the rational inhomogeneous term of each bounded difference is concerned, several *rounding-aware* implementation choices are available, including:

- bounded precision integer types;
- bounded precision floating point types;
- unbounded precision integer and rational types, as provided by GMP.

The user interface for BDSs is meant to be as similar as possible to the one developed for the domain of closed polyhedra: in particular, all operators on polyhedra are also available for the domain of BDSs, even though they are typically characterized by a lower degree of precision.

1.5.1 Widening and Extrapolation Operators on BD Shapes

For the domain of BDSs, the library provides a variant of the widening operator for convex polyhedra defined in [CH78]. The implementation follows the specification in [BHMZ05], resulting in an operator which is well-defined on the domain of BDSs (i.e., it does not depend on the internal representation of BDSs), while still ensuring convergence in a finite number of steps.

The library also implements an extension of the widening operator for intervals as defined in [CC76]. The reader is warned that such an extension, even though being well-defined on the domain of BDSs, is not provided with a convergence guarantee and is therefore an extrapolation operator.

1.6 The Powerset Construction

The PPL provides the finite powerset construction; this takes a pre-existing domain and upgrades it to one that can represent disjunctive information (by using a *finite* number of disjuncts). The construction follows the approach described in [Bag98], also summarized in [BHZ04] where there is an account of generic widenings for the powerset domain (some of which are supported in the instantiation of this construction by the domain of convex polyhedra and described in Section The Polyhedra Powerset Domain).

1.6.1 The Powerset Domain

The domain is built from a pre-existing base-level domain D which must include an entailment relation '$\vdash$', meet operation '$\otimes$', a top element '**1**' and bottom element '**0**'.

A set $\mathcal{S} \in \wp(D)$ is called *non-redundant* with respect to '$\vdash$' if and only if $\mathbf{0} \notin \mathcal{S}$ and $\forall d_1, d_2 \in \mathcal{S} : d_1 \vdash d_2 \implies d_1 = d_2$. The set of finite non-redundant subsets of D (with respect to '$\vdash$') is denoted by $\wp_{\mathrm{fn}}^{\vdash}(D)$. The function $\Omega_D^{\vdash} \colon \wp_{\mathrm{f}}(D) \to \wp_{\mathrm{fn}}^{\vdash}(D)$, called *Omega-reduction*, maps a finite set into its non-redundant counterpart; it is defined, for each $\mathcal{S} \in \wp_{\mathrm{f}}(D)$, by

$$\Omega_D^{\vdash}(\mathcal{S}) \stackrel{\mathrm{def}}{=} \mathcal{S} \setminus \{\, d \in \mathcal{S} \mid d = \mathbf{0} \text{ or } \exists d' \in \mathcal{S} \,.\, d \Vdash d' \,\}.$$

where $d \Vdash d'$ denotes $d \vdash d' \wedge d \neq d'$.

As the intended semantics of a powerset domain element $\mathcal{S} \in \wp_{\mathrm{f}}(D)$ is that of disjunction of the semantics of D, the finite set $\mathcal{S}$ is semantically equivalent to the non-redundant set $\Omega_D^{\vdash}(\mathcal{S})$; and elements of $\mathcal{S}$ will be called *disjuncts*. The restriction to the finite subsets reflects the fact that here disjunctions are implemented by explicit collections of disjuncts. As a consequence of this restriction, for any $\mathcal{S} \in \wp_{\mathrm{f}}(D)$ such that $\mathcal{S} \neq \{\mathbf{0}\}$, $\Omega_D^{\vdash}(\mathcal{S})$ is the (finite) set of the maximal elements of $\mathcal{S}$.

The *finite powerset domain* over a domain D is the set of all finite non-redundant sets of D and denoted by D_{P}. The domain includes an approximation ordering '$\vdash_{\mathrm{P}}$' defined so that, for any $\mathcal{S}_1$ and $\mathcal{S}_2 \in D_{\mathrm{P}}$, $\mathcal{S}_1 \vdash_{\mathrm{P}} \mathcal{S}_2$ if and only if

$$\forall d_1 \in \mathcal{S}_1 : \exists d_2 \in \mathcal{S}_2 \,.\, d_1 \vdash d_2.$$

Therefore the top element is $\{\mathbf{1}\}$ and the bottom element is the emptyset.

Note:

As far as Omega-reduction is concerned, the library adopts a *lazy* approach: an element of the powerset domain is represented by a potentially redundant sequence of disjuncts. Redundancies can be eliminated by explicitly invoking the operator `omega_reduce()`, e.g., before performing the output of a powerset element. Note that all the documented operators automatically perform Omega-reductions on their arguments, when needed or appropriate.

1.7 Operations on the Powerset Construction

In this section we briefly describe the generic operations on Powerset Domains that are provided by the library for any given base-level domain D.

1.7.1 Meet and Upper Bound

Given the sets $\mathcal{S}_1$ and $\mathcal{S}_2 \in D_{\mathrm{P}}$, the *meet* and *upper bound* operators provided by the library returns the set $\Omega_D^{\vdash}\bigl(\{\, d_1 \otimes d_2 \mid d_1 \in \mathcal{S}_1, d_2 \in \mathcal{S}_2 \,\}\bigr)$ and Omega-reduced set union $\Omega_D^{\vdash}(\mathcal{S}_1 \cup \mathcal{S}_2)$ respectively.

1.7.2 Adding a Disjunct

Given the powerset element $\mathcal{S} \in D_{\mathrm{P}}$ and the base-level element $d \in D$, the *add disjunct* operator provided by the library returns the powerset element $\Omega_D^{\vdash}\bigl(\mathcal{S} \cup \{d\}\bigr)$.

1.7.3 Collapsing a Powerset Element

If the given powerset element is not empty, then the *collapse* operator returns the singleton powerset consisting of an upper-bound of all the disjuncts.

1.8 The Polyhedra Powerset Domain

The Polyhedra powerset domain $(\mathbb{P}_n)_{\mathrm{P}}$ provided by the PPL is the finite powerset domain (defined in Section The Powerset Construction) over the domain of NNC polyhedra $\mathbb{P}_n$.

In addition to the operations described for the generic powerset domain in Section Operations on the Powerset Construction, we provide some operations that are specific to this instantiation. Of these, most correspond to the application of the equivalent operation on each of the NNC polyhedra that are in the given set. Here we just describe those operations that are particular to the polyhedra powerset domain.

1.8.1 Geometric Comparisons

Given the sets $\mathcal{S}_1, \mathcal{S}_2 \in (\mathbb{P}_n)_{\mathrm{P}}$, then we say that $\mathcal{S}_1$ *geometrically covers* $\mathcal{S}_2$ if every point (in some element) in a polyhedron in $\mathcal{S}_2$ is also a point in a polyhedron in $\mathcal{S}_1$. If $\mathcal{S}_1$ geometrically covers $\mathcal{S}_2$ and $\mathcal{S}_2$ geometrically covers $\mathcal{S}_1$, then we say that they are *geometrically equal*.

1.8.2 Pairwise Merge

Given the powerset $\mathcal{S} \in (\mathbb{P}_n)_{\mathrm{P}}$, then the *pairwise merge* operator takes pairs of distinct elements in $\mathcal{S}$ whose poly-hull is the same as their set-theoretical union and replaces them by their union. This replacement is done recursively so that, for each pair $\mathcal{P}$, $\mathcal{Q}$ of distinct polyhedra in the result set, we have $\mathcal{P} \uplus \mathcal{Q} \neq \mathcal{P} \cup \mathcal{Q}$.

1.8.3 Extrapolation Operators

The library implements a generalization of the extrapolation operator for powerset domains proposed in [BGP99]. The operator `BGP99_extrapolation_assign` is made parametric by allowing for the specification of a base-level extrapolation operator different from the H79 widening (e.g., the BHRZ03 widening can be used). Note that, in the general case, this operator cannot guarantee the convergence of the iteration sequence in a finite number of steps (for a counter-example, see [BHZ04]).

1.8.4 Certificate-Based Widenings

The PPL library provides support for the specification of proper widening operators on the powerset domain of convex polyhedra. In particular, this version of the library implements an instance of the *certificate-based widening framework* proposed in [BHZ03b].

A *finite convergence certificate* for an extrapolation operator is a formal way of ensuring that such an operator is indeed a widening on the considered domain. Given a widening operator on the base-level domain, together with the corresponding convergence certificate, the BHZ03 framework shows how it is possible to lift this widening so as to work on the finite powerset domain, while still ensuring convergence in a finite number of iterations.

Being highly parametric, the BHZ03 widening framework can be instantiated in many ways. The current implementation provides the templatic operator `BHZ03_widening_assign<Certificate, Widening>` which only exploits a fraction of this generality, by allowing the user to specify the base-level widening function and the corresponding certificate. The widening strategy is fixed and uses two extrapolation heuristics: first, the least upper bound is tried; second, the BGP99 extrapolation operator is tried, possibly applying pairwise merging. If both heuristics fail to converge according to the convergence certificate, then an attempt is made to apply the base-level widening to the poly-hulls of the two arguments, possibly improving the result obtained by means of the poly-difference operator. For more details and a justification of the overall approach, see [BHZ03b] and [BHZ04].

The library provides two convergence certificates: while BHRZ03_Certificate is compatible with both the BHRZ03 and the H79 widenings, H79_Certificate is only compatible with the latter. Note that using different certificates will change the results obtained, even when using the same base-level widening operator. It is also worth stressing that it is up to the user to see that the widening operator is actually compatible with a given convergence certificate. If such a requirement is not met, then an extrapolation operator will be obtained.

1.9 Rational Grids

In this section we introduce rational grids as provided by the library. See also [BDHMZ05] for a detailed description of this domain.

The libary supports two representations for the grids domain; *congruence systems* and *grid generator systems*. We first describe *linear congruence relations* which form the elements of a congruence system.

1.9.1 Congruences and Congruence Relations

For any $a, b, f \in \mathbb{R}$, $a \equiv_f b$ denotes the *congruence* $\exists \mu \in \mathbb{Z} \mathrel{.} a - b = \mu f$.

Let $\mathbb{S} \in \{\mathbb{Q}, \mathbb{R}\}$. For each vector $\boldsymbol{a} \in \mathbb{S}^n \setminus \{\boldsymbol{0}\}$ and scalars $b, f \in \mathbb{S}$, the notation $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_f b$ stands for the *linear congruence relation in* $\mathbb{S}^n$ defined by the set of vectors

$$\bigl\{\, \boldsymbol{v} \in \mathbb{R}^n \bigm| \exists \mu \in \mathbb{Z} \mathrel{.} \langle \boldsymbol{a}, \boldsymbol{v} \rangle = b + \mu f \,\bigr\};$$

when $f \neq 0$, the relation is said to be *proper*; $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_0 b$ (i.e., when $f = 0$) denotes the equality $\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b$. f is called the *frequency* or *modulus* and b the *base value* of the relation. Thus, provided $\boldsymbol{a} \neq \boldsymbol{0}$, the relation $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_f b$ defines the set of affine hyperplanes

$$\bigl\{\, \bigl(\langle \boldsymbol{a}, \boldsymbol{x} \rangle = b + \mu f\bigr) \bigm| \mu \in \mathbb{Z} \,\bigr\};$$

if $b \equiv_f 0$, $\langle \boldsymbol{0}, \boldsymbol{x} \rangle \equiv_f b$ defines the universe $\mathbb{R}^n$ and the empty set, otherwise.

1.9.2 Rational Grids

The set $\mathcal{L} \subseteq \mathbb{R}^n$ is a *rational grid* if and only if either $\mathcal{L}$ is the set of vectors in $\mathbb{R}^n$ that satisfy a finite system $\mathcal{C}$ of congruence relations in $\mathbb{Q}^n$ or $n = 0$ and $\mathcal{L} = \emptyset$.

We also say that $\mathcal{L}$ *is described by* $\mathcal{C}$ and that $\mathcal{C}$ *is a congruence system for* $\mathcal{L}$.

The *grid domain* $\mathbb{G}_n$ is the set of all rational grids described by finite sets of congruence relations in $\mathbb{Q}^n$.

If the congruence system $\mathcal{C}$ describes the $\emptyset$, the *empty* grid, then we say that $\mathcal{C}$ is *inconsistent*. For example, the congruence systems $\bigl\{\langle \boldsymbol{0}, \boldsymbol{x} \rangle \equiv_0 1\bigr\}$ meaning that $0 = 1$ and $\bigl\{\langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_2 0, \langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_2 1\bigr\}$, for any

$\boldsymbol{a} \in \mathbb{R}^n$, meaning that the value of an expression must be both even and odd are both inconsistent since both describe the empty grid.

When ordering grids by the set inclusion relation, the empty set $\varnothing$ and the vector space $\mathbb{R}^n$ (which is described by the empty set of congruence relations) are, respectively, the smallest and the biggest elements of $\mathbb{G}_n$. The vector space $\mathbb{R}^n$ is also called the *universe* grid.

In set theoretical terms, $\mathbb{G}_n$ is a *lattice* under set inclusion.

1.9.3 Integer Combinations

Let $S = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_k\} \subseteq \mathbb{R}^n$ be a finite set of vectors. For all scalars $\mu_1, \dots, \mu_k \in \mathbb{Z}$, the vector $\boldsymbol{v} = \sum_{j=1}^{k} \mu_j \boldsymbol{x}_j$ is said to be a *integer* combination of the vectors in S.

We denote by $\mathrm{int.hull}(S)$ (resp., $\mathrm{int.affine.hull}(S)$) the set of all the integer (resp., integer and affine) combinations of the vectors in S.

1.9.4 Points, Parameters and Lines

Let $\mathcal{L}$ be a grid. Then

- a vector $\boldsymbol{p} \in \mathcal{L}$ is called a *point* of $\mathcal{L}$;
- a vector $\boldsymbol{q} \in \mathbb{R}^n$, where $\boldsymbol{q} \neq \boldsymbol{0}$, is called a *parameter* of $\mathcal{L}$ if $\mathcal{L} \neq \varnothing$ and $\boldsymbol{p} + \mu\boldsymbol{q} \in \mathcal{L}$, for all points $\boldsymbol{p} \in \mathcal{L}$ and all $\mu \in \mathbb{Z}$;
- a vector $\boldsymbol{l} \in \mathbb{R}^n$ is called a *line* of $\mathcal{L}$ if $\mathcal{L} \neq \varnothing$ and $\boldsymbol{p} + \lambda\boldsymbol{l} \in \mathcal{L}$, for all points $\boldsymbol{p} \in \mathcal{L}$ and all $\lambda \in \mathbb{R}$.

1.9.5 The Grid Generator Representation

We can generate any rational grid in $\mathbb{G}_n$ from a finite subset of its points, parameters and lines; each point in a grid is obtained by adding a linear combination of its generating lines to an integral combination of its parameters and an integral affine combination of its generating points.

If L, Q, P are each finite subsets of $\mathbb{Q}^n$ and

$$\mathcal{L} = \mathrm{linear.hull}(L) + \mathrm{int.hull}(Q) + \mathrm{int.affine.hull}(P)$$

where the symbol '$+$' denotes the Minkowski's sum, then $\mathcal{L} \in \mathbb{G}_n$ is a rational grid (see Section 4.4 in [Sch99] and also Proposition 8 in [BDHMZ05]). The 3-tuple (L, Q, P) is said to be a *generator system* for $\mathcal{L}$ and we write $\mathcal{L} = \mathrm{ggen}(L, Q, P)$.

Note that the grid $\mathcal{L} = \mathrm{ggen}(L, Q, P) = \varnothing$ if and only if the set of points $P = \varnothing$. If $P \neq \varnothing$, then $\mathcal{L} = \mathrm{ggen}(L, \varnothing, Q_{\boldsymbol{p}} \cup P)$ where, for some $\boldsymbol{p} \in P$, $Q_{\boldsymbol{p}} = \{\, \boldsymbol{p} + \boldsymbol{q} \mid \boldsymbol{q} \in Q \,\}$.

1.9.6 Minimized Grid Representations

A *minimized* congruence system $\mathcal{C}$ for $\mathcal{L}$ is such that, if $\mathcal{C}'$ is another congruence system for $\mathcal{L}$, then $\#\,\mathcal{C} \leq \#\,\mathcal{C}'$. Note that a minimized congruence system for a non-empty grid has at most n congruence relations.

Similarly, a *minimized* generator system $\mathcal{G} = (L, Q, P)$ for $\mathcal{L}$ is such that, if $\mathcal{G}' = (L', Q', P')$ is another generator system for $\mathcal{L}$, then $\#\,L \leq \#\,L'$ and $\#\,Q + \#\,P \leq \#\,Q' + \#\,P'$. Note that a minimized generator system for a grid has no more than a total of $n + 1$ lines, parameters and points.

1.9.7 Double Description for Grids

As for convex polyhedra, any grid $\mathcal{L}$ can be described by using a congruence system $\mathcal{C}$ for $\mathcal{L}$, a generator system $\mathcal{G}$ for $\mathcal{L}$, or both by means of the *double description pair (DD pair)* $(\mathcal{C}, \mathcal{G})$. The *double description method* for grids is a collection of theoretical results very similar to those for convex polyhedra showing that, given one kind of representation, there are algorithms for computing a representation of the other kind and for minimizing both representations.

As for convex polyhedra, such changes of representation form a key step in the implementation of many operators on grids such as, for example, intersection and grid join.

1.9.8 Space Dimensions and Dimension-compatibility for Grids

The *space dimension* of a grid $\mathcal{L} \in \mathbb{G}_n$ is the dimension $n \in \mathbb{N}$ of the corresponding vector space $\mathbb{R}^n$. The space dimension of congruence relations, generators and other objects of the library is defined similarly.

1.9.9 Affine Independence and Affine Dimension for Grids

A *non-empty* grid $\mathcal{L} \in \mathbb{G}_n$ has *affine dimension* $k \in \mathbb{N}$, denoted by $\dim(\mathcal{G}) = k$, if the maximum number of affinely independent points in $\mathcal{G}$ is $k + 1$. The affine dimension of an empty grid is defined to be 0. Thus we have $0 \le \dim(\mathcal{G}) \le n$.

1.10 Operations on Rational Grids

In this section we briefly describe operations on rational grids that are provided by the library. These are similar to those described in Section Operations on Convex Polyhedra.

1.10.1 Grid Intersection and Grid Join

For any pair of grids $\mathcal{L}_1, \mathcal{L}_2 \in \mathbb{G}_n$, the *intersection* of $\mathcal{L}_1$ and $\mathcal{L}_2$, defined as the set intersection $\mathcal{L}_1 \cap \mathcal{L}_2$, is the largest grid included in both $\mathcal{L}_1$ and $\mathcal{L}_2$; similarly, the *grid join* of $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \uplus \mathcal{L}_2$, is the smallest grid that includes both $\mathcal{L}_1$ and $\mathcal{L}_2$.

In theoretical terms, the intersection and grid join operators defined above are the binary *meet* and the binary *join* operators on the lattice $\mathbb{G}_n$.

1.10.2 Grid Difference

For any pair of grids $\mathcal{L}_1, \mathcal{L}_2 \in \mathbb{G}_n$, the *grid difference* of $\mathcal{L}_1$ and $\mathcal{L}_2$ is defined as the smallest grid containing the set-theoretic difference of $\mathcal{L}_1$ and $\mathcal{L}_2$.

1.10.3 Concatenating Grids

Viewing a grid as a set of tuples (its points), it is sometimes useful to consider the set of tuples obtained by concatenating an ordered pair of grids. Formally, the *concatenation* of the grids $\mathcal{L}_1 \in \mathbb{G}_n$ and $\mathcal{L}_2 \in \mathbb{G}_m$ (taken in this order) is the grid in $\mathbb{G}_{n+m}$ defined as

$$\Bigl\{ (x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})^{\mathrm{T}} \in \mathbb{R}^{n+m} \Bigm| (x_0, \ldots, x_{n-1})^{\mathrm{T}} \in \mathcal{L}_1, (y_0, \ldots, y_{m-1})^{\mathrm{T}} \in \mathcal{L}_2 \Bigr\}.$$

Another way of seeing it is as follows: first embed grid $\mathcal{L}_1$ into a vector space of dimension $n + m$ and then add a suitably renamed-apart version of the congruence relations defining $\mathcal{L}_2$.

1.10.4 Adding New Dimensions to the Vector Space

The library provides two operators for adding a number i of space dimensions to a grid $\mathcal{L} \in \mathbb{G}_n$, therefore transforming it into a new grid in $\mathbb{G}_{n+i}$. In both cases, the added dimensions of the vector space are those having the highest indices.

The operator add_space_dimensions_and_embed *embeds* the grid $\mathcal{L}$ into the new vector space of dimension $i + n$ and returns the grid defined by all and only the congruences defining $\mathcal{L}$ (the variables corresponding to the added dimensions are unconstrained). For instance, when starting from a grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

$$\left\{\, (x_0, x_1, x_2)^{\mathrm{T}} \in \mathbb{R}^3 \;\middle|\; (x_0, x_1)^{\mathrm{T}} \in \mathcal{L} \,\right\}.$$

In contrast, the operator add_space_dimensions_and_project *projects* the grid $\mathcal{L}$ into the new vector space of dimension $i + n$ and returns the grid whose congruence system, besides the congruence relations defining $\mathcal{L}$, will include additional equalities on the added dimensions. Namely, the corresponding variables are all constrained to be equal to 0. For instance, when starting from a grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

$$\left\{\, (x_0, x_1, 0)^{\mathrm{T}} \in \mathbb{R}^3 \;\middle|\; (x_0, x_1)^{\mathrm{T}} \in \mathcal{L} \,\right\}.$$

1.10.5 Removing Dimensions from the Vector Space

The library provides two operators for removing space dimensions from a grid $\mathcal{L} \in \mathbb{G}_n$, therefore transforming it into a new grid in $\mathbb{G}_m$ where $m \leq n$.

Given a set of variables, the operator remove_space_dimensions removes all the space dimensions specified by the variables in the set.

Given a space dimension m less than or equal to that of the grid, the operator remove_higher_space_dimensions removes the space dimensions having indices greater than or equal to m.

1.10.6 Mapping the Dimensions of the Vector Space

The operator map_space_dimensions provided by the library maps the dimensions of the vector space $\mathbb{R}^n$ according to a partial injective function $\rho \colon \{0, \ldots, n-1\} \rightarrowtail \mathbb{N}$ such that

$$\rho\bigl(\{0, \ldots, n-1\}\bigr) = \{0, \ldots, m-1\}$$

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 grid $\mathcal{L} \in \mathbb{G}_n$ onto the zero-dimension space $\mathbb{R}^0$; otherwise the result is a grid in $\mathbb{G}_m$ given by

$$\left\{\, \bigl(v_{\rho^{-1}(0)}, \ldots, v_{\rho^{-1}(m-1)}\bigr)^{\mathrm{T}} \;\middle|\; (v_0, \ldots, v_{n-1})^{\mathrm{T}} \in \mathcal{L} \,\right\}.$$

1.10.7 Expanding One Dimension of the Vector Space to Multiple Dimensions

The operator expand_space_dimension provided by the library adds m new space dimensions to a grid $\mathcal{L} \in \mathbb{G}_n$, with $n > 0$, so that dimensions n, $n+1$, . . ., $n+m-1$ of the resulting grid are exact copies of the i-th space dimension of $\mathcal{L}$. More formally, the result is a grid in $\mathbb{G}_m$ given by

$$\left\{ \boldsymbol{u} \in \mathbb{R}^{n+m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v}, \boldsymbol{w} \in \mathcal{L} \,.\, u_i = v_i \\ \qquad \wedge \forall j = n, n+1, \ldots, n+m-1 : u_j = w_i \\ \qquad \wedge \forall k = 0, \ldots, n-1 : k \neq i \implies u_k = v_k = w_k \end{array} \right\}.$$

1.10.8 Folding Multiple Dimensions of the Vector Space into One Dimension

The operator `fold_space_dimensions` provided by the library, given a grid $\mathcal{L} \in \mathbb{G}_n$, with $n > 0$, folds a subset J of the set of space dimensions $\{0, \ldots, n-1\}$ into a space dimension $i < n$, where $i \notin J$. Letting $m = \# J$, the result is given by the grid join

$$\mathcal{L}_0 \uplus \cdots \uplus \mathcal{L}_m$$

where

$$\mathcal{L}_m \stackrel{\text{def}}{=} \left\{ \boldsymbol{u} \in \mathbb{R}^{n-m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v} \in \mathcal{L} \,.\, u_{i'} = v_i \\ \qquad \wedge \forall k = 0, \ldots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\},$$

for $d = 0, \ldots, m-1$,

$$\mathcal{L}_d \stackrel{\text{def}}{=} \left\{ \boldsymbol{u} \in \mathbb{R}^{n-m} \;\middle|\; \begin{array}{l} \exists \boldsymbol{v} \in \mathcal{L} \,.\, u_{i'} = v_{j_d} \\ \qquad \wedge \forall k = 0, \ldots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\}$$

and, for $k = 0, \ldots, n-1$,

$$k' \stackrel{\text{def}}{=} k - \#\{\, j \in J \mid k > j \,\}.$$

1.10.9 Affine Images and Preimages

As for convex polyhedra (see Single-Update Affine Functions), the library provides affine image and preimage operators for grids: given a variable x_k and linear expression $\mathrm{expr} = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$, these determine the affine transformation $\phi = \bigl(x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b\bigr) \colon \mathbb{R}^n \rightarrow \mathbb{R}^n$ that transforms any point $(v_0, \ldots, v_{n-1})^{\mathrm{T}}$ in a grid $\mathcal{L}$ to

$$\Bigl(v_0, \ldots, \bigl(\textstyle\sum_{i=0}^{n-1} a_i v_i + b\bigr), \ldots, v_{n-1}\Bigr)^{\mathrm{T}}.$$

The *affine image* operator computes the affine image of a grid $\mathcal{L}$ under $x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$. For instance, suppose the grid $\mathcal{L}$ to be transformed is the non-relational grid in $\mathbb{R}^2$ generated by the set of points $\bigl\{(0,0)^{\mathrm{T}}, (0,3)^{\mathrm{T}}, (3,0)^{\mathrm{T}}\bigr\}$. Then, if the considered variable is x_0 and the linear expression is $3x_0 + 2x_1 + 1$ (so that $k = 0$, $a_0 = 3, a_1 = 2, b = 1$), the affine image operator will translate $\mathcal{L}$ to the grid $\mathcal{L}_1$ generated by the set of points $\bigl\{(1,0)^{\mathrm{T}}, (7,3)^{\mathrm{T}}, (10,0)^{\mathrm{T}}\bigr\}$ which is the grid generated by the point $(1,0)$ and parameters $(3,-3), (0,9)$; or, alternatively defined by the congruence system $\{x \equiv_3 1, x + y \equiv_9 1\}$. 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 grid $\mathcal{L}_2$ is the grid containing all the points whose coordinates are integral multiples of 3 and lie on line $x = y$.

The affine preimage operator computes the affine preimage of a grid $\mathcal{L}$ 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 $3x_0 + 2x_1 + 1$ to the grid $\mathcal{L}_1$; then we get the original grid $\mathcal{L}$ 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 $\mathcal{L}_2$, then the resulting grid will consist of all the points in $\mathbb{R}^2$ where the y coordinate is an integral multiple of 3.

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

1.10.10 Generalized Affine Images

Similarly to convex polyhedra (see Generalized Affine Relations), the library provides two other grid operators that are generalizations of the single update affine image and preimage operators for grids. The *generalized affine image* operator $\phi = (\mathrm{lhs}', \mathrm{rhs}, f) \colon \mathbb{R}^n \rightarrow \mathbb{R}^n$, where $\mathrm{lhs} = \langle \boldsymbol{c}, \boldsymbol{x} \rangle + d$ and $\mathrm{rhs} = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$ are affine expressions and $f \in \mathbb{Q}$, is defined as

$$\forall \boldsymbol{v} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^n : (\boldsymbol{v}, \boldsymbol{w}) \in \phi \iff \bigl(\langle \boldsymbol{c}, \boldsymbol{w} \rangle + d \equiv_f \langle \boldsymbol{a}, \boldsymbol{v} \rangle + b\bigr) \land \Bigl(\bigwedge_{0 \leq i < n, c_i = 0} w_i = v_i \Bigr).$$

Note that, when $\mathrm{lhs} = x_k$ and $f = 0$, so that the transfer function is an equality, then the above operator is equivalent to the application of the standard affine image of $\mathcal{L}$ with respect to the variable x_k and the affine expression rhs.

1.10.11 Time-Elapse Operator

For any two grids $\mathcal{L}_1, \mathcal{L}_2 \in \mathbb{G}_n$, the *time-elapse* between $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted $\mathcal{L}_1 \nearrow \mathcal{L}_2$, is the grid

$$\bigl\{ \boldsymbol{p} + \mu \boldsymbol{q} \in \mathbb{R}^n \bigm| \boldsymbol{p} \in \mathcal{L}_1, \boldsymbol{q} \in \mathcal{L}_2, \mu \in \mathbb{Z} \bigr\}.$$

1.10.12 Relation-with Operators

The library provides operators for checking the relation holding between a grid and either a congruence or a generator.

Suppose $\mathcal{L}$ is a grid and $\mathcal{C}$ an arbitrary congruence system representing $\mathcal{L}$. Suppose also that $\mathrm{cg} = \bigl(\langle \boldsymbol{a}, \boldsymbol{x} \rangle \equiv_f b\bigr)$ is a congruence relation with $\mathcal{L}_{\mathrm{cg}} = \mathrm{gcon}\bigl(\{\mathrm{cg}\}\bigr)$. The possible relations between $\mathcal{L}$ and cg are as follows.

- $\mathcal{L}$ *is disjoint* from cg if $\mathcal{L} \cap \mathcal{L}_{\mathrm{cg}} = \emptyset$; that is, adding cg to $\mathcal{C}$ gives us the empty grid.
- $\mathcal{L}$ *strictly intersects* cg if $\mathcal{L} \cap \mathcal{L}_{\mathrm{cg}} \neq \emptyset$ and $\mathcal{L} \cap \mathcal{L}_{\mathrm{cg}} \subset \mathcal{L}$; that is, adding cg to $\mathcal{C}$ gives us a non-empty grid strictly smaller than $\mathcal{L}$.
- $\mathcal{L}$ *is included* in cg if $\mathcal{L} \subseteq \mathcal{L}_{\mathrm{cg}}$; that is, adding cg to $\mathcal{C}$ leaves $\mathcal{L}$ unchanged.

A grid $\mathcal{L}$ *subsumes* a generator g if adding g to any generator system representing $\mathcal{L}$ does not change $\mathcal{L}$.

1.10.13 Rectilinear Grids and Covering Boxes

Consider again the boxes used for convex polyhedra. An n-dimensional *rational box* is a box $\mathcal{B}$ in $\mathbb{Q}^n$; that is, where the bounded bounds in the intervals are each in $\mathbb{Q}$. We will say that an interval is *bounded* if both its bounds are bounded.

Each bounded interval in $\mathcal{B}$ determines a congruence cg in $\mathbb{Q}^n$. Letting $\boldsymbol{e}_i = (0, \ldots, 1, \ldots, 0)^{\mathrm{T}}$ be the vector in $\mathbb{R}^n$ with 1 in the i'th position and zeroes in every other position; if both the bounds of the interval are closed, then the congruence cg is defined as $\bigl(\langle \boldsymbol{e}_i, \boldsymbol{x} \rangle \equiv_f b\bigr)$, where b is the value of the lower bound and f is the (non-negative) difference between the lower and upper bounds. If one of the bounds is open, then cg is the congruence $\bigl(\langle \boldsymbol{0}, \boldsymbol{x} \rangle \equiv_0 1\bigr)$ representing the inconsistent equality $(0 = 1)$.

Let $\mathcal{C}$ be the set of congruences defined by the bounded intervals in a rational box $\mathcal{B}$; then we say that B *represents* the rational grid $\mathcal{L} = \mathrm{gcon}(\mathcal{C})$. Any grid $\mathcal{L} \in \mathbb{G}_n$ that can be represented by a box is said to be *rectilinear*.

A *covering box* of a grid $\mathcal{L}$ is a rational box representing the smallest rectilinear grid that contains $\mathcal{L}$.

As for convex polyhedra, the library will provide operations for computing the rectilinear grid corresponding to a given box and, also, a covering box for any given grid.

1.10.14 Widening Operators

The library provides a *grid widening* operator for the domain of grids. This follows the specification provided in [BDHMZ05]. Note that, as for the widenings provided for convex polyhedra, the widening $\mathcal{L}_1 \nabla \mathcal{L}_2$ of two grids $\mathcal{L}_1, \mathcal{L}_2 \in \mathbb{G}_n$ requires as a precondition that $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

Note:

As is the case for the other operators on grids, the implementation overwrites one of the two grid arguments with the result of the widening application. It is worth stressing that, if grid $\mathcal{L}_1$ and $\mathcal{L}_2$ (where $\mathcal{L}_1 \subseteq \mathcal{L}_2$) are identified by program variables `l_1` and `l_2`, respectively, then the call `l_2.grid_widening_assign(l_1)` will assign the grid $\mathcal{L}_1 \nabla \mathcal{L}_2$ to variable `l_2`. Namely, it is the bigger grid$\mathcal{L}_2$ which is overwritten by the result of the widening. The smaller grid is not modified. The same observation holds for all flavors of widenings and extrapolation operators that are implemented in the library and for all the foreign language interfaces.

1.10.15 Widening with Tokens

This will be as for widening with tokens for convex polyhedra.

1.10.16 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 grid widening that is provided, 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 congruence system as an additional parameter and uses it to improve the approximation yielded by the corresponding widening operator. Note that, as in the case for convex polyhedra, a convergence guarantee can only be obtained by suitably restricting the set of congruence relations that can occur in this additional parameter.

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 rectilinear grid that contains the two argument grids.

1.11 Using the Library

1.11.1 A Note on the Implementation of the Operators

When adopting the double description method for the representation of convex polyhedra, the implementation of most of the operators 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 knowledgeable 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.

One of the cases 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 emptiness check. Note anyway that the same effect can be obtained by interleaving the calls of the lazy operator with explicit emptiness checks.

1.11.2 On Object-Orientation and Polymorphism: A Disclaimer

The PPL library is mainly a collection of so-called “concrete data types”: while providing the user with a clean and friendly interface, these types are not meant to — i.e., they should not — be used polymorphically (since, e.g., most of the destructors are not declared `virtual`). In practice, this restriction means that the library types should not be used as *public base classes* to be derived from. A user willing to extend the library types, adding new functionalities, often can do so by using *containment* instead of inheritance; even when there is the need to override a `protected` method, non-public inheritance should suffice.

1.11.3 On Const-Correctness: A Warning about the Use of References and Iterators

Most operators of the library depend on one or more parameters that are declared “const”, meaning that they will not be changed by the application of the considered operator. Due to the adoption of lazy computation techniques, in many cases such a const-correctness guarantee only holds at the semantic level, whereas it does not necessarily hold at the implementation level. For a typical example, consider the extraction from a polyhedron of its constraint system representation. While this operation is not going to change the polyhedron, it might actually invoke the internal conversion algorithm and modify the generators representation of the polyhedron object, e.g., by reordering the generators and removing those that are detected as redundant. Thus, any previously computed reference to the generators of the polyhedron (be it a direct reference object or an indirect one, such as an iterator) will no longer be valid. For this reason, code fragments such as the following should be avoided, as they may result in undefined behavior:

```
// Find a reference to the first point of the non-empty polyhedron ‘ph’.
const Generator_System& gs = ph.generators();
Generator_System::const_iterator i = gs.begin();
for (Generator_System::const_iterator gs_end = gs.end(); i != gs_end; ++i)
  if (i->is_point())
    break;
const Generator& p = *i;
// Get the constraints of ‘ph’.
const Constraint_System& cs = ph.constraints();
// Both the const iterator ‘i’ and the reference ‘p’
// are no longer valid at this point.
cout << p.divisor() << endl;  // Undefined behavior!
++i;                          // Undefined behavior!
```

As a rule of thumb, if a polyhedron plays any role in a computation (even as a const parameter), then any previously computed reference to parts of the polyhedron may have been invalidated. Note that, in the example above, the computation of the constraint system could have been placed after the uses of the iterator i and the reference p. Anyway, if really needed, it is always possible to take a copy of, instead of a reference to, the parts of interest of the polyhedron; in the case above, one may have taken a copy of the generator system by replacing the second line of code with the following:

```
Generator_System gs = ph.generators();
```

The same observations, modulo syntactic sugar, apply to the operators defined in the C interface of the library.

1.12 Bibliography

[Bag98] R. Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. *Science of Computer Programming*, 30(1-2):119-155, 1998.

[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.

[BDHMZ05] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. A Linear Domain for Analyzing the Distribution of Numerical Values. Technical report 2005.06, School of Computing, University of Leeds, UK, 2005. Available at http://www.comp.leeds.ac.uk/research/pubs/reports.shtml.

[BHMZ05] R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening Operators for Weakly-Relational Numeric Abstractions. In C. Hankin and I. Silveroni, editors, *Static Analysis: Proceedings of the 12th International Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 3-18, London, UK, 2005. Springer-Verlag, Berlin.

[BHRZ03a] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 337-354, San Diego, California, USA, 2003. Springer-Verlag, Berlin.

[BHRZ03b] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. Quaderno 312, 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. Available at http://www.cs.unipr.it/Publications/.

[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.

[BHZ03a] R. Bagnara, P. M. Hill, and E. Zaffanella. A new encoding and implementation of not necessarily closed convex polyhedra. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *Proceedings of the 3rd Workshop on Automated Verification of Critical Systems*, pages 161-176, Southampton, UK, 2003. Published as TR Number DSSE-TR-2003-2, University of Southampton.

[BHZ03b] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In B. Steffen and G. Levi, editors, *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 135-148, Venice, Italy, 2003. Springer-Verlag, Berlin.

[BHZ04] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. Quaderno 349, Dipartimento di Matematica, Università di Parma, Italy, 2004. Available at http://www.cs.unipr.it/Publications/.

[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].

[CC76] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *Proceedings of the 2nd International Symposium on Programming*, pages 106-130, Paris, France, 1976.

[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.

[GDD$^+$ 04] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 512-529. Springer-Verlag, Berlin, 2004.

[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, Medford, MA, USA, 2001. ACM.

[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] 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.

[Mas92] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 226-235, Washington, DC, USA, 1992. ACM Press.

[Mas93] F. Masdupuy. *Array Indices Relational Semantic Analysis Using Rational Cosets and Trapezoids*. Thèse d'informatique, École Polytechnique, Palaiseau, France, December 1993.

[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.

[NR00] S. P. K. Nookala and T. Risset. A library for Z-polyhedral operations. *Publication interne* 1330, IRISA, Campus de Beaulieu, Rennes, France, 2000.

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

[Sch99] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1999.

[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 Class Index

5.1 PPL Class List

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

6 PPL Page Index

6.1 PPL Related Pages

Here is a list of all related documentation pages:

7 PPL Module Documentation

7.1 C++ Language Interface

The core implementation of the Parma Polyhedra Library is written in C++.

Namespaces

- namespace Parma_Polyhedra_Library

  The entire library is confined to this namespace.

- namespace Parma_Polyhedra_Library::IO_Operators

  All input/output operators are confined to this namespace.

- namespace std

  The standard C++ namespace.

Classes

- class Parma_Polyhedra_Library::Checked_Number< T, Policy >

A wrapper for numeric types implementing a given policy.

- class Parma_Polyhedra_Library::Throwable

  User objects the PPL can throw.

- struct Parma_Polyhedra_Library::From_Bounding_Box

  A tag class.

- struct Parma_Polyhedra_Library::From_Covering_Box

  A tag class.

- class Parma_Polyhedra_Library::Constraint_System

  A system of constraints.

- class Parma_Polyhedra_Library::Constraint_System::const_iterator

  An iterator over a system of constraints.

- class Parma_Polyhedra_Library::Variable

  A dimension of the vector space.

- struct Parma_Polyhedra_Library::Variable::Compare

  Binary predicate defining the total ordering on variables.

- class Parma_Polyhedra_Library::Poly_Con_Relation

  The relation between a polyhedron and a constraint.

- class Parma_Polyhedra_Library::Generator_System

  A system of generators.

- class Parma_Polyhedra_Library::Generator_System::const_iterator

  An iterator over a system of generators.

- class Parma_Polyhedra_Library::Grid_Generator_System

  A system of grid generators.

- class Parma_Polyhedra_Library::Grid_Generator_System::const_iterator

  An iterator over a system of grid generators.

- class Parma_Polyhedra_Library::Congruence_System

  A system of congruences.

- class Parma_Polyhedra_Library::Congruence_System::const_iterator

  An iterator over a system of congruences.

- class Parma_Polyhedra_Library::Linear_Expression

  A linear expression.

- class Parma_Polyhedra_Library::Constraint

  A linear equality or inequality.

- class Parma_Polyhedra_Library::Generator

  A line, ray, point or closure point.

- class Parma_Polyhedra_Library::Grid_Generator

  A line, parameter or point.

- class Parma_Polyhedra_Library::Congruence

  A linear congruence.

- class Parma_Polyhedra_Library::LP_Problem

  A Linear Programming problem.

- class Parma_Polyhedra_Library::Poly_Gen_Relation

  The relation between a polyhedron and a generator.

- class Parma_Polyhedra_Library::BHRZ03_Certificate

  The convergence certificate for the BHRZ03 widening operator.

- struct Parma_Polyhedra_Library::BHRZ03_Certificate::Compare

  A total ordering on BHRZ03 certificates.

- class Parma_Polyhedra_Library::H79_Certificate

  A convergence certificate for the H79 widening operator.

- struct Parma_Polyhedra_Library::H79_Certificate::Compare

  A total ordering on H79 certificates.

- class Parma_Polyhedra_Library::Polyhedron

  The base class for convex polyhedra.

- class Parma_Polyhedra_Library::Grid_Certificate

  The convergence certificate for the Grid widening operator.

- class Parma_Polyhedra_Library::Grid

  A grid.

- class Parma_Polyhedra_Library::C_Polyhedron

  A closed convex polyhedron.

- class Parma_Polyhedra_Library::NNC_Polyhedron

  A not necessarily closed convex polyhedron.

- class Parma_Polyhedra_Library::BD_Shape< T >

  A bounded difference shape.

- class Parma_Polyhedra_Library::Determinate< PH >

  Wraps a PPL class into a determinate constraint system interface.

- class Parma_Polyhedra_Library::Powerset< D >

  The powerset construction on a base-level domain.

- class Parma_Polyhedra_Library::Polyhedra_Powerset< PH >

    The powerset construction instantiated on PPL polyhedra.

- class Parma_Polyhedra_Library::GMP_Integer

    Unbounded integers as provided by the GMP library.

Defines

- #define PPL_VERSION_MAJOR 0

    The major number of the PPL version.

- #define PPL_VERSION_MINOR 9

    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.

- #define PPL_VERSION "0.9"

    A string containing the PPL version.

Typedefs

- typedef size_t Parma_Polyhedra_Library::dimension_type

    An unsigned integral type for representing space dimensions.

- typedef size_t Parma_Polyhedra_Library::memory_size_type

    An unsigned integral type for representing memory size in bytes.

- typedef COEFFICIENT_TYPE Parma_Polyhedra_Library::Coefficient

    An alias for easily naming the type of PPL coefficients.

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

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

Enumerations

- enum Parma_Polyhedra_Library::Result { ,

    Parma_Polyhedra_Library::VC_NORMAL, Parma_Polyhedra_Library::V_LT, Parma_Polyhedra_-Library::V_GT, Parma_Polyhedra_Library::V_EQ,

    Parma_Polyhedra_Library::V_NE, Parma_Polyhedra_Library::V_LE, Parma_Polyhedra_-Library::V_GE, Parma_Polyhedra_Library::V_LGE, Parma_Polyhedra_Library::VC_MINUS_INFINITY, Parma_Polyhedra_Library::V_NEG_-OVERFLOW, Parma_Polyhedra_Library::VC_PLUS_INFINITY, Parma_Polyhedra_Library::V_-POS_OVERFLOW,

Parma_Polyhedra_Library::VC_NAN, Parma_Polyhedra_Library::V_CVT_STR_UNK, Parma_-Polyhedra_Library::V_DIV_ZERO, Parma_Polyhedra_Library::V_INF_ADD_INF,

Parma_Polyhedra_Library::V_INF_DIV_INF, Parma_Polyhedra_Library::V_INF_MOD, Parma_-Polyhedra_Library::V_INF_MUL_ZERO, Parma_Polyhedra_Library::V_INF_SUB_INF,

Parma_Polyhedra_Library::V_MOD_ZERO, Parma_Polyhedra_Library::V_SQRT_NEG, Parma_-Polyhedra_Library::V_UNKNOWN_NEG_OVERFLOW, Parma_Polyhedra_Library::V_-UNKNOWN_POS_OVERFLOW,

Parma_Polyhedra_Library::V_UNORD_COMP }

Possible outcomes of a checked arithmetic computation.

- enum Parma_Polyhedra_Library::Rounding_Dir { Parma_Polyhedra_Library::ROUND_DOWN, Parma_Polyhedra_Library::ROUND_UP, Parma_Polyhedra_Library::ROUND_IGNORE , Parma_-Polyhedra_Library::ROUND_NOT_NEEDED }

  Rounding directions for arithmetic computations.

- enum Parma_Polyhedra_Library::Degenerate_Element { Parma_Polyhedra_Library::UNIVERSE, Parma_Polyhedra_Library::EMPTY }

  Kinds of degenerate abstract elements.

- enum Parma_Polyhedra_Library::Relation_Symbol {

  Parma_Polyhedra_Library::LESS_THAN, Parma_Polyhedra_Library::LESS_THAN_OR_EQUAL, Parma_Polyhedra_Library::EQUAL, Parma_Polyhedra_Library::GREATER_THAN_OR_EQUAL,

  Parma_Polyhedra_Library::GREATER_THAN }

  Relation symbols.

- enum Parma_Polyhedra_Library::Complexity_Class { Parma_Polyhedra_-Library::POLYNOMIAL_COMPLEXITY, Parma_Polyhedra_Library::SIMPLEX_-COMPLEXITY, Parma_Polyhedra_Library::ANY_COMPLEXITY }

  Complexity pseudo-classes.

- enum Parma_Polyhedra_Library::Optimization_Mode { Parma_Polyhedra_-Library::MINIMIZATION, Parma_Polyhedra_Library::MAXIMIZATION }

  Possible optimization modes.

- enum Parma_Polyhedra_Library::LP_Problem_Status { Parma_Polyhedra_-Library::UNFEASIBLE_LP_PROBLEM, Parma_Polyhedra_Library::UNBOUNDED_LP_-PROBLEM, Parma_Polyhedra_Library::OPTIMIZED_LP_PROBLEM }

  Possible outcomes of the LP_Problem solver.

Variables

- const Throwable ∗volatile Parma_Polyhedra_Library::abandon_expensive_computations

  A pointer to an exception object.

7.1.1 Detailed Description

The core implementation of the Parma Polyhedra Library is written in C++.

See Namespace, Hierarchical and Compound indexes for additional information about each single data type.

7.1.2 Define Documentation

7.1.2.1 #define PPL_VERSION "0.9"

A string containing the PPL version.

Let `M` and `m` denote the numbers associated to PPL_VERSION_MAJOR and PPL_VERSION_MINOR, respectively. The format of PPL_VERSION is `M "." m` if both PPL_VERSION_REVISION (`r`) and PPL_VERSION_BETA (`b`)are zero, `M "." m "pre" b` if PPL_VERSION_REVISION is zero and PPL_VERSION_BETA is not zero, `M "." m "." r` if PPL_VERSION_REVISION is not zero and PPL_VERSION_BETA is zero, `M "." m "." r "pre" b` if neither PPL_VERSION_REVISION nor PPL_VERSION_BETA are zero.

7.1.3 Typedef Documentation

7.1.3.1 typedef COEFFICIENT_TYPE Parma_Polyhedra_Library::Coefficient

An alias for easily naming the type of PPL coefficients.

Objects of type Coefficient are used to implement the integral valued coefficients occurring in linear expressions, constraints, generators, intervals, bounding boxes and so on. Depending on the chosen configuration options (see file `README.configure`), a Coefficient may actually be:

- The GMP_Integer type, which in turn is an alias for the `mpz_class` type implemented by the C++ interface of the GMP library (this is the default configuration);
- An instance of the Checked_Number class template: with its default policy (Checked_Number_-Default_Policy), this implements overflow detection on top of a native integral type (available template instances include checked integers having 8, 16, 32 or 64 bits); with the Checked_Number_-Transparent_Policy, this is a wrapper for native integral types with no overflow detection (available template instances are as above).

7.1.4 Enumeration Type Documentation

7.1.4.1 enum Parma_Polyhedra_Library::Result

Possible outcomes of a checked arithmetic computation.

Enumerator:

VC_NORMAL Ordinary result class.

V_LT The computed result is inexact and rounded up.

V_GT The computed result is inexact and rounded down.

V_EQ The computed result is exact.

V_NE The computed result is inexact.

V_LE The computed result may be inexact and rounded up.

V_GE The computed result may be inexact and rounded down.

V_LGE The computed result may be inexact.
VC_MINUS_INFINITY Negative infinity unrepresentable result class.
V_NEG_OVERFLOW A negative overflow occurred.
VC_PLUS_INFINITY Positive infinity unrepresentable result class.
V_POS_OVERFLOW A positive overflow occurred.
VC_NAN Not a number result class.
V_CVT_STR_UNK Converting from unknown string.
V_DIV_ZERO Dividing by zero.
V_INF_ADD_INF Adding two infinities having opposite signs.
V_INF_DIV_INF Dividing two infinities.
V_INF_MOD Taking the modulus of an infinity.
V_INF_MUL_ZERO Multiplying an infinity by zero.
V_INF_SUB_INF Subtracting two infinities having the same sign.
V_MOD_ZERO Computing a remainder modulo zero.
V_SQRT_NEG Taking the square root of a negative number.
V_UNKNOWN_NEG_OVERFLOW Unknown result due to intermediate negative overflow.
V_UNKNOWN_POS_OVERFLOW Unknown result due to intermediate positive overflow.
V_UNORD_COMP Unordered comparison.

7.1.4.2 enum Parma_Polyhedra_Library::Rounding_Dir

Rounding directions for arithmetic computations.

Enumerator:

ROUND_DOWN Round toward $-\infty$.
ROUND_UP Round toward $+\infty$.
ROUND_IGNORE Rounding is delegated to lower level. Result info is evaluated lazily.
ROUND_NOT_NEEDED Rounding is not needed: client code must ensure the operation is exact.

7.1.4.3 enum Parma_Polyhedra_Library::Degenerate_Element

Kinds of degenerate abstract elements.

Enumerator:

UNIVERSE The universe element, i.e., the whole vector space.
EMPTY The empty element, i.e., the empty set.

7.1.4.4 enum Parma_Polyhedra_Library::Relation_Symbol

Relation symbols.

Enumerator:

LESS_THAN Less than.
LESS_THAN_OR_EQUAL Less than or equal to.
EQUAL Equal to.
GREATER_THAN_OR_EQUAL Greater than or equal to.
GREATER_THAN Greater than.

7.1.4.5 enum Parma_Polyhedra_Library::Complexity_Class

Complexity pseudo-classes.

Enumerator:

POLYNOMIAL_COMPLEXITY Worst-case polynomial complexity.

SIMPLEX_COMPLEXITY Worst-case exponential complexity but typically polynomial behavior.

ANY_COMPLEXITY Any complexity.

7.1.4.6 enum Parma_Polyhedra_Library::Optimization_Mode

Possible optimization modes.

Enumerator:

MINIMIZATION Minimization is requested.

MAXIMIZATION Maximization is requested.

7.1.4.7 enum Parma_Polyhedra_Library::LP_Problem_Status

Possible outcomes of the LP_Problem solver.

Enumerator:

UNFEASIBLE_LP_PROBLEM The problem is unfeasible.

UNBOUNDED_LP_PROBLEM The problem is unbounded.

OPTIMIZED_LP_PROBLEM The problem has an optimal solution.

7.1.5 Variable Documentation

7.1.5.1 const Throwable∗ volatile Parma_Polyhedra_Library::abandon_expensive_computations

A pointer to an exception object.

This pointer, which is initialized to zero, is repeatedly checked along any super-linear (i.e., computationally expensive) computation path in the library. When it is found nonzero the exception it points to is thrown. In other words, making this pointer point to an exception (and leaving it in this state) ensures that the library will return control to the client application, possibly by throwing the given exception, within a time that is a linear function of the size of the representation of the biggest object (powerset of polyhedra, polyhedron, system of constraints or generators) on which the library is operating upon.

Note:

The only sensible way to assign to this pointer is from within a signal handler or from a parallel thread. For this reason, the library, apart from ensuring that the pointer is initially set to zero, never assigns to it. In particular, it does not zero it again when the exception is thrown: it is the client's responsibility to do so.

7.2 C Language Interface

Some details about the C Interface.

Version Checking

- #define PPL_VERSION_MAJOR 0

  The major number of the PPL version.

- #define PPL_VERSION_MINOR 9

  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.

- #define PPL_VERSION "0.9"

  A string containing the PPL version.

- int ppl_version_major (void)

  Returns the major number of the PPL version.

- int ppl_version_minor (void)

  Returns the minor number of the PPL version.

- int ppl_version_revision (void)

  Returns the revision number of the PPL version.

- int ppl_version_beta (void)

  Returns the beta number of the PPL version.

- int ppl_version (const char ∗∗p)

  Writes to `∗p` *a pointer to a character string containing the PPL version.*

- int ppl_banner (const char ∗∗p)

  Writes to `∗p` *a pointer to a character string containing the PPL banner.*

Simple I/O Functions

- typedef const char ∗ ppl_io_variable_output_function_type (ppl_dimension_type var)

  The type of output functions used for printing variables.

- int ppl_io_print_variable (ppl_dimension_type var)

  Pretty-prints `var` *to* `stdout`*.*

- int ppl_io_fprint_variable (FILE ∗stream, ppl_dimension_type var)

  Pretty-prints `var` *to the given output* `stream`*.*

- int ppl_io_set_variable_output_function (ppl_io_variable_output_function_type ∗p)

  Sets the output function to be used for printing variables to `p`*.*

- int ppl_io_get_variable_output_function (ppl_io_variable_output_function_type ∗∗pp)

  Writes a pointer to the current variable output function to `pp`.

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 at by `h`.

Functions Related to Coefficients

- int ppl_new_Coefficient (ppl_Coefficient_t ∗pc)

  Creates a new coefficient with value 0 and writes a 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 a 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 a 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 makes some noise if* `c` *is broken. Useful for debugging purposes.*

- int ppl_Coefficient_is_bounded (void)

  Returns a positive integer if coefficients are bounded; returns 0 otherwise.

- int ppl_Coefficient_min (mpz_t min)

  Returns a positive integer if coefficients are bounded, in which case `min` *is set to their minimum value; returns 0 otherwise.*

- int ppl_Coefficient_max (mpz_t max)

  Returns a positive integer if coefficients are bounded, in which case `max` *is set to their maximum value; returns 0 otherwise.*

Functions Related to Linear Expressions

- int ppl_new_Linear_Expression (ppl_Linear_Expression_t ∗ple)

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

- int ppl_new_Linear_Expression_with_dimension (ppl_Linear_Expression_t ∗ple, ppl_dimension_type d)

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

- int ppl_new_Linear_Expression_from_Linear_Expression (ppl_Linear_Expression_t ∗ple, ppl_const_Linear_Expression_t le)

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

- int ppl_new_Linear_Expression_from_Constraint (ppl_Linear_Expression_t ∗ple, ppl_const_Constraint_t c)

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

- int ppl_new_Linear_Expression_from_Generator (ppl_Linear_Expression_t ∗ple, ppl_const_Generator_t g)

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

- int ppl_delete_Linear_Expression (ppl_const_Linear_Expression_t le)

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

- int ppl_assign_Linear_Expression_from_Linear_Expression (ppl_Linear_Expression_t dst, ppl_const_Linear_Expression_t src)

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

- int ppl_Linear_Expression_add_to_coefficient (ppl_Linear_Expression_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_Linear_Expression_add_to_inhomogeneous (ppl_Linear_Expression_t le, ppl_const_-Coefficient_t n)

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

- int ppl_add_Linear_Expression_to_Linear_Expression (ppl_Linear_Expression_t dst, ppl_const_-Linear_Expression_t src)

  Adds the linear expression src *to* dst.

- int ppl_subtract_Linear_Expression_from_Linear_Expression (ppl_Linear_Expression_t dst, ppl_-const_Linear_Expression_t src)

  Subtracts the linear expression src *from* dst.

- int ppl_multiply_Linear_Expression_by_Coefficient (ppl_Linear_Expression_t le, ppl_const_-Coefficient_t n)

  Multiply the linear expression dst *by* n.

- int ppl_Linear_Expression_space_dimension (ppl_const_Linear_Expression_t le, ppl_dimension_-type ∗m)

  Writes to m *the space dimension of* le.

- int ppl_Linear_Expression_coefficient (ppl_const_Linear_Expression_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_Linear_Expression_inhomogeneous_term (ppl_const_Linear_Expression_t le, ppl_-Coefficient_t n)

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

- int ppl_Linear_Expression_OK (ppl_const_Linear_Expression_t le)

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

Functions Related to Constraints

- int ppl_new_Constraint (ppl_Constraint_t ∗pc, ppl_const_Linear_Expression_t le, enum ppl_-enum_Constraint_Type rel)

  *Creates the new constraint '*le rel *0' and writes a 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 a 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. *A 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 a 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, ppl_dimension_type ∗m)

  Writes to `m` *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 makes some noise if* `c` *is broken. Useful for debugging purposes.*

Functions Related to Constraint Systems

- int ppl_new_Constraint_System (ppl_Constraint_System_t ∗pcs)

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

- int ppl_new_Constraint_System_zero_dim_empty (ppl_Constraint_System_t ∗pcs)

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

- int ppl_new_Constraint_System_from_Constraint (ppl_Constraint_System_t ∗pcs, ppl_const_-Constraint_t c)

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

- int ppl_new_Constraint_System_from_Constraint_System (ppl_Constraint_System_t ∗pcs, ppl_-const_Constraint_System_t cs)

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

- int ppl_delete_Constraint_System (ppl_const_Constraint_System_t cs)

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

- int ppl_assign_Constraint_System_from_Constraint_System (ppl_Constraint_System_t dst, ppl_-const_Constraint_System_t src)

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

- int ppl_Constraint_System_space_dimension (ppl_const_Constraint_System_t cs, ppl_dimension_-type ∗m)

Writes to m *the dimension of the vector space enclosing* cs*.*

- int ppl_Constraint_System_has_strict_inequalities (ppl_const_Constraint_System_t cs)

  Returns a positive integer if cs *contains any (non-trivial) strict inequality; returns 0 otherwise.*

- int ppl_Constraint_System_clear (ppl_Constraint_System_t cs)

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

- int ppl_Constraint_System_insert_Constraint (ppl_Constraint_System_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_Constraint_System_OK (ppl_const_Constraint_System_t c)

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

- int ppl_new_Constraint_System_const_iterator (ppl_Constraint_System_const_iterator_t ∗pcit)

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

- int ppl_new_Constraint_System_const_iterator_from_Constraint_System_const_iterator (ppl_-Constraint_System_const_iterator_t ∗pcit, ppl_const_Constraint_System_const_iterator_t cit)

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

- int ppl_delete_Constraint_System_const_iterator (ppl_const_Constraint_System_const_iterator_-t cit)

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

- int ppl_assign_Constraint_System_const_iterator_from_Constraint_System_const_iterator (ppl_-Constraint_System_const_iterator_t dst, ppl_const_Constraint_System_const_iterator_t src)

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

- int ppl_Constraint_System_begin (ppl_const_Constraint_System_t cs, ppl_Constraint_System_-const_iterator_t cit)

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

- int ppl_Constraint_System_end (ppl_const_Constraint_System_t cs, ppl_Constraint_System_-const_iterator_t cit)

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

- int ppl_Constraint_System_const_iterator_dereference (ppl_const_Constraint_System_const_-iterator_t cit, ppl_const_Constraint_t ∗pc)

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

- int ppl_Constraint_System_const_iterator_increment (ppl_Constraint_System_const_iterator_t cit)

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

- int ppl_Constraint_System_const_iterator_equal_test (ppl_const_Constraint_System_const_-iterator_t x, ppl_const_Constraint_System_const_iterator_t y)

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

Functions Related to Generators

- int ppl_new_Generator (ppl_Generator_t ∗pg, ppl_const_Linear_Expression_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. A 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 a 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 a 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 a 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, ppl_dimension_type ∗m)

  Writes to `m` *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 makes some noise if* `g` *is broken. Useful for debugging purposes.*

Functions Related to Generator Systems

- int ppl_new_Generator_System (ppl_Generator_System_t ∗pgs)

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

- int ppl_new_Generator_System_from_Generator (ppl_Generator_System_t ∗pgs, ppl_const_-Generator_t g)

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

- int ppl_new_Generator_System_from_Generator_System (ppl_Generator_System_t ∗pgs, ppl_-const_Generator_System_t gs)

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

- int ppl_delete_Generator_System (ppl_const_Generator_System_t gs)

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

- int ppl_assign_Generator_System_from_Generator_System (ppl_Generator_System_t dst, ppl_-const_Generator_System_t src)

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

- int ppl_Generator_System_space_dimension (ppl_const_Generator_System_t gs, ppl_dimension_-type ∗m)

  Writes to `m` *the dimension of the vector space enclosing* `gs`*.*

- int ppl_Generator_System_clear (ppl_Generator_System_t gs)

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

- int ppl_Generator_System_insert_Generator (ppl_Generator_System_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_Generator_System_OK (ppl_const_Generator_System_t c)

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

- int ppl_new_Generator_System_const_iterator (ppl_Generator_System_const_iterator_t ∗pgit)

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

- int ppl_new_Generator_System_const_iterator_from_Generator_System_const_iterator (ppl_-Generator_System_const_iterator_t ∗pgit, ppl_const_Generator_System_const_iterator_t git)

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

- int ppl_delete_Generator_System_const_iterator (ppl_const_Generator_System_const_iterator_-t git)

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

- int ppl_assign_Generator_System_const_iterator_from_Generator_System_const_iterator (ppl_-Generator_System_const_iterator_t dst, ppl_const_Generator_System_const_iterator_t src)

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

- int ppl_Generator_System_begin (ppl_const_Generator_System_t gs, ppl_Generator_System_-const_iterator_t git)

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

- int ppl_Generator_System_end (ppl_const_Generator_System_t gs, ppl_Generator_System_const_-iterator_t git)

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

- int ppl_Generator_System_const_iterator_dereference (ppl_const_Generator_System_const_-iterator_t git, ppl_const_Generator_t ∗pg)

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

- int ppl_Generator_System_const_iterator_increment (ppl_Generator_System_const_iterator_t git)

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

- int ppl_Generator_System_const_iterator_equal_test (ppl_const_Generator_System_const_-iterator_t x, ppl_const_Generator_System_const_iterator_t y)

  Returns a positive integer if the iterators corresponding to `x` *and* `y` *are equal; returns 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 a universe C 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 a 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 C polyhedron of space 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 space 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 C polyhedron that is a copy of `ph`*; writes a 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 C polyhedron that is a copy of of the NNC polyhedron `ph`*; writes a 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 the C polyhedron `ph`*; writes a 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 a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_C_Polyhedron_from_Constraint_System (ppl_Polyhedron_t ∗pph, ppl_const_-Constraint_System_t cs)

  Builds a new C polyhedron from the system of constraints `cs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_C_Polyhedron_recycle_Constraint_System (ppl_Polyhedron_t ∗pph, ppl_Constraint_-System_t cs)

  Builds a new C polyhedron recycling the system of constraints `cs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_NNC_Polyhedron_from_Constraint_System (ppl_Polyhedron_t ∗pph, ppl_const_-Constraint_System_t cs)

  Builds a new NNC polyhedron from the system of constraints `cs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_NNC_Polyhedron_recycle_Constraint_System (ppl_Polyhedron_t ∗pph, ppl_-Constraint_System_t cs)

  Builds a new NNC polyhedron recycling the system of constraints `cs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_C_Polyhedron_from_Generator_System (ppl_Polyhedron_t ∗pph, ppl_const_-Generator_System_t gs)

  Builds a new C polyhedron from the system of generators `gs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_C_Polyhedron_recycle_Generator_System (ppl_Polyhedron_t ∗pph, ppl_Generator_-System_t gs)

  Builds a new C polyhedron recycling the system of generators `gs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_NNC_Polyhedron_from_Generator_System (ppl_Polyhedron_t ∗pph, ppl_const_-Generator_System_t gs)

  Builds a new NNC polyhedron from the system of generators `gs` *and writes a handle for the newly created polyhedron at address* `pph`.

- int ppl_new_NNC_Polyhedron_recycle_Generator_System (ppl_Polyhedron_t ∗pph, ppl_-Generator_System_t gs)

  Builds a new NNC polyhedron recycling the system of generators `gs` *and writes a handle for the newly created polyhedron at address* `pph`.

- 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 NNC 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 C polyhedron src *to the C 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, ppl_dimension_type ∗m)

  Writes to m *the dimension of the vector space enclosing* ph*.*

- int ppl_Polyhedron_affine_dimension (ppl_const_Polyhedron_t ph)

  Writes to m *the affine dimension of* ph *(not to be confused with the dimension of its enclosing vector space) or 0, if* ph *is empty.*

- int ppl_Polyhedron_constraints (ppl_const_Polyhedron_t ph, ppl_const_Constraint_System_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_Constraint_-System_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_Generator_System_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_Generator_-System_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 *and 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 *and 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_Linear_-Expression_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_Linear_-Expression_t le)

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

- int ppl_Polyhedron_maximize (ppl_const_Polyhedron_t ph, ppl_const_Linear_Expression_t le, ppl_Coefficient_t sup_n, ppl_Coefficient_t sup_d, int ∗pmaximum, ppl_Generator_t point)

  Returns a positive integer if `ph` *is not empty and* `le` *is bounded from above in* `ph`*, in which case the supremum value and a point where* `le` *reaches it are computed.*

- int ppl_Polyhedron_minimize (ppl_const_Polyhedron_t ph, ppl_const_Linear_Expression_t le, ppl_Coefficient_t inf_n, ppl_Coefficient_t inf_d, int ∗pminimum, ppl_Generator_t point)

  Returns a positive integer if `ph` *is not empty and* `le` *is bounded from below in* `ph`*, in which case the infimum value and a point where* `le` *reaches it are computed.*

- 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; returns 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 makes 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_const_Constraint_System_t cs)

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

- int ppl_Polyhedron_add_constraints_and_minimize (ppl_Polyhedron_t ph, ppl_const_Constraint_-System_t cs)

  Adds a copy of 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_const_Generator_System_t gs)

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

- int ppl_Polyhedron_add_generators_and_minimize (ppl_Polyhedron_t ph, ppl_const_Generator_-System_t gs)

  Adds a copy of 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_add_recycled_constraints (ppl_Polyhedron_t ph, ppl_Constraint_System_t cs)

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

- int ppl_Polyhedron_add_recycled_constraints_and_minimize (ppl_Polyhedron_t ph, ppl_-Constraint_System_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_recycled_generators (ppl_Polyhedron_t ph, ppl_Generator_System_t gs)

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

- int ppl_Polyhedron_add_recycled_generators_and_minimize (ppl_Polyhedron_t ph, ppl_-Generator_System_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 to* `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 to* `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* `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* `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-difference of* `x` *and* `y`*.*

- int ppl_Polyhedron_affine_image (ppl_Polyhedron_t ph, ppl_dimension_type var, ppl_const_-Linear_Expression_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_-Linear_Expression_t le, ppl_const_Coefficient_t d)

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

- int ppl_Polyhedron_bounded_affine_image (ppl_Polyhedron_t ph, ppl_dimension_type var, ppl_-const_Linear_Expression_t lb, ppl_const_Linear_Expression_t ub, ppl_const_Coefficient_t d)

  Assigns to `ph` *the image of* `ph` *with respect to the generalized affine transfer relation* $\frac{\mathrm{lb}}{\mathrm{d}} \leq \mathrm{var}' \leq \frac{\mathrm{ub}}{\mathrm{d}}$.

- int ppl_Polyhedron_bounded_affine_preimage (ppl_Polyhedron_t ph, ppl_dimension_type var, ppl_const_Linear_Expression_t lb, ppl_const_Linear_Expression_t ub, ppl_const_Coefficient_t d)

  Assigns to `ph` *the preimage of* `ph` *with respect to the generalized affine transfer relation* $\frac{\mathrm{lb}}{\mathrm{d}} \leq \mathrm{var}' \leq \frac{\mathrm{ub}}{\mathrm{d}}$.

- int ppl_Polyhedron_generalized_affine_image (ppl_Polyhedron_t ph, ppl_dimension_type var, enum ppl_enum_Constraint_Type relsym, ppl_const_Linear_Expression_t le, ppl_const_-Coefficient_t d)

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

- int ppl_Polyhedron_generalized_affine_preimage (ppl_Polyhedron_t ph, ppl_dimension_type var, enum ppl_enum_Constraint_Type relsym, ppl_const_Linear_Expression_t le, ppl_const_-Coefficient_t d)

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

- int ppl_Polyhedron_generalized_affine_image_lhs_rhs (ppl_Polyhedron_t ph, ppl_const_Linear_-Expression_t lhs, enum ppl_enum_Constraint_Type relsym, ppl_const_Linear_Expression_t rhs)

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

- int ppl_Polyhedron_generalized_affine_preimage_lhs_rhs (ppl_Polyhedron_t ph, ppl_const_Linear_Expression_t lhs, enum ppl_enum_Constraint_Type relsym, ppl_const_Linear_Expression_t rhs)

  Assigns to ph *the preimage of* ph *with respect to the generalized affine transfer relation* $\mathrm{lhs}' \bowtie \mathrm{rhs}$, *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_Constraint_System_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_Constraint_System_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_Constraint_System_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_Constraint_System_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_Constraint_System_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_Constraint_System_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_Constraint_System_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_Constraint_System_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_space_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_space_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_space_dimensions (ppl_Polyhedron_t ph, ppl_dimension_type ds[ ], size_t n)

  Removes from the vector space enclosing `ph` *the space 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_space_dimensions (ppl_Polyhedron_t ph, ppl_dimension_type d)

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

- int ppl_Polyhedron_map_space_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.*

- int ppl_Polyhedron_expand_space_dimension (ppl_Polyhedron_t ph, ppl_dimension_type d, ppl_-dimension_type m)

  Expands the d-th dimension of the vector space enclosing `ph` *to* `m` *new space dimensions.*

- int ppl_Polyhedron_fold_space_dimensions (ppl_Polyhedron_t ph, ppl_dimension_type ds[ ], size_t n, ppl_dimension_type d)

  Modifies `ph` *by folding the space dimensions contained in the first* `n` *positions of the array* `ds` *into dimension* `d`*. The presence of duplicates in* `ds` *is a waste but an innocuous one.*

Functions Related to LP Problems

- int ppl_new_LP_Problem_trivial (ppl_LP_Problem_t ∗plp)

  Builds a trivial LP problem and writes a handle to it at address `plp`*.*

- int ppl_new_LP_Problem (ppl_LP_Problem_t ∗plp, ppl_const_Constraint_System_t cs, ppl_const_-Linear_Expression_t le, int m)

  Builds an LP problem having feasible region `cs`*, objective function* `le` *and optimization mode* `m`*; writes a handle to it at address* `plp`*.*

- int ppl_new_LP_Problem_from_LP_Problem (ppl_LP_Problem_t ∗plp, ppl_const_LP_Problem_-t lp)

  Builds an LP problem that is a copy of `lp`*; writes a handle for the newly created system at address* `plp`*.*

- int ppl_delete_LP_Problem (ppl_const_LP_Problem_t lp)

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

- int ppl_assign_LP_Problem_from_LP_Problem (ppl_LP_Problem_t dst, ppl_const_LP_Problem_t src)

  Assigns a copy of the LP problem `src` *to* `dst`*.*

- int ppl_LP_Problem_space_dimension (ppl_const_LP_Problem_t lp, ppl_dimension_type ∗m)

  Writes to `m` *the dimension of the vector space enclosing* `lp`*.*

- int ppl_LP_Problem_constraints (ppl_const_LP_Problem_t lp, ppl_const_Constraint_System_-t ∗pcs)

  Writes a const handle to the constraint system defining the feasible region of the LP problem `lp` *at address* `pcs`*.*

- int ppl_LP_Problem_objective_function (ppl_const_LP_Problem_t lp, ppl_const_Linear_-Expression_t ∗ple)

  Writes a const handle to the linear expression defining the objective function of the LP problem `lp` *at address* `ple`*.*

- int ppl_LP_Problem_optimization_mode (ppl_const_LP_Problem_t lp)

  Returns the optimization mode of the LP problem `lp`*.*

- int ppl_LP_Problem_clear (ppl_LP_Problem_t lp)

  Resets the LP problem to be a trivial problem of space dimension 0.

- int ppl_LP_Problem_add_constraint (ppl_LP_Problem_t lp, ppl_const_Constraint_t c)

  Modifies the feasible region of the LP problem `lp` *by adding a copy of the constraint* `c`*; the space dimension is increased, if necessary.*

- int ppl_LP_Problem_add_constraints (ppl_LP_Problem_t lp, ppl_const_Constraint_System_t cs)

  Modifies the feasible region of the LP problem `lp` *by adding a copy of the constraints in* `cs`*; the space dimension is increased, if necessary.*

- int ppl_LP_Problem_set_objective_function (ppl_LP_Problem_t lp, ppl_const_Linear_-Expression_t le)

  Sets the objective function of the LP problem `lp` *to a copy of* `le`.

- int ppl_LP_Problem_set_optimization_mode (ppl_LP_Problem_t lp, int mode)

  Sets the optimization mode of the LP problem `lp` *to* `mode`.

- int ppl_LP_Problem_is_satisfiable (ppl_const_LP_Problem_t lp)

  Returns a positive integer if `lp` *is satisfiable; returns 0 otherwise.*

- int ppl_LP_Problem_solve (ppl_const_LP_Problem_t lp)

  Solves the LP problem `lp`*, returning an exit status.*

- int ppl_LP_Problem_evaluate_objective_function (ppl_const_LP_Problem_t lp, ppl_const_-Generator_t g, ppl_Coefficient_t num, ppl_Coefficient_t den)

  Evaluates the objective function of `lp` *on point* `g`.

- int ppl_LP_Problem_feasible_point (ppl_const_LP_Problem_t lp, ppl_const_Generator_t ∗pg)

  Writes a const handle to a feasible point for the LP problem `lp` *at address* `pg`.

- int ppl_LP_Problem_optimizing_point (ppl_const_LP_Problem_t lp, ppl_const_Generator_t ∗pg)

  Writes a const handle to an optimizing point for the LP problem `lp` *at address* `pg`.

- int ppl_LP_Problem_optimal_value (ppl_const_LP_Problem_t lp, ppl_Coefficient_t num, ppl_-Coefficient_t den)

  Returns the optimal value for `lp`.

- int ppl_LP_Problem_OK (ppl_const_LP_Problem_t lp)

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

Typedefs

- typedef size_t ppl_dimension_type

  An unsigned integral type for representing space dimensions.

- typedef ppl_Coefficient_tag ∗ ppl_Coefficient_t

  Opaque pointer.

- typedef ppl_Coefficient_tag const ∗ ppl_const_Coefficient_t

  Opaque pointer to const object.

- typedef ppl_Linear_Expression_tag ∗ ppl_Linear_Expression_t

  Opaque pointer.

- typedef ppl_Linear_Expression_tag const ∗ ppl_const_Linear_Expression_t

  Opaque pointer to const object.

- typedef ppl_Constraint_tag ∗ ppl_Constraint_t

  Opaque pointer.

- typedef ppl_Constraint_tag const ∗ ppl_const_Constraint_t

  Opaque pointer to const object.

- typedef ppl_Constraint_System_tag ∗ ppl_Constraint_System_t

  Opaque pointer.

- typedef ppl_Constraint_System_tag const ∗ ppl_const_Constraint_System_t

  Opaque pointer to const object.

- typedef ppl_Constraint_System_const_iterator_tag ∗ ppl_Constraint_System_const_iterator_t

  Opaque pointer.

- typedef ppl_Constraint_System_const_iterator_tag const ∗ ppl_const_Constraint_System_const_-iterator_t

  Opaque pointer to const object.

- typedef ppl_Generator_tag ∗ ppl_Generator_t

  Opaque pointer.

- typedef ppl_Generator_tag const ∗ ppl_const_Generator_t

  Opaque pointer to const object.

- typedef ppl_Generator_System_tag ∗ ppl_Generator_System_t

  Opaque pointer.

- typedef ppl_Generator_System_tag const ∗ ppl_const_Generator_System_t

  Opaque pointer to const object.

- typedef ppl_Generator_System_const_iterator_tag ∗ ppl_Generator_System_const_iterator_t

  Opaque pointer.

- typedef ppl_Generator_System_const_iterator_tag const ∗ ppl_const_Generator_System_const_-iterator_t

  Opaque pointer to const object.

- typedef ppl_Polyhedron_tag ∗ ppl_Polyhedron_t

  Opaque pointer.

- typedef ppl_Polyhedron_tag const ∗ ppl_const_Polyhedron_t

  Opaque pointer to const object.

- typedef ppl_LP_Problem_tag ∗ ppl_LP_Problem_t

  Opaque pointer.

- typedef ppl_LP_Problem_tag const ∗ ppl_const_LP_Problem_t

  Opaque pointer to const object.

Enumerations

- enum ppl_enum_error_code {

  PPL_ERROR_OUT_OF_MEMORY, PPL_ERROR_INVALID_ARGUMENT, PPL_ERROR_-DOMAIN_ERROR, PPL_ERROR_LENGTH_ERROR,

  PPL_ARITHMETIC_OVERFLOW, PPL_STDIO_ERROR, PPL_ERROR_INTERNAL_ERROR, PPL_ERROR_UNKNOWN_STANDARD_EXCEPTION,

  PPL_ERROR_UNEXPECTED_ERROR }

  Defines the error codes that any function may 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.

- int PPL_LP_PROBLEM_STATUS_UNFEASIBLE

  Code of the "unfeasible LP problem" status.

- int PPL_LP_PROBLEM_STATUS_UNBOUNDED

  Code of the "unbounded LP problem" status.

- int PPL_LP_PROBLEM_STATUS_OPTIMIZED

Code of the "optimized LP problem" status.

- int PPL_LP_PROBLEM_MAXIMIZATION

  Code of the "maximization" optimization mode.

- int PPL_LP_PROBLEM_MINIMIZATION

  Code of the "minimization" optimization mode.

- 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.

7.2.1 Detailed Description

Some details about the C Interface.

All the declarations needed for using the PPL's C interface (preprocessor symbols, data types, variables and functions) are collected in the header file `ppl_c.h`. This file, which is designed to work with pre-ANSI and ANSI C compilers as well as C99 and C++ compilers, should be included, either directly or via some other header file, with the directive

```
#include <ppl_c.h>
```

If this directive does not work, then your compiler is unable to find the file `ppl_c.h`. So check that the library is installed (if it is not installed, you may want to `make install`, perhaps with root privileges) in the right place (if not you may want to reconfigure the library using the appropriate pathname for the `-prefix` option); and that your compiler knows where it is installed (if not you should add the path to the directory where `ppl_c.h` is located to the compiler's include file search path; this is usually done with the `-I` option).

The name space of the PPL's C interface is `PPL_*` for preprocessor symbols, enumeration values and variables; and `ppl_*` for data types and function names. The interface systematically uses *opaque data types* (generic pointers that completely hide the internal representations from the client code) and provides all required access functions. By using just the interface, the client code can exploit all the functionalities of the library yet avoid directly manipulating the library's data structures. The advantages are that (1) applications do not depend on the internals of the library (these may change from release to release), and (2) the interface invariants can be thoroughly checked (by the access functions).

The PPL's C interface is initialized by means of the `ppl_initialize` function. This function must be called *before using any other interface of the library*. The application can release the resources allocated by the library by calling the `ppl_finalize` function. After this function is called *no other interface of the library may be used* until the interface is re-initialized using `ppl_initialize`.

Any application using the PPL should make sure that only the intended version(s) of the library are ever used. The version used can be checked at compile-time thanks to the macros PPL_VERSION_MAJOR, PPL_VERSION_MINOR, PPL_VERSION_REVISION and PPL_VERSION_BETA, which give, respectively major, minor, revision and beta numbers of the PPL version. This is an example of their use:

```
#if PPL_VERSION_MAJOR == 0 && PPL_VERSION_MINOR < 6
# error "PPL version 0.6 or following is required"
#endif
```

Compile-time checking, however, is not normally enough, particularly in an environment where there is dynamic linking. Run-time checking can be performed by means of the functions `ppl_version_major`, `ppl_version_minor`, `ppl_version_revision`, and `ppl_version_beta`. The PPL's C interface also provides functions `ppl_version`, returning character string containing the full version number, and `ppl_banner`, returning a string that, in addition, provides (pointers to) other useful information for the library user.

All programs using the PPL's C interface must link with the following libraries: `libppl_c` (PPL's C interface), `libppl` (PPL's core), `libgmpxx` (GMP's C++ interface), and `libgmp` (GMP's library core). On most Unix-like systems, this is done by adding `-lppl_c`, `-lppl`, `-lgmpxx`, and `-lgmp` to the compiler's or linker's command line. For example:

```
gcc myprogram.o -lppl_c -lppl -lgmpxx -lgmp
```

If this does not work, it means that your compiler/linker is not finding the libraries where it expects. Again, this could be because you forgot to install the library or you installed it in a non-standard location. In the latter case you will need to use the appropriate options (usually `-L`) and, if you use shared libraries, some sort of run-time path selection mechanisms. Consult your compiler's documentation for details. Notice that the PPL is built using `Libtool` and an application can exploit this fact to significantly simplify the linking phase. See Libtool's documentation for details. Those working under Linux can find a lot of useful information on how to use program libraries (including static, shared, and dynamically loaded libraries) in the `Program Library HOWTO`.

For examples on how to use the functions provided by the C interface, you are referred to the directory `demos/ppl_lpsol/` in the source distribution. It contains a *Linear Programming* solver written in C. In order to use this solver you will need to install `GLPK` (the GNU Linear Programming Kit): this is used to read linear programs in MPS format.

7.2.2 Define Documentation

7.2.2.1 #define PPL_VERSION "0.9"

A string containing the PPL version.

Let `M` and `m` denote the numbers associated to PPL_VERSION_MAJOR and PPL_VERSION_MINOR, respectively. The format of PPL_VERSION is `M "." m` if both PPL_VERSION_REVISION (`r`) and PPL_VERSION_BETA (`b`)are zero, `M "." m "pre" b` if PPL_VERSION_REVISION is zero and PPL_VERSION_BETA is not zero, `M "." m "." r` if PPL_VERSION_REVISION is not zero and PPL_VERSION_BETA is zero, `M "." m "." r "pre" b` if neither PPL_VERSION_REVISION nor PPL_VERSION_BETA are zero.

7.2.3 Typedef Documentation

7.2.3.1 typedef const char∗ ppl_io_variable_output_function_type(ppl_dimension_type var)

The type of output functions used for printing variables.

An output function for variables must write a textual representation for `var` to a character buffer, null-terminate it, and return a pointer to the beginning of the buffer. In case the operation fails, 0 should be returned and perhaps `errno` should be set in a meaningful way. The library does nothing with the buffer, besides printing its contents.

7.2.4 Enumeration Type Documentation

7.2.4.1 enum ppl_enum_error_code

Defines the error codes that any function may return.

Enumerator:

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_DOMAIN_ERROR A function has been invoked outside its domain of definition.

PPL_ERROR_LENGTH_ERROR The construction of an object that would exceed its maximum permitted size was attempted.

PPL_ARITHMETIC_OVERFLOW An arithmetic overflow occurred and the computation was consequently interrupted. This can *only* happen in library's incarnations using bounded integers as coefficients.

PPL_STDIO_ERROR An error occurred during a C input/output operation. A more precise indication of what went wrong is available via `errno`.

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.

7.2.4.2 enum ppl_enum_Constraint_Type

Describes the relations represented by a constraint.

Enumerator:

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 $e > 0$.

7.2.4.3 enum ppl_enum_Generator_Type

Describes the different kinds of generators.

Enumerator:

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.

7.2.5 Function Documentation

7.2.5.1 int ppl_banner (const char ∗∗ *p*)

Writes to `*p` a pointer to a character string containing the PPL banner.

The banner provides 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.

7.2.5.2 int ppl_initialize (void)

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

Returns:

`PPL_ERROR_INVALID_ARGUMENT` if the library was already initialized.

7.2.5.3 int ppl_finalize (void)

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

Returns:

`PPL_ERROR_INVALID_ARGUMENT` if the library was already finalized.

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

Installs the user-defined error handler pointed at 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 is not guaranteed after the handler has returned.

7.2.5.5 int ppl_new_C_Polyhedron_from_Constraint_System (ppl_Polyhedron_t ∗ *pph*, ppl_const_Constraint_System_t *cs*)

Builds a new C polyhedron from the system of constraints `cs` and writes a handle for the newly created polyhedron at address `pph`.

The new polyhedron will inherit the space dimension of `cs`.

7.2.5.6 int ppl_new_C_Polyhedron_recycle_Constraint_System (ppl_Polyhedron_t * *pph*, ppl_-Constraint_System_t *cs*)

Builds a new C polyhedron recycling the system of constraints `cs` and writes a 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.

7.2.5.7 int ppl_new_NNC_Polyhedron_from_Constraint_System (ppl_Polyhedron_t * *pph*, ppl_-const_Constraint_System_t *cs*)

Builds a new NNC polyhedron from the system of constraints `cs` and writes a handle for the newly created polyhedron at address `pph`.

The new polyhedron will inherit the space dimension of `cs`.

7.2.5.8 int ppl_new_NNC_Polyhedron_recycle_Constraint_System (ppl_Polyhedron_t * *pph*, ppl_-Constraint_System_t *cs*)

Builds a new NNC polyhedron recycling the system of constraints `cs` and writes a 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.

7.2.5.9 int ppl_new_C_Polyhedron_from_Generator_System (ppl_Polyhedron_t * *pph*, ppl_const_-Generator_System_t *gs*)

Builds a new C polyhedron from the system of generators `gs` and writes a handle for the newly created polyhedron at address `pph`.

The new polyhedron will inherit the space dimension of `gs`.

7.2.5.10 int ppl_new_C_Polyhedron_recycle_Generator_System (ppl_Polyhedron_t * *pph*, ppl_-Generator_System_t *gs*)

Builds a new C polyhedron recycling the system of generators `gs` and writes a 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.

7.2.5.11 int ppl_new_NNC_Polyhedron_from_Generator_System (ppl_Polyhedron_t * *pph*, ppl_const_Generator_System_t *gs*)

Builds a new NNC polyhedron from the system of generators `gs` and writes a handle for the newly created polyhedron at address `pph`.

The new polyhedron will inherit the space dimension of `gs`.

7.2.5.12 int ppl_new_NNC_Polyhedron_recycle_Generator_System (ppl_Polyhedron_t * *pph*, ppl_Generator_System_t *gs*)

Builds a new NNC polyhedron recycling the system of generators `gs` and writes a 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.

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

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 space 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 space 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.

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

Builds a new NNC 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 space 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 space 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.

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

Checks the relation between the polyhedron ph and 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.

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

Checks the relation between the polyhedron `ph` and 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`.

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

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

Parameters:

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

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;

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 space 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 space 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.

7.2.5.18 int ppl_Polyhedron_maximize (ppl_const_Polyhedron_t *ph*, ppl_const_Linear_Expression_t *le*, ppl_Coefficient_t *sup_n*, ppl_Coefficient_t *sup_d*, int ∗ *pmaximum*, ppl_Generator_t *point*)

Returns a positive integer if `ph` is not empty and `le` is bounded from above in `ph`, in which case the supremum value and a point where `le` reaches it are computed.

Parameters:

ph The polyhedron constraining `le`;

le The linear expression to be maximized subject to `ph`;

sup_n Will be assigned the numerator of the supremum value;

sup_d Will be assigned the denominator of the supremum value;

pmaximum Will store 1 in this location if the supremum is also the maximum, will store 0 otherwise;

point Will be assigned the point or closure point where `le` reaches the extremum value.

If `ph` is empty or `le` is not bounded from above, 0 will be returned and `sup_n`, `sup_d`, `*pmaximum` and `point` will be left untouched.

7.2.5.19 int ppl_Polyhedron_minimize (ppl_const_Polyhedron_t *ph*, ppl_const_Linear_-Expression_t *le*, ppl_Coefficient_t *inf_n*, ppl_Coefficient_t *inf_d*, int ∗ *pminimum*, ppl_Generator_t *point*)

Returns a positive integer if `ph` is not empty and `le` is bounded from below in `ph`, in which case the infimum value and a point where `le` reaches it are computed.

Parameters:

- ***ph*** The polyhedron constraining `le`;
- ***le*** The linear expression to be minimized subject to `ph`;
- ***inf_n*** Will be assigned the numerator of the infimum value;
- ***inf_d*** Will be assigned the denominator of the infimum value;
- ***pminimum*** Will store 1 in this location if the infimum is also the minimum, will store 0 otherwise;
- ***point*** Will be assigned the point or closure point where `le` reaches the extremum value.

If `ph` is empty or `le` is not bounded from below, 0 will be returned and `sup_n`, `sup_d`, `*pmaximum` and `point` will be left untouched.

7.2.5.20 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; returns 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.

7.2.5.21 int ppl_Polyhedron_add_recycled_constraints (ppl_Polyhedron_t *ph*, ppl_Constraint_-System_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.

7.2.5.22 int ppl_Polyhedron_add_recycled_constraints_and_minimize (ppl_Polyhedron_t *ph*, ppl_-Constraint_System_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.

7.2.5.23 int ppl_Polyhedron_add_recycled_generators (ppl_Polyhedron_t *ph*, ppl_Generator_System_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.

7.2.5.24 int ppl_Polyhedron_add_recycled_generators_and_minimize (ppl_Polyhedron_t *ph*, ppl_Generator_System_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.

7.2.5.25 int ppl_Polyhedron_affine_image (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_Linear_Expression_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.

7.2.5.26 int ppl_Polyhedron_affine_preimage (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_Linear_Expression_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.

7.2.5.27 int ppl_Polyhedron_bounded_affine_image (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_Linear_Expression_t *lb*, ppl_const_Linear_Expression_t *ub*, ppl_const_Coefficient_t *d*)

Assigns to `ph` the image of `ph` with respect to the generalized affine transfer relation $\frac{\mathrm{lb}}{\mathrm{d}} \leq \mathrm{var}' \leq \frac{\mathrm{ub}}{\mathrm{d}}$.

Parameters:

ph The polyhedron that is transformed;

var The variable bounded by the generalized affine transfer relation;

lb The numerator of the lower bounding affine expression;

ub The numerator of the upper bounding affine expression;

d The (common) denominator of the lower and upper bounding affine expressions.

7.2.5.28 int ppl_Polyhedron_bounded_affine_preimage (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, ppl_const_Linear_Expression_t *lb*, ppl_const_Linear_Expression_t *ub*, ppl_const_Coefficient_t *d*)

Assigns to `ph` the preimage of `ph` with respect to the generalized affine transfer relation $\frac{\mathrm{lb}}{\mathrm{d}} \leq \mathrm{var}' \leq \frac{\mathrm{ub}}{\mathrm{d}}$.

Parameters:

ph The polyhedron that is transformed;

var The variable bounded by the generalized affine transfer relation;

lb The numerator of the lower bounding affine expression;

ub The numerator of the upper bounding affine expression;

d The (common) denominator of the lower and upper bounding affine expressions.

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

Assigns to `ph` the image of `ph` with respect to the generalized affine transfer relation $\mathrm{var}' \bowtie \frac{\mathrm{le}}{\mathrm{d}}$, 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 relation;

relsym The relation symbol;

le The numerator of the right hand side affine expression;

d The denominator of the right hand side affine expression.

7.2.5.30 int ppl_Polyhedron_generalized_affine_preimage (ppl_Polyhedron_t *ph*, ppl_dimension_type *var*, enum ppl_enum_Constraint_Type *relsym*, ppl_const_Linear_Expression_t *le*, ppl_const_Coefficient_t *d*)

Assigns to `ph` the preimage of `ph` with respect to the generalized affine transfer relation $\mathrm{var}' \bowtie \frac{\mathrm{le}}{\mathrm{d}}$, 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 relation;

relsym The relation symbol;

le The numerator of the right hand side affine expression;

d The denominator of the right hand side affine expression.

7.2.5.31 int ppl_Polyhedron_generalized_affine_image_lhs_rhs (ppl_Polyhedron_t *ph*, ppl_const_Linear_Expression_t *lhs*, enum ppl_enum_Constraint_Type *relsym*, ppl_const_Linear_Expression_t *rhs*)

Assigns to `ph` the image of `ph` with respect to the generalized affine transfer relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, 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.

7.2.5.32 int ppl_Polyhedron_generalized_affine_preimage_lhs_rhs (ppl_Polyhedron_t *ph*, ppl_const_Linear_Expression_t *lhs*, enum ppl_enum_Constraint_Type *relsym*, ppl_const_Linear_Expression_t *rhs*)

Assigns to `ph` the preimage of `ph` with respect to the generalized affine transfer relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, 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.

7.2.5.33 int ppl_Polyhedron_map_space_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.

7.2.5.34 int ppl_LP_Problem_solve (ppl_const_LP_Problem_t *lp*)

Solves the LP problem `lp`, returning an exit status.

Returns:

> `PPL_LP_PROBLEM_STATUS_UNFEASIBLE` if the LP problem is not satisfiable; `PPL_LP_-PROBLEM_STATUS_UNBOUNDED` if the LP problem is satisfiable but there is no finite bound to the value of the objective function; `PPL_LP_PROBLEM_STATUS_OPTIMIZED` if the LP problem admits an optimal solution.

7.2.5.35 int ppl_LP_Problem_evaluate_objective_function (ppl_const_LP_Problem_t *lp*, ppl_const_Generator_t *g*, ppl_Coefficient_t *num*, ppl_Coefficient_t *den*)

Evaluates the objective function of `lp` on point `g`.

Parameters:

lp The LP problem defining the objective function;

g The generator on which the objective function will be evaluated;

num Will be assigned the numerator of the objective function value;

den Will be assigned the denominator of the objective function value;

7.2.5.36 int ppl_LP_Problem_optimal_value (ppl_const_LP_Problem_t *lp*, ppl_Coefficient_t *num*, ppl_Coefficient_t *den*)

Returns the optimal value for `lp`.

Parameters:

lp The LP problem;

num Will be assigned the numerator of the optimal value;

den Will be assigned the denominator of the optimal value.

7.3 Prolog Language Interface

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.

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

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 The Main Features, 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.

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

- The Prolog interface files are all installed in the directory `prefix/lib/ppl`. Since this includes shared and dynamically loaded libraries, you must make your dynamic linker/loader aware of this fact. If you use a GNU/Linux system, try the commands `man ld.so` and `man ldconfig` for more information.

- The Prolog interface to the PPL is initialized and finalized by the predicates `ppl_initialize/0` and `ppl_finalize/0`. 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.

- 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 predicates such as

  ```
  ppl_new_C_Polyhedron_from_space_dimension/3,
  ppl_new_C_Polyhedron_from_C_Polyhedron/2,
  ppl_new_C_Polyhedron_from_constraints/2,
  ppl_new_C_Polyhedron_from_generators/2,
  ppl_new_C_Polyhedron_from_bounding_box/2.
  ```

  These predicates will create or copy a PPL polyhedron and construct a valid handle for referencing it. The last 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.

- Any application using the PPL should make sure that only the intended version(s) of the library are ever used. Predicates

  ```
  ppl_version_major/1,
  ppl_version_minor/1,
  ppl_version_revision/1,
  ppl_version_beta/1,
  ppl_version/1,
  ppl_banner.
  ```

  allow run-time checking of information about the version being used.

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

`ppl_version_major(?C_int)`

`ppl_version_minor(?C_int)`

`ppl_version_revision(?C_int)`

`ppl_version_beta(?C_int)`

`ppl_version(?Atom)`

`ppl_banner(?Atom)`

`ppl_max_space_dimension(?Dimension_Type)`

`ppl_Coefficient_is_bounded`

`ppl_Coefficient_max(?Coefficient)`

`ppl_Coefficient_min(?Coefficient)`

`ppl_initialize`

`ppl_finalize`

`ppl_set_timeout_exception_atom(+Atom)`

`ppl_set_timeout(+C_unsigned)`

`ppl_reset_timeout`

`ppl_new_C_Polyhedron_from_space_dimension(+Dimension_Type, +Universe_-or_Empty, -Handle)`

`ppl_new_NNC_Polyhedron_from_space_dimension(+Dimension_Type, +Universe_or_Empty, -Handle)`

`ppl_new_C_Polyhedron_from_C_Polyhedron(+Handle_1, -Handle_2)`

`ppl_new_C_Polyhedron_from_NNC_Polyhedron(+Handle_1, -Handle_2)`

`ppl_new_NNC_Polyhedron_from_C_Polyhedron(+Handle_1, -Handle_2)`

`ppl_new_NNC_Polyhedron_from_NNC_Polyhedron(+Handle_1, -Handle_2)`

`ppl_new_C_Polyhedron_from_constraints(+Constraint_System, -Handle)`

`ppl_new_NNC_Polyhedron_from_constraints(+Constraint_System, -Handle)`

`ppl_new_C_Polyhedron_from_generators(+Generator_System, -Handle)`

`ppl_new_NNC_Polyhedron_from_generators(+Generator_System, -Handle)`

`ppl_new_C_Polyhedron_from_bounding_box(+Box, -Handle)`

`ppl_new_NNC_Polyhedron_from_bounding_box(+Box, -Handle)` `ppl_Polyhedron_swap(+Handle_1, +Handle_2)`

`ppl_delete_Polyhedron(+Handle)`

`ppl_Polyhedron_space_dimension(+Handle, ?Dimension_Type)`

`ppl_Polyhedron_affine_dimension(+Handle, ?Dimension_Type)`

`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, +Lin_Expr)`

`ppl_Polyhedron_bounds_from_below(+Handle, +Lin_Expr)`

`ppl_Polyhedron_maximize(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean)`

`ppl_Polyhedron_maximize_with_point(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean, ?Point)`

`ppl_Polyhedron_minimize(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean)`

`ppl_Polyhedron_minimize_with_point(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean, ?Point)`

`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, +Lin_Expr, +Coefficient)

ppl_Polyhedron_affine_preimage(+Handle, +PPL_Var, +Lin_Expr, +Coefficient)

ppl_Polyhedron_bounded_affine_image(+Handle, +PPL_Var, +Lin_Expr_1, +Lin_Expr_2, +Coefficient)

ppl_Polyhedron_generalized_affine_image(+Handle, +PPL_Var, +Relation_-Symbol, +Lin_Expr, +Coefficient)

ppl_Polyhedron_generalized_affine_image_lhs_rhs(+Handle, +Lin_Expr_1, +Relation_Symbol, +Lin_Expr_2)

ppl_Polyhedron_time_elapse_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_BHRZ03_widening_assign_with_tokens(+Handle_1, +Handle_-2, +C_unsigned_1, ?C_unsigned_2)

ppl_Polyhedron_BHRZ03_widening_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_limited_BHRZ03_extrapolation_assign_with_-tokens(+Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_-unsigned_2)

ppl_Polyhedron_limited_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)

ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign_with_-tokens(+Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_-unsigned_2)

ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)

ppl_Polyhedron_H79_widening_assign_with_tokens(+Handle_1, +Handle_2, +C_unsigned_1, ?C_unsigned_2)

ppl_Polyhedron_H79_widening_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_limited_H79_extrapolation_assign_with_tokens(+Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_2)

ppl_Polyhedron_limited_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)

ppl_Polyhedron_bounded_H79_extrapolation_assign_with_tokens(+Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_2) ppl_Polyhedron_bounded_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)

ppl_Polyhedron_topological_closure_assign(+Handle)

ppl_Polyhedron_add_space_dimensions_and_embed(+Handle, +Dimension_-Type)

ppl_Polyhedron_add_space_dimensions_and_project(+Handle, +Dimension_-Type)

ppl_Polyhedron_concatenate_assign(+Handle_1, +Handle_2)

ppl_Polyhedron_remove_space_dimensions(+Handle, +List_of_PPL_Vars)

ppl_Polyhedron_remove_higher_space_dimensions(+Handle, +Dimension_-Type))

ppl_Polyhedron_expand_space_dimension(+Handle, +PPL_Var, +Dimension_-Type))

ppl_Polyhedron_fold_space_dimensions(+Handle, +List_of_PPL_Vars, +PPL_Var))

ppl_Polyhedron_map_space_dimensions(+Handle, +P_Func))

ppl_new_LP_Problem_trivial(-Handle)

ppl_new_LP_Problem(+Constraint_System, +Lin_Expr, +Optimization_Mode, -Handle)

ppl_new_LP_Problem_from_LP_Problem(+Handle_1, -Handle_2)

ppl_LP_Problem_swap(+Handle_1, +Handle_2)

ppl_delete_LP_Problem(+Handle)

ppl_LP_Problem_space_dimension(+Handle, ?Dimension_Type)

ppl_LP_Problem_constraints(+Handle, -Constraint_System)

ppl_LP_Problem_objective_function(+Handle, -Lin_Expr)

ppl_LP_Problem_optimization_mode(+Handle, ?Optimization_Mode)

ppl_LP_Problem_clear(+Handle)

ppl_LP_Problem_add_constraint(+Handle, +Constraint)

ppl_LP_Problem_add_constraints(+Handle, +Constraint_System)

ppl_LP_Problem_set_objective_function(+Handle, +Lin_Expr)

ppl_LP_Problem_set_optimization_mode(+Handle, +Optimization_Mode)

ppl_LP_Problem_is_satisfiable(+Handle)

ppl_LP_Problem_solve(+Handle, ?LP_Problem_Status)

ppl_LP_Problem_feasible_point(+Handle, -Generator)

ppl_LP_Problem_optimizing_point(+Handle, -Generator)

ppl_LP_Problem_optimal_value(+Handle, ?Coefficient_1, ?Coefficient_2)

ppl_LP_Problem_evaluate_objective_function(+Handle, +Generator, ?Coefficient_1, ?Coefficient_2)

ppl_LP_Problem_OK(+Handle)

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

```
Number      --> unsigned integer       ranging from 0 to an upper bound
                                       depending on the actual Prolog system.

C_int       --> Number | - Number      C integer

C_unsigned  --> Number                 C unsigned integer

Coefficient --> Number                 used in linear expressions;
                                       the upper bound will depend on how
                                       the PPL has been configured

Dimension_Type
            --> Number                 used for the number of affine and
                                       space dimensions and the names of
                                       the dimensions;
                                       the upper bound will depend on
                                       the maximum number of dimensions
                                       allowed by the PPL
                                       (see ppl_max_space_dimensions/1)

Boolean     --> true | false

Handle      --> Prolog term            used to identify a Polyhedron

Topology    --> c | nnc                Polyhedral kind;
                                       c is closed and nnc is NNC

VarId       --> Dimension_Type         variable identifier

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

Lin_Expr    --> PPL_Var                PPL variable
            | Coefficient
            | Lin_Expr                 unary plus
            | - Lin_Expr               unary minus
            | Lin_Expr + Lin_Expr      addition
            | Lin_Expr - Lin_Expr      subtraction
            | Coefficient * Lin_Expr   multiplication
            | Lin_Expr * Coefficient   multiplication

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

Constraint  --> Lin_Expr Relation_Symbol Lin_Expr
                                       constraint

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

Generator_Denominator --> Coefficient  must be non-zero
            | Coefficient
            | - Coefficient

Generator   --> point(Lin_Expr)        point
            | point(Lin_Expr, Generator_Denominator)
                                       point
            | closure_point(Lin_Expr)  closure point
            | closure_point(Lin_Expr, Generator_Denominator)
                                       closure point
```

```
           | ray(Lin_Expr)              ray
           | line(Lin_Expr)             line

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

Atom        --> Prolog atom

Universe_or_Empty                       polyhedron
            --> universe
           | empty

Poly_Relation                           polyhedron relation:
            --> is_disjoint             with a constraint
           | strictly_intersects        with a constraint
           | is_included                with a constraint
           | saturates                  with a constraint
           | subsumes                   with a generator

Poly_Relation_List                      list of polyhedron relations
            --> []
           | [Poly_Relation | Poly_Relation_List]

Complexity  --> polynomial | simplex | any

Rational_Numerator
            --> Coefficient | - Coefficient

Rational_Denominator
            --> Coefficient             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].

Optimization_Mode
            --> max | min

LP_Problem_Status
            --> unfeasible | unbounded | optimized
```

Below is a short description of each of the interface predicates. For full definitions of terminology used here, see Sections The Main Features, Convex Polyhedra, Representations of Convex Polyhedra and Operations on Convex Polyhedra of this manual.

`ppl_version_major(?C_int)` Unifies `C_int` with the major number of the PPL version.

`ppl_version_minor(?C_int)` Unifies `C_int` with the minor number of the PPL version.

`ppl_version_revision(?C_int)` Unifies `C_int` with the revision number of the PPL version.

`ppl_version_beta(?C_int)` Unifies `C_int` with the beta number of the PPL version.

`ppl_version(?Atom)` Unifies `Atom` with the PPL version.

`ppl_banner(?Atom)` Unifies `Atom` with 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.

`ppl_Coefficient_is_bounded` Succeeds if and only if the Coefficients in the C++ interface are bounded.

`ppl_Coefficient_max(Max)` If the Coefficients in the C++ interface are bounded, then the maximum coefficient the C++ interface can handle is unified with `Max`. If the Prolog system cannot handle this coefficient, then an exception is thrown. It fails if the Coefficients in the C++ interface are unbounded.

`ppl_Coefficient_min(Min)` If the Coefficients in the C++ interface are bounded, then the minimum coefficient the C++ interface can handle is unified with `Min`. If the Prolog system cannot handle this coefficient, then an exception is thrown. It fails if the Coefficients in the C++ interface are unbounded.

`ppl_max_space_dimension(?Dimension_Type)` Unifies `Dimension_Type` with the maximum space dimension this library can handle.

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

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

`ppl_new_C_Polyhedron_from_space_dimension(+Dimension_Type, +Universe_or_Empty, -Handle)` Creates a C polyhedron $\mathcal{P}$ with `Dimension_Type` dimensions; it is empty or the universe polyhedron depending on whether `Atom` is `empty` or `universe`, respectively. `Handle` is unified with the handle for $\mathcal{P}$. Thus the query

```
?- ppl_new_C_Polyhedron_from_space_dimension(3, universe, 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_NNC_Polyhedron_from_space_dimension(+Dimension_Type, +Universe_or_Empty, -Handle)` Creates an NNC polyhedron $\mathcal{P}$ with `Dimension_Type` dimensions; it is empty or the universe polyhedron depending on whether `Atom` is `empty` or `universe`, respectively. `Handle` is unified with the handle for $\mathcal{P}$. Thus the query

```
?- ppl_new_NNC_Polyhedron_from_space_dimension(3, empty, X).
```

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

`ppl_new_C_Polyhedron_from_C_Polyhedron(+Handle_1, -Handle_2)` If `Handle_1` refers to a C polyhedron $\mathcal{P}_1$, then this creates a copy $\mathcal{P}_2$ of $\mathcal{P}_1$. `Handle_2` is unified with the handle for $\mathcal{P}_2$.

`ppl_new_C_Polyhedron_from_NNC_Polyhedron(+Handle_1, -Handle_2)` If `Handle_1` refers to an NNC polyhedron $\mathcal{P}_1$, then this creates a copy of $\mathcal{P}_1$ as a C polyhedron $\mathcal{P}_2$. `Handle_2` is unified with the handle for $\mathcal{P}_2$. Thus the query

```
?- ppl_new_NNC_Polyhedron_from_space_dimension(3, empty, X),
   ppl_new_C_Polyhedron_from_NNC_Polyhedron(X, Y).
```

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

When using `ppl_new_C_Polyhedron_from_NNC_Polyhedron/2`, care must be taken that the source polyhedron referenced by `Handle_1` is topologically closed.

`ppl_new_NNC_Polyhedron_from_C_Polyhedron(+Handle_1, -Handle_2)` If `Handle_1` refers to a C polyhedron $\mathcal{P}_1$, then this creates a copy of $\mathcal{P}_1$ as an NNC polyhedron $\mathcal{P}_2$. `Handle_2` is unified with the handle for $\mathcal{P}_2$.

`ppl_new_NNC_Polyhedron_from_NNC_Polyhedron(+Handle_1, -Handle_2)` If `Handle_1` refers to a NNC polyhedron $\mathcal{P}_1$, then this creates a copy $\mathcal{P}_2$ of $\mathcal{P}_1$. `Handle_2` is unified with the handle for $\mathcal{P}_2$.

`ppl_new_C_Polyhedron_from_constraints(+Constraint_System, -Handle)` Creates a C polyhedron $\mathcal{P}$ represented by `Constraint_System`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_NNC_Polyhedron_from_constraints(+Constraint_System, -Handle)` Creates an NNC polyhedron $\mathcal{P}$ represented by `Constraint_System`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_C_Polyhedron_from_generators(+Generator_System, -Handle)` Creates a C polyhedron $\mathcal{P}$ represented by `Generator_System`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_NNC_Polyhedron_from_generators(+Generator_System, -Handle)` Creates an NNC polyhedron $\mathcal{P}$ represented by `Generator_System`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_new_C_Polyhedron_from_bounding_box(+Box, -Handle)` Creates a C polyhedron $\mathcal{P}$ represented by `Box`. `Handle` is unified with the handle for $\mathcal{P}$. There must be no bounds of the form `o(Rational)` in an interval in `Box`.

`ppl_new_NNC_Polyhedron_from_bounding_box(+Box, -Handle)` Creates an NNC polyhedron $\mathcal{P}$ represented by `Box`. `Handle` is unified with the handle for $\mathcal{P}$.

`ppl_Polyhedron_swap(+Handle_1, +Handle_2)` Swaps the polyhedron referenced by `Handle_1` with the one referenced by `Handle_2`. 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, ?Dimension_Type)` Unifies the dimension of the vector space in which the polyhedron referenced by `Handle` is embedded with `Dimension_Type`.

`ppl_Polyhedron_affine_dimension(+Handle, ?Dimension_Type)` Unifies the actual dimension of the polyhedron referenced by `Handle` with `Dimension_Type`.

`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, ?Poly_-Relation_List) Unifies Poly_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 specifying the relation_with operations.

ppl_Polyhedron_relation_with_generator(+Handle, +Generator, ?Poly_-Relation_List) Unifies Poly_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 specifying the relation_with operations.

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_NNC_Polyhedron_from_constraints([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_NNC_Polyhedron_from_constraints([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, +Lin_Expr) Succeeds if and only if Lin_Expr is bounded from above in the polyhedron referenced by Handle.

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

`ppl_Polyhedron_maximize(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean)` Succeeds if and only if the polyhedron P referenced by `Handle` is not empty and `Lin_Expr` is bounded from above in P.

`Coefficient_1` is unified with the numerator of the supremum value and `Coefficient_2` with the denominator of the supremum value. If the supremum is also the maximum, `Boolean` is unified with the atom `true` and, otherwise, unified with the atom `false`.

`ppl_Polyhedron_maximize_with_point(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean, ?Point)` Succeeds if and only if the polyhedron P referenced by `Handle` is not empty and `Lin_Expr` is bounded from above in P.

`Coefficient_1` is unified with the numerator of the supremum value, `Coefficient_2` with the denominator of the supremum value, and `Point` with a point or closure point where `Lin_Expr` reaches this value. If the supremum is also the maximum, `Boolean` is unified with the atom `true` and, otherwise, unified with the atom `false`.

`ppl_Polyhedron_minimize(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean)` Succeeds if and only if the polyhedron P referenced by `Handle` is not empty and `Lin_Expr` is bounded from below in P.

`Coefficient_1` is unified with the numerator of the infimum value and `Coefficient_2` with the denominator of the infimum value. If the infimum is also the minimum, `Boolean` is unified with the atom `true` and, otherwise, unified with the atom `false`.

`ppl_Polyhedron_minimize_with_point(+Handle, +Lin_Expr, ?Coefficient_1, ?Coefficient_2, ?Boolean, ?Point)` Succeeds if and only if the polyhedron P referenced by `Handle` is not empty and `Lin_Expr` is bounded from below in P.

`Coefficient_1` is unified with the numerator of the infimum value, `Coefficient_2` with the denominator of the infimum value, and `Point` with a point or closure point where `Lin_Expr` reaches this value. If the infimum is also the minimum, `Boolean` is unified with the atom `true` and, otherwise, unified with the atom `false`.

`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_2` is included in or equal to the polyhedron referenced by `Handle_1`.

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

`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_C_Polyhedron_from_space_dimension(3, universe, 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_C_Polyhedron_from_space_dimension(3, universe, 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_C_Polyhedron_from_space_dimension(2, universe, 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_C_Polyhedron_from_space_dimension(2, universe, 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_C_Polyhedron_from_space_dimension(2, universe, X),
   ppl_Polyhedron_add_constraints_and_minimize(X,
     [4*A + B >= 3, A = 0, B =< 0]),
   ppl_Polyhedron_get_constraints(X, CS).
```

`ppl_C_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 Section Generators Representation). 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_C_Polyhedron_from_space_dimension(3, empty, 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_C_Polyhedron_from_space_dimension(3, empty, 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 polyhedron 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 polyhedron 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, +Lin_Expr, +Coefficient)` Transforms the polyhedron referenced by `Handle` assigning the affine expression `Lin_Expr/Coefficient` to `PPL_Var`.

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

`ppl_Polyhedron_bounded_affine_image(+Handle, +PPL_Var, +Lin_Expr_1, +Lin_Expr_2, +Coefficient)` Transforms the polyhedron referenced by `Handle` assigning the image with respect to the transfer relation `Lin_Expr_1/Coefficient <= PPL_Var <= Lin_Expr_2/Coefficient`.

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

`ppl_Polyhedron_generalized_affine_image_lhs_rhs(+Handle, +Lin_Expr_1, +Relation_Symbol +Lin_Expr_2)` Transforms the polyhedron referenced by `Handle` assigning the generalized affine image with respect to the transfer function `Lin_Expr_1 Relation_Symbol Lin_Expr_2`.

`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 polyhedron $\mathcal{Q}$ referenced by `Handle_2`.

`ppl_Polyhedron_BHRZ03_widening_assign(+Handle_1, +Handle_2)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the BHRZ03-widening of $\mathcal{P}_1$ with $\mathcal{P}_2$.

`ppl_Polyhedron_BHRZ03_widening_assign_with_tokens(+Handle_1, +Handle_-2, +C_unsigned_1, ?C_unsigned_2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the BHRZ03-widening of $\mathcal{P}_1$ with $\mathcal{P}_2$, Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$ and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

`ppl_Polyhedron_limited_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the BHRZ03-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$ improved by enforcing the constraints in `Constraint_System`.

`ppl_Polyhedron_limited_BHRZ03_extrapolation_assign_with_tokens( +Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_-2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the BHRZ03-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$, improved by enforcing those constraints in `Constraint_System`.

Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_-unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$ and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the BHRZ03-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$ improved by enforcing the constraints in `Constraint_System` together with all constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by every point in $\mathcal{P}_1$.

`ppl_Polyhedron_bounded_BHRZ03_extrapolation_assign_with_tokens( +Handle_1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_-2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the BHRZ03-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$, improved by enforcing those constraints in `Constraint_System` together with all constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by every point in $\mathcal{P}_1$.

Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_-unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$ and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

`ppl_Polyhedron_H79_widening_assign(+Handle_1, +Handle_2)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the H79-widening of $\mathcal{P}_1$ with $\mathcal{P}_2$.

`ppl_Polyhedron_H79_widening_assign_with_tokens(+Handle_1, +Handle_-2, +C_unsigned_1, ?C_unsigned_2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the H79-widening of $\mathcal{P}_1$ with $\mathcal{P}_2$,

Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_-unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

`ppl_Polyhedron_limited_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the H79-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$ improved by enforcing the constraints in `Constraint_System`.

`ppl_Polyhedron_limited_H79_extrapolation_assign_with_tokens( +Handle_-1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the H79-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$, improved by enforcing those constraints in `Constraint_System`.

Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_-unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$ and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

`ppl_Polyhedron_bounded_H79_extrapolation_assign(+Handle_1, +Handle_2, +Constraint_System)` If the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, then `Handle_1` will refer to the H79-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$ improved by enforcing the constraints in `Constraint_System` together with all constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by every point in $\mathcal{P}_1$.

`ppl_Polyhedron_bounded_H79_extrapolation_assign_with_tokens( +Handle_-1, +Handle_2, +Constraint_System, +C_unsigned_1, ?C_unsigned_2)` It is assumed that the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` contains the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`; let $\mathcal{P}$ denote the H79-extrapolation of $\mathcal{P}_1$ with $\mathcal{P}_2$, improved by enforcing those constraints in `Constraint_System` together with all constraints of the form $\pm x \leq r$ and $\pm x < r$ that are satisfied by every point in $\mathcal{P}_1$.

Assuming that the quantity t_1 given by `C_unsigned_1` is the number of tokens available, `C_-unsigned_2` will be unified with the number of tokens t_2 remaining at the end of the operation.

If $t_1 > 0$, then the polyhedron referenced by `Handle_1` will remain as $\mathcal{P}_1$ and either $t_2 = t_1 - 1$ or $t_2 = t_1$ depending on whether or not $\mathcal{P}$ is different from $\mathcal{P}_1$ itself.

If $t_1 = 0$, then $t_2 = 0$ and the polyhedron referenced by `Handle_1` is updated to $\mathcal{P}$.

In all cases, the polyhedron referenced by `Handle_2` will remain unchanged as $\mathcal{P}_2$.

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

`ppl_Polyhedron_add_space_dimensions_and_embed( +Handle, +Dimension_-Type)` Embeds the polyhedron referenced by `Handle` in a space that is enlarged by `Dimension_-Type` dimensions, E.g.,

```
?- ppl_new_C_Polyhedron_from_space_dimension(0, empty, X),
   ppl_Polyhedron_add_space_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_concatenate_assign(+Handle_1, +Handle_2)` Updates the polyhedron $\mathcal{P}_1$ referenced by `Handle_1` by first embedding $\mathcal{P}_1$ in a new space enlarged by the space dimensions of the polyhedron $\mathcal{P}_2$ referenced by `Handle_2`, and then adds to its system of constraints a renamed-apart version of the constraints of $\mathcal{P}_2$.

E.g.,

```
?- ppl_new_NNC_Polyhedron_from_space_dimension(2, universe, X),
   A = '$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   D = '$VAR'(3), E = '$VAR'(4),
   ppl_new_NNC_Polyhedron_from_constraints([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_add_space_dimensions_and_project(+Handle, +Dimension_-Type)` Projects the polyhedron referenced by `Handle` onto a space that is enlarged by `Dimension_-Type` dimensions, E.g.,

```
?- ppl_new_C_Polyhedron_from_space_dimension(0, empty, X),
   ppl_Polyhedron_add_space_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_remove_space_dimensions(+Handle, +List_of_PPL_Vars)` Removes the space 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_C_Polyhedron_from_space_dimension(3, empty, X),
   A='$VAR'(0), B = '$VAR'(1), C = '$VAR'(2),
   ppl_Polyhedron_remove_space_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_space_dimensions(+Handle, +Dimension_-Type))` Projects the polyhedron referenced to by `Handle` onto the first `Dimension_Type` dimension. E.g.,

```
?- ppl_new_C_Polyhedron_from_space_dimension(5, empty, X),
   ppl_Polyhedron_remove_higher_space_dimensions(X, 3),
   ppl_Polyhedron_space_dimension(X, K).
```

`ppl_Polyhedron_expand_space_dimension(+Handle, +PPL_Var, +Dimension_Type))` `Dimension_Type` copies of the space dimension referenced by `PPL_Var` are added to the polyhedron referenced to by `Handle`.

`ppl_Polyhedron_fold_space_dimensions(+Handle, +List_of_PPL_Vars, +PPL_Var))` The space dimensions referenced by the PPL variables in list `List_of_PPL_Vars` are folded into the dimension referenced by `PPL_Var` and removed. The result is undefined if `List_of_PPL_Vars` does not have the properties described in Section fold_space_dimensions.

`ppl_Polyhedron_map_space_dimensions(+Handle, +P_Func))` Maps the space 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 Section specifying the map_space_dimensions operator.

`ppl_new_LP_Problem_trivial(-Handle)` Creates an LP Problem LP with the feasible region the 0-dimensional universe, objective function 0 and optimization mode `max`. `Handle` is unified with the handle for LP.

`ppl_new_LP_Problem(+Constraint_System, +Lin_Expr, +Optimization_Mode, -Handle)` Creates an LP Problem LP with the feasible region represented by `Constraint_System`, objective function `Lin_Expr` and optimization mode `Optimization_Mode`. `Handle` is unified with the handle for LP.

`ppl_LP_Problem_swap(+Handle_1, +Handle_2)` Swaps the LP Problem referenced by `Handle_1` with the one referenced by `Handle_2`.

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

`ppl_LP_Problem_space_dimension(+Handle, ?Dimension_Type)` Unifies the dimension of the vector space in which the LP Problem referenced by `Handle` is embedded with `Dimension_Type`.

`ppl_LP_Problem_constraints(+Handle, -Constraint_System)` Unifies `Constraint_System` with a list of the constraints in the constraints system representing the feasible region for the LP Problem referenced by `Handle`.

`ppl_LP_Problem_objective_function(+Handle, -Lin_Expr)` Unifies `Lin_Expr` with the objective function for the LP Problem referenced by `Handle`.

`ppl_LP_Problem_optimization_mode(+Handle, ?Optimization_Mode)` Unifies `Optimization_Mode` with the optimization mode for the LP Problem referenced by `Handle`.

`ppl_LP_Problem_clear(+Handle)` Resets the LP problem referenced by `Handle` to be the trivial problem with the feasible region the 0-dimensional universe, objective function 0 and optimization mode `max`.

`ppl_LP_Problem_add_constraint(+Handle, +Constraint)` Updates the LP Problem referenced by `Handle` so that the feasible region is represented by the original constraint system together with the constraint `Constraint`.

`ppl_LP_Problem_add_constraints(+Handle, +Constraint_System)` Updates the LP Problem referenced by `Handle` so that the feasible region is represented by the original constraint system together with all the constraints in `Constraint_System`.

`ppl_LP_Problem_set_objective_function(+Handle, +Lin_Expr)` Updates the LP Problem referenced by `Handle` so that the objective function is changed to `Lin_Expr`.

`ppl_LP_Problem_set_optimization_mode(+Handle, +Optimization_Mode)` Updates the LP Problem referenced by `Handle` so that the optimization mode is changed to `Optimization_Mode`.

`ppl_LP_Problem_is_satisfiable(+Handle)` Succeeds if and only if the LP Problem referenced by `Handle` is satisfiable.

`ppl_LP_Problem_solve(+Handle, ?LP_Problem_Status)` Solves the LP problem referenced by `Handle` and unifies `LP_Problem_Status` with: `unfeasible`, if the LP problem is not satisfiable; `unbounded`, if the LP problem is satisfiable but there is no finite bound to the value of the objective function; `optimized`, if the LP problem admits an optimal solution.

`ppl_LP_Problem_feasible_point(+Handle, ?Generator)` Unifies `Generator` with a feasible point for the LP problem referenced by `Handle`.

`ppl_LP_Problem_optimizing_point(+Handle, ?Generator)` Unifies `Generator` with an optimizing point for the LP problem referenced by `Handle`.

`ppl_LP_Problem_optimal_value(+Handle, ?Coefficient_1, ?Coefficient_2)` Unifies `Coefficient_1` and `Coefficient_2` with the numerator and denominator, respectively, for the optimal value for the LP problem referenced by `Handle`.

`ppl_LP_Problem_evaluate_objective_function(+Handle, +Generator, ?Coefficient_1, ?Coefficient_2)` Evaluates the objective function of the LP problem referenced by `Handle` at point `Generator`. `Coefficient_1` is unified with the numerator and `Coefficient_2` is unified with the denominator of the objective function value at `Generator`.

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

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.

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" ...
```

System-Dependent Features

CIAO Prolog The Ciao Prolog interface to the PPL is available both as “PPL enhanced” Ciao Prolog interpreter and as a library that can be linked to Ciao Prolog programs. Only Ciao Prolog versions 1.10 #5 and later are supported.

So that it can be used with the Ciao Prolog PPL interface, the Ciao Prolog installation must be configured with the `-disable-regs` option.

The `ppl_ciao` Executable If an appropriate version of Ciao Prolog is installed on the machine on which you compiled the library, the command `make install` will install the executable `ppl_ciao` in the directory `prefix/bin`. The `ppl_ciao` executable is simply the Ciao 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 Ciao Prolog Programs In order to allow linking Ciao Prolog programs to the PPL, the following files are installed in the directory `prefix/lib/ppl`: `ppl_ciao.pl` contains the required foreign declarations; `libppl_ciao.*` contain the executable code for the Ciao Prolog interface in various formats (static library, shared library, libtool library). If your Ciao 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

```
ciaoc -o myprog prefix/lib/ppl/ppl_ciao.pl ciao_pl_check.pl \
  -L '-Lprefix/lib/ppl -lppl_ciao -Lprefix/lib -lppl -lgmpxx -lgmp -lstdc++'
```

GNU Prolog The GNU Prolog interface to the PPL is available both as a “PPL enhanced” GNU Prolog interpreter and as a library that can be linked to GNU Prolog programs. The only GNU Prolog version that is known to work is a patched version of the “unstable version” tagged `20040608` (which unpacks to a directory called `gprolog-1.2.18`). The patch is contained in the `interfaces/Prolog/GNU/README` file of the PPL’s distribution.

So that it can be used with the GNU Prolog PPL interface (and, for that matter, with any foreign code), the GNU Prolog installation must be configured with the `-disable-regs` option.

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 is available both as a statically linked module or as a dynamically linked one. Only SICStus Prolog versions 3.9.0 and later are 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 to the PPL is available both as a statically linked module or as a dynamically linked one. Only SWI-Prolog version 5.6.0 and later versions are 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` and `ppl_finalize/0` automatically. 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 is available as a dynamically linked module. Only some CVS versions of XSB starting from 2 July 2005 are known to work. CVS versions starting from 11 November 2005 are known not to work.

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.xwam` 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 is available as a dynamically linked module. Only YAP versions following 5.1.0 and CVS HEAD versions starting from 4 January 2006 are supported. Notice that support for unbounded integers in YAP is young and may have errors that could affect programs using the PPL (see, e.g., `http://www.cs.unipr.it/pipermail/ppl-devel/2006-January/007780.html`).

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.

8 PPL Namespace Documentation

8.1 Parma_Polyhedra_Library Namespace Reference

The entire library is confined to this namespace.

Classes

- class Checked_Number

  A wrapper for numeric types implementing a given policy.

- class Throwable

  User objects the PPL can throw.

- struct From_Bounding_Box

  A tag class.

- struct From_Covering_Box

  A tag class.

- class Constraint_System

  A system of constraints.

- class Variable

  A dimension of the vector space.

- class Poly_Con_Relation

  The relation between a polyhedron and a constraint.

- class Generator_System

A system of generators.

- class Grid_Generator_System

  A system of grid generators.
- class Congruence_System

  A system of congruences.
- class Linear_Expression

  A linear expression.
- class Constraint

  A linear equality or inequality.
- class Generator

  A line, ray, point or closure point.
- class Grid_Generator

  A line, parameter or point.
- class Congruence

  A linear congruence.
- class LP_Problem

  A Linear Programming problem.
- class Poly_Gen_Relation

  The relation between a polyhedron and a generator.
- class BHRZ03_Certificate

  The convergence certificate for the BHRZ03 widening operator.
- class H79_Certificate

  A convergence certificate for the H79 widening operator.
- class Polyhedron

  The base class for convex polyhedra.
- class Grid_Certificate

  The convergence certificate for the Grid widening operator.
- class Grid

  A grid.
- class C_Polyhedron

  A closed convex polyhedron.
- class NNC_Polyhedron

  A not necessarily closed convex polyhedron.

- class BD_Shape

  A bounded difference shape.

- class Determinate

  Wraps a PPL class into a determinate constraint system interface.

- class Powerset

  The powerset construction on a base-level domain.

- class Polyhedra_Powerset

  The powerset construction instantiated on PPL polyhedra.

- class GMP_Integer

  Unbounded integers as provided by the GMP library.

Namespaces

- namespace IO_Operators

  All input/output operators are confined to this namespace.

Typedefs

- typedef size_t dimension_type

  An unsigned integral type for representing space dimensions.

- typedef size_t memory_size_type

  An unsigned integral type for representing memory size in bytes.

- typedef COEFFICIENT_TYPE Coefficient

  An alias for easily naming the type of PPL coefficients.

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

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

Enumerations

- enum Result { ,
  VC_NORMAL, V_LT, V_GT, V_EQ,
  V_NE, V_LE, V_GE, V_LGE,
  VC_MINUS_INFINITY, V_NEG_OVERFLOW, VC_PLUS_INFINITY, V_POS_OVERFLOW,
  VC_NAN, V_CVT_STR_UNK, V_DIV_ZERO, V_INF_ADD_INF,
  V_INF_DIV_INF, V_INF_MOD, V_INF_MUL_ZERO, V_INF_SUB_INF,
  V_MOD_ZERO, V_SQRT_NEG, V_UNKNOWN_NEG_OVERFLOW, V_UNKNOWN_POS_-OVERFLOW,
  V_UNORD_COMP }

Possible outcomes of a checked arithmetic computation.

- enum Rounding_Dir { ROUND_DOWN, ROUND_UP, ROUND_IGNORE , ROUND_NOT_-NEEDED }

  Rounding directions for arithmetic computations.

- enum Degenerate_Element { UNIVERSE, EMPTY }

  Kinds of degenerate abstract elements.

- enum Relation_Symbol {
  LESS_THAN, LESS_THAN_OR_EQUAL, EQUAL, GREATER_THAN_OR_EQUAL,
  GREATER_THAN }

  Relation symbols.

- enum Complexity_Class { POLYNOMIAL_COMPLEXITY, SIMPLEX_COMPLEXITY, ANY_-COMPLEXITY }

  Complexity pseudo-classes.

- enum Optimization_Mode { MINIMIZATION, MAXIMIZATION }

  Possible optimization modes.

- enum LP_Problem_Status { UNFEASIBLE_LP_PROBLEM, UNBOUNDED_LP_PROBLEM, OPTIMIZED_LP_PROBLEM }

  Possible outcomes of the LP_Problem solver.

Functions

- unsigned version_major ()

  Returns the major number of the PPL version.

- unsigned version_minor ()

  Returns the minor number of the PPL version.

- unsigned version_revision ()

  Returns the revision number of the PPL version.

- unsigned version_beta ()

  Returns the beta number of the PPL version.

- const char ∗ version ()

  Returns a character string containing the PPL version.

- const char ∗ banner ()

  Returns a character string containing the PPL banner.

- fpu_rounding_direction_type fpu_get_rounding_direction ()

  Returns the current FPU rounding direction.

- void fpu_set_rounding_direction (fpu_rounding_direction_type dir)

Sets the FPU rounding direction to `dir`.

- fpu_rounding_control_word_type fpu_save_rounding_direction (fpu_rounding_direction_type dir)

  Sets the FPU rounding direction to `dir` *and returns the rounding control word previously in use.*

- fpu_rounding_control_word_type fpu_save_rounding_direction_reset_inexact (fpu_rounding_-direction_type dir)

  Sets the FPU rounding direction to `dir`*, clears the* inexact computation *status, and returns the rounding control word previously in use.*

- void fpu_restore_rounding_direction (fpu_rounding_control_word_type w)

  Restores the FPU rounding rounding control word to `cw`.

- void fpu_reset_inexact ()

  Clears the inexact computation *status.*

- int fpu_check_inexact ()

  Queries the inexact computation *status.*

- Rounding_Dir inverse (Rounding_Dir dir)

  Returns the inverse rounding mode of `dir`, `ROUND_IGNORE` *being the inverse of itself.*

- unsigned rational_sqrt_precision_parameter ()

  Returns the precision parameter used for rational square root calculations.

- void set_rational_sqrt_precision_parameter (const unsigned p)

  Sets the precision parameter used for rational square root calculations.

- dimension_type not_a_dimension ()

  Returns a value that does not designate a valid dimension.

- dimension_type max_space_dimension ()

  Returns the maximum space dimension this library can handle.

Variables

- const Throwable ∗volatile abandon_expensive_computations

  A pointer to an exception object.

8.1.1 Detailed Description

The entire library is confined to this namespace.

8.1.2 Function Documentation

8.1.2.1 const char∗ Parma_Polyhedra_Library::banner ()

Returns a character string containing the PPL banner.

The banner provides 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.

8.1.2.2 int Parma_Polyhedra_Library::fpu_check_inexact () `[inline]`

Queries the *inexact computation* status.

Returns 0 if the computation was definitely exact, 1 if it was definitely inexact, -1 if definite exactness information is unavailable.

8.1.2.3 void Parma_Polyhedra_Library::set_rational_sqrt_precision_parameter (const unsigned *p*) `[inline]`

Sets the precision parameter used for rational square root calculations.

If `p` is less than or equal to `INT_MAX`, sets the precision parameter used for rational square root calculations to `p`.

Exceptions:

std::invalid_argument Thrown if `p` is greater than `INT_MAX`.

8.2 Parma_Polyhedra_Library::IO_Operators Namespace Reference

All input/output operators are confined to this namespace.

8.2.1 Detailed Description

All input/output operators are confined to 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 Constraint_System& 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<<;
}
```

8.3 std Namespace Reference

The standard C++ namespace.

8.3.1 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 function templates `swap()` and `iter_swap()` (25.2.2, [lib.alg.swap]), and for the class template `numeric_limits` (18.2.1, [lib.limits]).

Note:

The PPL provides the specializations of the class template `numeric_limits` not only for PPL-specific numeric types, but also for the GMP types `mpz_class` and `mpq_class`. These specializations will be removed as soon as they will be provided by the C++ interface of GMP.

9 PPL Class Documentation

9.1 Parma_Polyhedra_Library::BD_Shape< T > Class Template Reference

A bounded difference shape.

Public Types

- typedef T base_type

  The numeric base type upon which bounded differences are built.

- typedef N coefficient_type

  The (extended) numeric type of the inhomogeneous term of the inequalities defining a BDS.

Public Member Functions

Constructors, Assignment, Swap and Destructor

- BD_Shape (dimension_type num_dimensions=0, Degenerate_Element kind=UNIVERSE)

  Builds a universe or empty BDS of the specified space dimension.

- BD_Shape (const BD_Shape &y)

  Ordinary copy-constructor.

- template<typename U> BD_Shape (const BD_Shape< U > &y)

  Builds a conservative, upward approximation of `y`.

- BD_Shape (const Constraint_System &cs)

  Builds a BDS from the system of constraints `cs`.

- BD_Shape (const Generator_System &gs)

Builds a BDS from the system of generators `gs`*.*

- BD_Shape (const Polyhedron &ph, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a BDS from the polyhedron `ph`*.*

- BD_Shape & operator= (const BD_Shape &y)

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

- void swap (BD_Shape &y)

  Swaps `*this` *with* `y` *(*`*this` *and* `y` *can be dimension-incompatible).*

- ~BD_Shape ()

  Destructor.

Member Functions that Do Not Modify the BD_Shape

- dimension_type space_dimension () const

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

- dimension_type affine_dimension () const

  Returns 0*, if* `*this` *is empty; otherwise, returns the affine dimension of* `*this`*.*

- Constraint_System constraints () const

  Returns a system of constraints defining `*this`*.*

- Constraint_System minimized_constraints () const

  Returns a minimized system of constraints defining `*this`*.*

- bool contains (const BD_Shape &y) const

  Returns `true` *if and only if* `*this` *contains* `y`*.*

- bool strictly_contains (const BD_Shape &y) const

  Returns `true` *if and only if* `*this` *strictly contains* `y`*.*

- Poly_Con_Relation relation_with (const Constraint &c) const

  Returns the relations holding between `*this` *and the constraint* `c`*.*

- Poly_Gen_Relation relation_with (const Generator &g) const

  Returns the relations holding between `*this` *and the generator* `g`*.*

- bool is_empty () const

  Returns `true` *if and only if* `*this` *is an empty BDS.*

- bool is_universe () const

  Returns `true` *if and only if* `*this` *is a universe BDS.*

- bool OK () const

  Returns `true` *if and only if* `*this` *satisfies all its invariants.*

Space-Dimension Preserving Member Functions that May Modify the BD_Shape

- void add_constraint (const Constraint &c)

Adds a copy of constraint `c` *to the system of bounded differences defining* `*this`.

- bool add_constraint_and_minimize (const Constraint &c)

  Adds a copy of constraint `c` *to the system of bounded differences defining* `*this`.

- void add_constraints (const Constraint_System &cs)

  Adds the constraints in `cs` *to the system of bounded differences defining* `*this`.

- bool add_constraints_and_minimize (const Constraint_System &cs)

  Adds the constraints in `cs` *to the system of bounded differences defining* `*this`.

- void intersection_assign (const BD_Shape &y)

  Assigns to `*this` *the intersection of* `*this` *and* `y`.

- bool intersection_assign_and_minimize (const BD_Shape &y)

  Assigns to `*this` *the intersection of* `*this` *and* `y`.

- void bds_hull_assign (const BD_Shape &y)

  Assigns to `*this` *the smallest BDS containing the convex union of* `*this` *and* `y`.

- bool bds_hull_assign_and_minimize (const BD_Shape &y)

  Assigns to `*this` *the smallest BDS containing the convex union of* `*this` *and* `y`.

- void upper_bound_assign (const BD_Shape &y)

  Same as bds_hull_assign.

- bool bds_hull_assign_if_exact (const BD_Shape &y)

  If the bds-hull of `*this` *and* `y` *is exact, it is assigned to* `*this` *and* `true` *is returned, otherwise* `false` *is returned.*

- bool upper_bound_assign_if_exact (const BD_Shape &y)

  Same as bds_hull_assign_if_exact.

- void bds_difference_assign (const BD_Shape &y)

  Assigns to `*this` *the poly-difference of* `*this` *and* `y`.

- void difference_assign (const BD_Shape &y)

  Same as bds_difference_assign.

- void affine_image (Variable var, const Linear_Expression &expr, Coefficient_traits::const_-reference denominator=Coefficient_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 Linear_Expression &expr, Coefficient_traits::const_-reference denominator=Coefficient_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, Relation_Symbol relsym, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

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

- void generalized_affine_image (const Linear_Expression &lhs, Relation_Symbol relsym, const Linear_Expression &rhs)

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

- void generalized_affine_preimage (Variable var, Relation_Symbol relsym, const Linear_-Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

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

- void time_elapse_assign (const BD_Shape &y)

  Assigns to `*this` *the result of computing the time-elapse between* `*this` *and* `y`.

- void CC76_extrapolation_assign (const BD_Shape &y, unsigned *tp=0)

  Assigns to `*this` *the result of computing the CC76-extrapolation between* `*this` *and* `y`.

- template<typename Iterator> void CC76_extrapolation_assign (const BD_Shape &y, Iterator first, Iterator last, unsigned *tp=0)

  Assigns to `*this` *the result of computing the CC76-extrapolation between* `*this` *and* `y`.

- void BHMZ05_widening_assign (const BD_Shape &y, unsigned *tp=0)

  Assigns to `*this` *the result of computing the BHMZ05-widening of* `*this` *and* `y`.

- void limited_BHMZ05_extrapolation_assign (const BD_Shape &y, const Constraint_System &cs, unsigned *tp=0)

  Improves the result of the BHMZ05-widening computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`.

- void CC76_narrowing_assign (const BD_Shape &y)

  Assigns to `*this` *the result of restoring in* `y` *the constraints of* `*this` *that were lost by CC76-extrapolation applications.*

- void limited_CC76_extrapolation_assign (const BD_Shape &y, const Constraint_System &cs, unsigned *tp=0)

  Improves the result of the CC76-extrapolation computation by also enforcing those constraints in `cs` *that are satisfied by all the points of* `*this`.

- void H79_widening_assign (const BD_Shape &y, unsigned *tp=0)

  Assigns to `*this` *the result of computing the H79-widening between* `*this` *and* `y`.

- void limited_H79_extrapolation_assign (const BD_Shape &y, const Constraint_System &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`.

Member Functions that May Modify the Dimension of the Vector Space

- void add_space_dimensions_and_embed (dimension_type m)

  Adds `m` *new dimensions and embeds the old BDS into the new space.*

- void add_space_dimensions_and_project (dimension_type m)

  Adds `m` *new dimensions to the BDS and does not embed it in the new vector space.*

- void concatenate_assign (const BD_Shape &y)

  Seeing a BDS 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_space_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified dimensions.

- void remove_higher_space_dimensions (dimension_type new_dimension)

  Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.

- template<typename PartialFunction> void map_space_dimensions (const PartialFunction &pfunc)

  Remaps the dimensions of the vector space according to a partial function.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension that a BDS can handle.

Friends

- std::ostream & operator<< (std::ostream &s, const BD_Shape< T > &c)

  Output operator.

Related Functions

(Note that these are not member functions.)

- bool operator== (const BD_Shape< T > &x, const BD_Shape< T > &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same BDS.*

- bool operator!= (const BD_Shape< T > &x, const BD_Shape< T > &y)

  Returns `true` *if and only if* `x` *and* `y` *aren't the same BDS.*

- bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir)

  Computes the rectilinear (or Manhattan) distance between `x` *and* `y`.

- bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

  Computes the rectilinear (or Manhattan) distance between `x` *and* `y`.

- bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir)

  Computes the euclidean distance between `x` *and* `y`.

- bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

Computes the euclidean distance between x *and* y.

- bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir)

  Computes the L_∞ *distance between* x *and* y.

- bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const BD_Shape< T > &x, const BD_Shape< T > &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

  Computes the L_∞ *distance between* x *and* y.

- void swap (Parma_Polyhedra_Library::BD_Shape< T > &x, Parma_Polyhedra_Library::BD_-Shape< T > &y)

  Specializes `std::swap`.

9.1.1 Detailed Description

template<typename T> class Parma_Polyhedra_Library::BD_Shape< T >

A bounded difference shape.

The class template BD_Shape<T> allows for the efficient representation of a restricted kind of *topologically closed* convex polyhedra called *bounded difference shapes* (BDSs, for short). The name comes from the fact that the closed affine half-spaces that characterize the polyhedron can be expressed by constraints of the form $\pm x_i \leq k$ or $x_i - x_j \leq k$, where the inhomogeneous term k is a rational number.

Based on the class template type parameter `T`, a family of extended numbers is built and used to approximate the inhomogeneous term of bounded differences. These extended numbers provide a representation for the value $+\infty$, as well as *rounding-aware* implementations for several arithmetic functions. The value of the type parameter `T` may be one of the following:

- a bounded precision integer type (e.g., `int32_t` or `int64_t`);
- a bounded precision floating point type (e.g., `float` or `double`);
- an unbounded integer or rational type, as provided by GMP (i.e., `mpz_class` or `mpq_class`).

The user interface for BDSs is meant to be as similar as possible to the one developed for the polyhedron class C_Polyhedron. At the interface level, bounded differences are specified using objects of type Constraint: such a constraint is a bounded difference if it is of the form

$$a_i x_i - a_j x_j \bowtie b$$

where $\bowtie \in \{\leq, =, \geq\}$ and a_i, a_j, b are integer coefficients such that $a_i = 0$, or $a_j = 0$, or $a_i = a_j$. The user is warned that the above Constraint object will be mapped into a *correct* approximation that, depending on the expressive power of the chosen template argument `T`, may loose some precision. In particular, constraint objects that do not encode a bounded difference will be simply (and safely) ignored.

For instance, a Constraint object encoding $3x - 3y \leq 1$ will be approximated by:

- $x - y \leq 1$, if `T` is a (bounded or unbounded) integer type;
- $x - y \leq \frac{1}{3}$, if `T` is the unbounded rational type `mpq_class`;
- $x - y \leq k$, where $k > \frac{1}{3}$, if `T` is a floating point type (having no exact representation for $\frac{1}{3}$).

On the other hand, a Constraint object encoding $3x - y \leq 1$ will be safely ignored in all of the above cases.

In the following examples it is assumed that the type argument T is one of the possible instances listed above and that variables x, y and z are defined (where they are used) as follows:

```
Variable x(0);
Variable y(1);
Variable z(2);
```

Example 1

The following code builds a BDS corresponding to a cube in $\mathbb{R}^3$, given as a system of constraints:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 1);
cs.insert(y >= 0);
cs.insert(y <= 1);
cs.insert(z >= 0);
cs.insert(z <= 1);
BD_Shape<T> bd(cs);
```

Since only those constraints having the syntactic form of a *bounded difference* are considered, the following code will build the same BDS as above (i.e., the constraints 7, 8, and 9 are ignored):

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 1);
cs.insert(y >= 0);
cs.insert(y <= 1);
cs.insert(z >= 0);
cs.insert(z <= 1);
cs.insert(x + y <= 0);      // 7
cs.insert(x - z + x >= 0);  // 8
cs.insert(3*z - y <= 1);    // 9
BD_Shape<T> bd(cs);
```

9.1.2 Constructor & Destructor Documentation

9.1.2.1 template<typename T> Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (dimension_type *num_dimensions* = 0, Degenerate_Element *kind* = UNIVERSE) [inline, explicit]

Builds a universe or empty BDS of the specified space dimension.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the BDS;

kind Specifies whether the universe or the empty BDS has to be built.

9.1.2.2 template<typename T> Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Constraint_System & *cs*) [inline]

Builds a BDS from the system of constraints cs.

The BDS inherits the space dimension of cs.

Parameters:

cs A system of constraints: constraints that are not bounded differences are ignored (even though they may have contributed to the space dimension).

Exceptions:

std::invalid_argument Thrown if the system of constraints `cs` contains strict inequalities.

9.1.2.3 template<typename T> Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Generator_System & *gs*)

Builds a BDS from the system of generators `gs`.

Builds the smallest BDS containing the polyhedron defined by `gs`. The BDS inherits the space dimension of `gs`.

Exceptions:

std::invalid_argument Thrown if the system of generators is not empty but has no points.

9.1.2.4 template<typename T> Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Polyhedron & *ph*, Complexity_Class *complexity* = `ANY_COMPLEXITY`)

Builds a BDS from the polyhedron `ph`.

Builds a BDS containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the BDS built is the smallest one containing `ph`.

9.1.3 Member Function Documentation

9.1.3.1 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::contains (const BD_Shape< T > & *y*) const

Returns `true` if and only if `*this` contains `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.2 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::strictly_-contains (const BD_Shape< T > & *y*) const `[inline]`

Returns `true` if and only if `*this` strictly contains `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.3 template<typename T> Poly_Con_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation_with (const Constraint & *c*) const

Returns the relations holding between `*this` and the constraint `c`.

Exceptions:

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible or if `c` is a strict inequality or if `c` is not a bounded difference constraint.

9.1.3.4 template<typename T> Poly_Gen_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation_with (const Generator & *g*) const

Returns the relations holding between `*this` and the generator `g`.

Exceptions:

> ***std::invalid_argument*** Thrown if `*this` and generator `g` are dimension-incompatible.

9.1.3.5 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_-constraint (const Constraint & *c*)

Adds a copy of constraint `c` to the system of bounded differences defining `*this`.

Parameters:

> ***c*** The constraint to be added. If it is not a bounded difference, it will be simply ignored.

Exceptions:

> ***std::invalid_argument*** Thrown if `*this` and constraint `c` are dimension-incompatible, or if `c` is a strict inequality.

9.1.3.6 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::add_-constraint_and_minimize (const Constraint & *c*) `[inline]`

Adds a copy of constraint `c` to the system of bounded differences defining `*this`.

Returns:

> `false` if and only if the result is empty.

Parameters:

> ***c*** The constraint to be added. If it is not a bounded difference, it will be simply ignored.

Exceptions:

> ***std::invalid_argument*** Thrown if `*this` and constraint `c` are dimension-incompatible, or if `c` is a strict inequality.

9.1.3.7 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_-constraints (const Constraint_System & *cs*) `[inline]`

Adds the constraints in `cs` to the system of bounded differences defining `*this`.

Parameters:

> ***cs*** The constraints that will be added. Constraints that are not bounded differences will be simply ignored.

Exceptions:

> ***std::invalid_argument*** Thrown if `*this` and `cs` are dimension-incompatible, or if `cs` contains a strict inequality.

9.1.3.8 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::add_-constraints_and_minimize (const Constraint_System & *cs*) `[inline]`

Adds the constraints in `cs` to the system of bounded differences defining `*this`.

Returns:

`false` if and only if the result is empty.

Parameters:

cs The constraints that will be added. Constraints that are not bounded differences will be simply ignored.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible, or if `cs` contains a strict inequality.

9.1.3.9 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::intersection_-assign (const BD_Shape< T > & *y*)

Assigns to `*this` the intersection of `*this` and `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.10 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::intersection_assign_and_minimize (const BD_Shape< T > & *y*) `[inline]`

Assigns to `*this` the intersection of `*this` and `y`.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.11 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::bds_hull_-assign (const BD_Shape< T > & *y*)

Assigns to `*this` the smallest BDS containing the convex union of `*this` and `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.12 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::bds_hull_assign_and_minimize (const BD_Shape< T > & *y*) [inline]

Assigns to `*this` the smallest BDS containing the convex union of `*this` and `y`.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.13 template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::bds_hull_assign_if_exact (const BD_Shape< T > & *y*) [inline]

If the bds-hull of `*this` and `y` is exact, it is assigned to `*this` and `true` is returned, otherwise `false` is returned.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.14 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::bds_difference_assign (const BD_Shape< T > & *y*)

Assigns to `*this` the poly-difference of `*this` and `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.15 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_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.

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`.

9.1.3.16 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::affine_- preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_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.

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`.

9.1.3.17 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the image of `*this` with respect to the affine relation $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$, 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.

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 `relsym` is a strict relation symbol.

9.1.3.18 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*)

Assigns to `*this` the image of `*this` with respect to the affine relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, 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 `relsym` is a strict relation symbol.

9.1.3.19 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the preimage of `*this` with respect to the affine relation $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$, 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.

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 `relsym` is a strict relation symbol.

9.1.3.20 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::time_elapse_assign (const BD_Shape< T > & *y*) [inline]

Assigns to `*this` the result of computing the time-elapse between `*this` and `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.1.3.21 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::CC76_extrapolation_assign (const BD_Shape< T > & *y*, unsigned * *tp* = `0`) [inline]

Assigns to `*this` the result of computing the CC76-extrapolation between `*this` and `y`.

Parameters:

y A BDS 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 dimension-incompatible.

9.1.3.22 template<typename T> template<typename Iterator> void Parma_Polyhedra_Library::BD_Shape< T >::CC76_extrapolation_assign (const BD_Shape< T > & *y*, Iterator *first*, Iterator *last*, unsigned * *tp* = `0`)

Assigns to `*this` the result of computing the CC76-extrapolation between `*this` and `y`.

Parameters:

y A BDS that *must* be contained in `*this`.

first An iterator referencing the first stop-point.

last An iterator referencing one past the last stop-point.

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 dimension-incompatible.

9.1.3.23 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::BHMZ05_-widening_assign (const BD_Shape< T > & *y*, unsigned ∗ *tp* = `0`)

Assigns to `*this` the result of computing the BHMZ05-widening of `*this` and `y`.

Parameters:

y A BDS 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 dimension-incompatible.

9.1.3.24 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::limited_-BHMZ05_extrapolation_assign (const BD_Shape< T > & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = `0`)

Improves the result of the BHMZ05-widening computation by also enforcing those constraints in `cs` that are satisfied by all the points of `*this`.

Parameters:

y A BDS that *must* be contained in `*this`.

cs The system of constraints used to improve the widened BDS.

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 dimension-incompatible or if `cs` contains a strict inequality.

9.1.3.25 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::CC76_-narrowing_assign (const BD_Shape< T > & *y*)

Assigns to `*this` the result of restoring in `y` the constraints of `*this` that were lost by CC76-extrapolation applications.

Parameters:

y A BDS that *must* contain `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

Note:

As was the case for widening operators, the argument `y` is meant to denote the value computed in the previous iteration step, whereas `*this` denotes the value computed in the current iteration step (in the *descreasing* iteration sequence). Hence, the call `x.CC76_narrowing_assign(y)` will assign to `x` the result of the computation $y \Delta x$.

9.1.3.26 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::limited_-CC76_extrapolation_assign (const BD_Shape< T > & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0)

Improves the result of the CC76-extrapolation computation by also enforcing those constraints in `cs` that are satisfied by all the points of `*this`.

Parameters:

y A BDS that *must* be contained in `*this`.

cs The system of constraints used to improve the widened BDS.

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 dimension-incompatible or if `cs` contains a strict inequality.

9.1.3.27 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::H79_-widening_assign (const BD_Shape< T > & *y*, unsigned ∗ *tp* = 0) `[inline]`

Assigns to `*this` the result of computing the H79-widening between `*this` and `y`.

Parameters:

y A BDS 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 dimension-incompatible.

9.1.3.28 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::limited_-H79_extrapolation_assign (const BD_Shape< T > & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0) `[inline]`

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 BDS that *must* be contained in `*this`.

cs The system of constraints used to improve the widened BDS.

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 dimension-incompatible.

9.1.3.29 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_embed (dimension_type *m*)

Adds `m` new dimensions and embeds the old BDS 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 BDS, which is defined by a system of bounded differences in which the variables running through the new dimensions are unconstrained. For instance, when starting from the BDS $\mathcal{B} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the BDS

$$\bigl\{\, (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x, y)^{\mathrm{T}} \in \mathcal{B} \,\bigr\}.$$

9.1.3.30 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_project (dimension_type *m*)

Adds `m` new dimensions to the BDS and does not embed it in the new vector space.

Parameters:

m The number of dimensions to add.

The new dimensions will be those having the highest indexes in the new BDS, which is defined by a system of bounded differences in which the variables running through the new dimensions are all constrained to be equal to 0. For instance, when starting from the BDS $\mathcal{B} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the BDS

$$\bigl\{\, (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x, y)^{\mathrm{T}} \in \mathcal{B} \,\bigr\}.$$

9.1.3.31 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::concatenate_assign (const BD_Shape< T > & *y*)

Seeing a BDS 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 $B \subseteq \mathbb{R}^n$ and $D \subseteq \mathbb{R}^m$ be the BDSs corresponding, on entry, to `*this` and `y`, respectively. Upon successful completion, `*this` will represent the BDS $R \subseteq \mathbb{R}^{n+m}$ such that

$$R \stackrel{\mathrm{def}}{=} \Bigl\{\, (x_1, \ldots, x_n, y_1, \ldots, y_m)^{\mathrm{T}} \Bigm| (x_1, \ldots, x_n)^{\mathrm{T}} \in B, (y_1, \ldots, y_m)^{\mathrm{T}} \in D \,\Bigr\}.$$

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`.

9.1.3.32 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::remove_space_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`.

9.1.3.33 template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::remove_higher_space_dimensions (dimension_type *new_dimension*) `[inline]`

Removes the higher dimensions so that the resulting space will have dimension `new_dimension`.

Exceptions:

std::invalid_argument Thrown if `new_dimension` is greater than the space dimension of `*this`.

9.1.3.34 template<typename T> template<typename PartialFunction> void Parma_Polyhedra_Library::BD_Shape< T >::map_space_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 co-domain (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 co-domain 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.

9.1.4 Friends And Related Function Documentation

9.1.4.1 template<typename T> std::ostream & operator<< (std::ostream & *s*, const BD_Shape< T > & *c*) `[friend]`

Output operator.

Writes a textual representation of `bds` on `s`: `false` is written if `bds` is an empty polyhedron; `true` is written if `bds` is the universe polyhedron; a system of constraints defining `bds` is written otherwise, all constraints separated by ", ".

9.1.4.2 template<typename T> bool operator== (const BD_Shape< T > & *x*, const BD_Shape< T > & *y*) `[related]`

Returns `true` if and only if `x` and `y` are the same BDS.

Note that `x` and `y` may be dimension-incompatible shapes: in this case, the value `false` is returned.

9.1.4.3 template<typename T> bool operator!= (const BD_Shape< T > & *x*, const BD_Shape< T > & *y*) `[related]`

Returns `true` if and only if `x` and `y` aren't the same BDS.

Note that `x` and `y` may be dimension-incompatible shapes: in this case, the value `true` is returned.

9.1.4.4 template<typename T> bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*) `[related]`

Computes the rectilinear (or Manhattan) distance between `x` and `y`.

If the rectilinear distance between `x` and `y` is defined, stores an approximation of it into `r` and returns `true`; returns `false` otherwise.

The direction of the approximation is specified by `dir`.

All computations are performed using variables of type Checked_Number<To, Extended_Number_-Policy>.

9.1.4.5 template<typename T> bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*, Temp & *tmp0*, Temp & *tmp1*, Temp & *tmp2*) `[related]`

Computes the rectilinear (or Manhattan) distance between `x` and `y`.

If the rectilinear distance between `x` and `y` is defined, stores an approximation of it into `r` and returns `true`; returns `false` otherwise.

The direction of the approximation is specified by `dir`.

All computations are performed using the temporary variables `tmp0`, `tmp1` and `tmp2`.

9.1.4.6 template<typename T> bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*) `[related]`

Computes the euclidean distance between `x` and `y`.

If the euclidean distance between x and y is defined, stores an approximation of it into r and returns true; returns false otherwise.

The direction of the approximation is specified by dir.

All computations are performed using variables of type Checked_Number<To, Extended_Number_-Policy>.

9.1.4.7 template<typename T> bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*, Temp & *tmp0*, Temp & *tmp1*, Temp & *tmp2*) [related]

Computes the euclidean distance between x and y.

If the euclidean distance between x and y is defined, stores an approximation of it into r and returns true; returns false otherwise.

The direction of the approximation is specified by dir.

All computations are performed using the temporary variables tmp0, tmp1 and tmp2.

9.1.4.8 template<typename T> bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*) [related]

Computes the L_∞ distance between x and y.

If the L_∞ distance between x and y is defined, stores an approximation of it into r and returns true; returns false otherwise.

The direction of the approximation is specified by dir.

All computations are performed using variables of type Checked_Number<To, Extended_Number_-Policy>.

9.1.4.9 template<typename T> bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, const Rounding_Dir *dir*, Temp & *tmp0*, Temp & *tmp1*, Temp & *tmp2*) [related]

Computes the L_∞ distance between x and y.

If the L_∞ distance between x and y is defined, stores an approximation of it into r and returns true; returns false otherwise.

The direction of the approximation is specified by dir.

All computations are performed using the temporary variables tmp0, tmp1 and tmp2.

9.2 Parma_Polyhedra_Library::BHRZ03_Certificate Class Reference

The convergence certificate for the BHRZ03 widening operator.

Public Member Functions

- BHRZ03_Certificate ()

  Default constructor.

- BHRZ03_Certificate (const Polyhedron &ph)

  Constructor: computes the certificate for `ph`.

- BHRZ03_Certificate (const BHRZ03_Certificate &y)

  Copy constructor.

- ~BHRZ03_Certificate ()

  Destructor.

- int compare (const BHRZ03_Certificate &y) const

  The comparison function for certificates.

- int compare (const Polyhedron &ph) const

  Compares `*this` *with the certificate for polyhedron* `ph`.

Classes

- struct Compare

  A total ordering on BHRZ03 certificates.

9.2.1 Detailed Description

The convergence certificate for the BHRZ03 widening operator.

Convergence certificates are used to instantiate the BHZ03 framework so as to define widening operators for the finite powerset domain.

Note:

Each convergence certificate has to be used together with a compatible widening operator. In particular, BHRZ03_Certificate can certify the convergence of both the BHRZ03 and the H79 widenings.

9.2.2 Member Function Documentation

9.2.2.1 int Parma_Polyhedra_Library::BHRZ03_Certificate::compare (const BHRZ03_Certificate & *y*) const

The comparison function for certificates.

Returns:

-1, 0 or 1 depending on whether `*this` is smaller than, equal to, or greater than `y`, respectively.

Compares `*this` with `y`, using a total ordering which is a refinement of the limited growth ordering relation for the BHRZ03 widening.

9.3 Parma_Polyhedra_Library::BHRZ03_Certificate::Compare Struct Reference

A total ordering on BHRZ03 certificates.

Public Member Functions

- bool operator() (const BHRZ03_Certificate &x, const BHRZ03_Certificate &y) const

  Returns `true` *if and only if* `x` *comes before* `y`*.*

9.3.1 Detailed Description

A total ordering on BHRZ03 certificates.

This binary predicate defines a total ordering on BHRZ03 certificates which is used when storing information about sets of polyhedra.

9.4 Parma_Polyhedra_Library::C_Polyhedron Class Reference

A closed convex polyhedron.

Inherits Parma_Polyhedra_Library::Polyhedron.

Public Member Functions

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

  Builds either the universe or the empty C polyhedron.

- C_Polyhedron (const Constraint_System &cs)

  Builds a C polyhedron from a system of constraints.

- C_Polyhedron (Constraint_System &cs)

  Builds a C polyhedron recycling a system of constraints.

- C_Polyhedron (const Generator_System &gs)

  Builds a C polyhedron from a system of generators.

- C_Polyhedron (Generator_System &gs)

  Builds a C polyhedron recycling a system of generators.

- C_Polyhedron (const Congruence_System &cgs)

  Builds a C polyhedron from a system of congruences.

- C_Polyhedron (Congruence_System &cgs)

  Builds an C polyhedron recycling a system of congruences.

- C_Polyhedron (const NNC_Polyhedron &y)

  Builds a C polyhedron representing the topological closure of 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 & operator= (const NNC_Polyhedron &y)

  Assigns to `*this` the topological closure of the NNC polyhedron `y`.

- ~C_Polyhedron ()

  Destructor.

- bool poly_hull_assign_if_exact (const C_Polyhedron &y)

  If the poly-hull of `*this` and `y` is exact it is assigned to `*this` and `true` is returned, otherwise `false` is returned.

- bool upper_bound_assign_if_exact (const C_Polyhedron &y)

  Same as poly_hull_assign_if_exact(y).

9.4.1 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.

9.4.2 Constructor & Destructor Documentation

9.4.2.1 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (dimension_type *num_dimensions* = `0`, Degenerate_Element *kind* = `UNIVERSE`) `[inline, 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.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

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

9.4.2.2 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Constraint_System & *cs*) [inline, explicit]

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.

9.4.2.3 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Constraint_System & *cs*) [inline, explicit]

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.

9.4.2.4 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Generator_System & *gs*) [inline, explicit]

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.

9.4.2.5 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Generator_System & *gs*) [inline, explicit]

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.

9.4.2.6 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Congruence_System & *cgs*) `[explicit]`

Builds a C polyhedron from a system of congruences.

The polyhedron inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

9.4.2.7 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Congruence_System & *cgs*) `[explicit]`

Builds an C polyhedron recycling a system of congruences.

The polyhedron inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

9.4.2.8 template<typename Box> Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Box & *box*, From_Bounding_Box *dummy*) `[inline]`

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::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

std::invalid_argument Thrown if `box` has intervals that are not topologically closed (i.e., having some finite but open bounds).

9.4.3 Member Function Documentation

9.4.3.1 bool Parma_Polyhedra_Library::C_Polyhedron::poly_hull_assign_if_exact (const C_-Polyhedron & y)

If the poly-hull of `*this` and `y` is exact it is assigned to `*this` and `true` is returned, otherwise `false` is returned.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.5 Parma_Polyhedra_Library::Checked_Number< T, Policy > Class Template Reference

A wrapper for numeric types implementing a given policy.

Public Member Functions

- bool OK () const

  Checks if all the invariants are satisfied.

- Result classify (bool nan=true, bool inf=true, bool sign=true) const

  Classifies ∗this.

Constructors

- Checked_Number ()

  Default constructor.

- Checked_Number (const Checked_Number &y)

  Copy-constructor.

- template<typename From, typename From_Policy> Checked_Number (const Checked_-Number< From, From_Policy > &y, Rounding_Dir dir)

  Direct initialization from a Checked_Number and rounding mode.

- Checked_Number (const signed char y, Rounding_Dir dir)

  Direct initialization from a signed char and rounding mode.

- Checked_Number (const signed short y, Rounding_Dir dir)

  Direct initialization from a signed short and rounding mode.

- Checked_Number (const signed int y, Rounding_Dir dir)

  Direct initialization from a signed int and rounding mode.

- Checked_Number (const signed long y, Rounding_Dir dir)

  Direct initialization from a signed long and rounding mode.

- Checked_Number (const signed long long y, Rounding_Dir dir)

  Direct initialization from a signed long long and rounding mode.

- Checked_Number (const unsigned char y, Rounding_Dir dir)

  Direct initialization from an unsigned char and rounding mode.
- Checked_Number (const unsigned short y, Rounding_Dir dir)

  Direct initialization from an unsigned short and rounding mode.
- Checked_Number (const unsigned int y, Rounding_Dir dir)

  Direct initialization from an unsigned int and rounding mode.
- Checked_Number (const unsigned long y, Rounding_Dir dir)

  Direct initialization from an unsigned long and rounding mode.
- Checked_Number (const unsigned long long y, Rounding_Dir dir)

  Direct initialization from an unsigned long long and rounding mode.
- Checked_Number (const float y, Rounding_Dir dir)

  Direct initialization from a float and rounding mode.
- Checked_Number (const double y, Rounding_Dir dir)

  Direct initialization from a double and rounding mode.
- Checked_Number (const long double y, Rounding_Dir dir)

  Direct initialization from a long double and rounding mode.
- Checked_Number (const mpq_class &y, Rounding_Dir dir)

  Direct initialization from a rational and rounding mode.
- Checked_Number (const mpz_class &y, Rounding_Dir dir)

  Direct initialization from an unbounded integer and rounding mode.
- Checked_Number (const char ∗y, Rounding_Dir dir)

  Direct initialization from a C string and rounding mode.
- Checked_Number (const Minus_Infinity &y, Rounding_Dir dir)

  Direct initialization from minus infinity and rounding mode.
- Checked_Number (const Plus_Infinity &y, Rounding_Dir dir)

  Direct initialization from plus infinity and rounding mode.
- Checked_Number (const Not_A_Number &y, Rounding_Dir dir)

  Direct initialization from NAN and rounding mode.
- template<typename From, typename From_Policy> Checked_Number (const Checked_-Number< From, From_Policy > &y)

  Direct initialization from a Checked_Number, default rounding mode.
- Checked_Number (const signed char y)

  Direct initialization from a signed char, default rounding mode.
- Checked_Number (const signed short y)

  Direct initialization from a signed short, default rounding mode.

- Checked_Number (const signed int y)

  Direct initialization from a signed int, default rounding mode.

- Checked_Number (const signed long y)

  Direct initialization from a signed long, default rounding mode.

- Checked_Number (const signed long long y)

  Direct initialization from a signed long long, default rounding mode.

- Checked_Number (const unsigned char y)

  Direct initialization from an unsigned char, default rounding mode.

- Checked_Number (const unsigned short y)

  Direct initialization from an unsigned short, default rounding mode.

- Checked_Number (const unsigned int y)

  Direct initialization from an unsigned int, default rounding mode.

- Checked_Number (const unsigned long y)

  Direct initialization from an unsigned long, default rounding mode.

- Checked_Number (const unsigned long long y)

  Direct initialization from an unsigned long long, default rounding mode.

- Checked_Number (const float y)

  Direct initialization from a float, default rounding mode.

- Checked_Number (const double y)

  Direct initialization from a double, default rounding mode.

- Checked_Number (const long double y)

  Direct initialization from a long double, default rounding mode.

- Checked_Number (const mpq_class &y)

  Direct initialization from a rational, default rounding mode.

- Checked_Number (const mpz_class &y)

  Direct initialization from an unbounded integer, default rounding mode.

- Checked_Number (const char ∗y)

  Direct initialization from a C string, default rounding mode.

- Checked_Number (const Minus_Infinity &y)

  Direct initialization from minus infinity, default rounding mode.

- Checked_Number (const Plus_Infinity &y)

  Direct initialization from plus infinity, default rounding mode.

- Checked_Number (const Not_A_Number &y)

  Direct initialization from NAN, default rounding mode.

Accessors and Conversions

- operator T () const

  Conversion operator: returns a copy of the underlying numeric value.

- T & raw_value ()

  Returns a reference to the underlying numeric value.

- const T & raw_value () const

  Returns a const reference to the underlying numeric value.

Assignment Operators

- Checked_Number & operator= (const Checked_Number &y)

  Assignment operator.

- template<typename From, typename From_Policy> Checked_Number & operator= (const Checked_Number< From, From_Policy > &y)

  Assignment operator.

- template<typename From> Checked_Number & operator= (const From &y)

  Assignment operator.

- Checked_Number & operator= (const Not_A_Number &y)

  Assignment operator.

- Checked_Number & operator= (const Minus_Infinity &y)

  Assignment operator.

- Checked_Number & operator= (const Plus_Infinity &y)

  Assignment operator.

- template<typename From_Policy> Checked_Number & operator+= (const Checked_Number< T, From_Policy > &y)

  Add and assign operator.

- Checked_Number & operator+= (const T &y)

  Add and assign operator.

- template<typename From, typename From_Policy> Checked_Number & operator+= (const Checked_Number< From, From_Policy > &y)

  Add and assign operator.

- template<typename From_Policy> Checked_Number & operator-= (const Checked_Number< T, From_Policy > &y)

  Subtract and assign operator.

- Checked_Number & operator-= (const T &y)

  Subtract and assign operator.

- template<typename From, typename From_Policy> Checked_Number & operator-= (const Checked_Number< From, From_Policy > &y)

  Subtract and assign operator.

- template<typename From> Checked_Number & operator-= (const From &y)

  Subtract and assign operator.

- template<typename From_Policy> Checked_Number & operator *= (const Checked_Number< T, From_Policy > &y)

  Multiply and assign operator.

- Checked_Number & operator *= (const T &y)

  Multiply and assign operator.

- template<typename From, typename From_Policy> Checked_Number & operator *= (const Checked_Number< From, From_Policy > &y)

  Multiply and assign operator.

- template<typename From> Checked_Number & operator *= (const From &y)

  Multiply and assign operator.

- template<typename From_Policy> Checked_Number & operator/= (const Checked_Number< T, From_Policy > &y)

  Divide and assign operator.

- Checked_Number & operator/= (const T &y)

  Divide and assign operator.

- template<typename From, typename From_Policy> Checked_Number & operator/= (const Checked_Number< From, From_Policy > &y)

  Divide and assign operator.

- template<typename From> Checked_Number & operator/= (const From &y)

  Divide and assign operator.

- template<typename From_Policy> Checked_Number & operator%= (const Checked_Number< T, From_Policy > &y)

  Compute remainder and assign operator.

- Checked_Number & operator%= (const T &y)

  Compute remainder and assign operator.

- template<typename From, typename From_Policy> Checked_Number & operator%= (const Checked_Number< From, From_Policy > &y)

  Compute remainder and assign operator.

- template<typename From> Checked_Number & operator%= (const From &y)

  Compute remainder and assign operator.

Increment and Decrement Operators

- Checked_Number & operator++ ()

  Pre-increment operator.

- Checked_Number operator++ (int)

  Post-increment operator.

- Checked_Number & operator-- ()

  Pre-decrement operator.

- Checked_Number operator-- (int)

  Post-decrement operator.

Related Functions

(Note that these are not member functions.)

- void swap (Checked_Number< T, Policy > &x, Checked_Number< T, Policy > &y)

  Swaps `*this` *with* `y`.

Accessor Functions

- const T & raw_value (const Checked_Number< T, Policy > &x)

  Returns a const reference to the underlying native integer value.

- T & raw_value (Checked_Number< T, Policy > &x)

  Returns a reference to the underlying native integer value.

Memory Size Inspection Functions

- size_t total_memory_in_bytes (const Checked_Number< T, Policy > &x)

  Returns the total size in bytes of the memory occupied by `x`.

- size_t external_memory_in_bytes (const Checked_Number< T, Policy > &x)

  Returns the size in bytes of the memory managed by `x`.

Arithmetic Operators

- Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > &x)

  Unary plus operator.

- Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > &x)

  Unary minus operator.

- void neg_assign (Checked_Number< T, Policy > &x)

  Assigns to `x` *its negation.*

- void add_mul_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  Assigns to `x` *the value* `x + y * z`.

- void sub_mul_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  Assigns to `x` *the value* `x - y * z`.

- void gcd_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  Assigns to `x` *the greatest common divisor of* `y` *and* `z`*.*

- void gcdext_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z, Checked_Number< T, Policy > &s, Checked_Number< T, Policy > &t)

  Assigns to `x` *the greatest common divisor of* `y` *and* `z`*, setting* `s` *and* `t` *such that s*∗*y + t*∗*z = x = gcd(y, z).*

- void lcm_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  Assigns to `x` *the least common multiple of* `y` *and* `z`*.*

- void exact_div_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  If `z` *divides* `y`*, assigns to* `x` *the quotient of the integer division of* `y` *and* `z`*.*

- void sqrt_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)

  Assigns to `x` *the integer square root of* `y`*.*

Relational Operators and Comparison Functions

- bool operator== (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Equality operator.

- bool operator!= (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Disequality operator.

- bool operator>= (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Greater than or equal to operator.

- bool operator> (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Greater than operator.

- bool operator<= (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Less than or equal to operator.

- bool operator< (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

  Less than operator.

- int sgn (const Checked_Number< T, Policy > &x)

  Returns -1*,* 0 *or* 1 *depending on whether the value of* `x` *is negative, zero or positive, respectively.*

- int cmp (const Checked_Number< T1, Policy1 > &x, const Checked_Number< T2, Policy2 > &y)

Returns a negative, zero or positive value depending on whether `x` *is lower than, equal to or greater than* `y`*, respectively.*

Input-Output Operators

- std::ostream & operator<< (std::ostream &os, const Checked_Number< T, Policy > &x)
  Output operator.
- Result input (std::istream &is, Checked_Number< T, Policy > &x, Rounding_Dir dir)
  Input function.
- std::istream & operator>> (std::istream &is, Checked_Number< T, Policy > &x)
  Input operator.

9.5.1 Detailed Description

template<typename T, typename Policy> class Parma_Polyhedra_Library::Checked_Number< T, Policy >

A wrapper for numeric types implementing a given policy.

The wrapper and related functions implement an interface which is common to all kinds of coefficient types, therefore allowing for a uniform coding style. This class also implements the policy encoded by the second template parameter. The default policy is to perform the detection of overflow errors.

9.5.2 Member Function Documentation

9.5.2.1 template<typename T, typename Policy> Result Parma_Polyhedra_Library::Checked_Number< T, Policy >::classify (bool *nan* = `true`, bool *inf* = `true`, bool *sign* = `true`) const `[inline]`

Classifies ∗this.

Returns the appropriate Result characterizing:

- whether `*this` is NAN, if `nan` is `true`;
- whether `*this` is a (positive or negative) infinity, if `inf` is `true`;
- the sign of `*this`, if `sign` is `true`.

9.5.3 Friends And Related Function Documentation

9.5.3.1 template<typename T, typename Policy> void exact_div_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) `[related]`

If `z` divides `y`, assigns to `x` the quotient of the integer division of `y` and `z`.

The behavior is undefined if `z` does not divide `y`.

9.5.3.2 template<typename T, typename Policy> Result input (std::istream & *is*, Checked_Number< T, Policy > & *x*, Rounding_Dir *dir*) `[related]`

Input function.

Parameters:

is Input stream to read from;

x Number (possibly extended) to assign to in case of successful reading;

dir Rounding mode to be applied.

Returns:

Result of the input operation. Success, success with imprecision, overflow, parsing error: all possibilities are taken into account, checked for, and properly reported.

This function attempts reading a (possibly extended) number from the given stream `is`, possibly rounding as specified by `dir`, assigning the result to `x` upon success, and returning the appropriate Result.

The input syntax allows the specification of:

- plain base-10 integer numbers as `34976098`, `-77` and `+13`;
- base-10 integer numbers in scientific notation as `15e2` and `15*^2` (both meaning $15 \cdot 10^2 = 1500$), `9200e-2` and `-18*^+11111111111111111`;
- base-10 rational numbers in fraction notation as `15/3` and `15/-3`;
- base-10 rational numbers in fraction/scientific notation as `15/30e-1` (meaning 5) and `15*^-3/29e2` (meaning $3/580000$);
- base-10 rational numbers in floating point notation as `71.3` (meaning $713/10$) and `-0.123456` (meaning $-1929/15625$);
- base-10 rational numbers in floating point scientific notation as `2.2e-1` (meaning $11/50$) and `-2.20001*^+3` (meaning $-220001/100$);
- integers and rationals (in fractional, floating point and scientific notations) specified by using Mathematica-style bases, in the range from 2 to 36, as `2^^11` (meaning 3), `36^^z` (meaning 35), `36^^xyz` (meaning 44027), `2^^11.1` (meaning $7/2$), `10^^2e3` (meaning 2000), `8^^2e3` (meaning 1024), `8^^2.1e3` (meaning 1088), `8^^20402543.120347e7` (meaning 9073863231288), `8^^2.1` (meaning $17/8$); note that the base and the exponent are always written as plain base-10 integer numbers; also, when an ambiguity may arise, the character `e` is interpreted as a digit, so that `16^^1e2` (meaning 482) is different from `16^^1*^2` (meaning 256);
- the C-style hexadecimal prefix `0x` is interpreted as the Mathematica-style prefix `16^^`;
- special values like `inf` and `+inf` (meaning $+\infty$), `-inf` (meaning $-\infty$), and `nan` (meaning "not a number").

The rationale behind the accepted syntax can be summarized as follows:

- if the syntax is accepted by Mathematica, then this function accepts it with the same semantics;
- if the syntax is acceptable as standard C++ integer or floating point literal (except for octal notation and type suffixes, which are not supported), then this function accepts it with the same semantics;
- natural extensions of the above are accepted with the natural extensions of the semantics;

- special values are accepted.

Valid syntax is more formally and completely specified by the following grammar, with the additional provisos that everything is *case insensitive*, that the syntactic category `BDIGIT` is further restricted by the current base and that for all bases above 14, any `e` is always interpreted as a digit and never as a delimiter for the exponent part (if such a delimiter is desired, it has to be written as `*^`).

```
number   : NAN
         | SIGN INF
         | INF
         | num
         | num DIV num
         ;

num      : unum
         | SIGN unum

unum     : unum1
         | HEX unum1
         | base BASE unum1
         ;

unum1    : mantissa
         | mantissa EXP exponent
         ;

mantissa: bdigits
         | POINT bdigits
         | bdigits POINT
         | bdigits POINT bdigits
         ;

exponent: SIGN digits
         | digits
         ;

bdigits  : BDIGIT
         | bdigits BDIGIT
         ;

digits   : DIGIT
         | digits DIGIT
         ;
```

```
INF      : 'inf'
         ;

NAN      : 'nan'
         ;

SIGN     : '-'
         | '+'
         ;

EXP      : 'e'
         | '*^'
         ;

POINT    : '.'
         ;

DIV      : '/'
         ;

MINUS    : '-'
         ;

PLUS     : '+'
;

HEX      : '0x'
;

BASE     : '^^'
         ;

DIGIT    : '0' .. '9'
         ;

BDIGIT   : '0' .. '9'
         | 'a' .. 'z'
         ;
```

9.6 Parma_Polyhedra_Library::Congruence Class Reference

A linear congruence.

Public Member Functions

- Congruence (const Congruence &cg)

  Ordinary copy-constructor.

- Congruence (const Constraint &c)

  Copy-constructs (modulo 0) from equality constraint `c`*.*

- ∼Congruence ()

Destructor.

- Congruence & operator= (const Congruence &cg)

  Assignment operator.

- dimension_type space_dimension () const

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

- Coefficient_traits::const_reference coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- Coefficient_traits::const_reference inhomogeneous_term () const

  Returns the inhomogeneous term of `*this`.

- Coefficient_traits::const_reference modulus () const

  Returns a const reference to the modulus of `*this`.

- Congruence & operator/= (const Coefficient_traits::const_reference k)

  Multiplies `k` *into the modulus of* `*this`.

- bool is_trivial_true () const

  Returns `true` *if and only if* `*this` *is a trivially true congruence.*

- bool is_trivial_false () const

  Returns `true` *if and only if* `*this` *is a trivially false congruence.*

- bool is_proper_congruence () const

  Returns `true` *if the modulus is greater than zero.*

- bool is_equality () const

  Returns `true` *if* `*this` *is an equality.*

- bool is_equal_at_dimension (dimension_type dim, const Congruence &cg) const

  Returns `true` *if* `*this` *is equal to* `cg` *in dimension* `dim`.

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- bool OK () const

  Checks if all the invariants are satisfied.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Congruence can handle.

- static const Congruence & zero_dim_integrality ()

  Returns a reference to the true (zero-dimension space) congruence $0 = 1 \pmod{1}$, *also known as the* integrality congruence.

- static const Congruence & zero_dim_false ()

  Returns a reference to the false (zero-dimension space) congruence $0 = 1 \pmod{0}$.

Protected Member Functions

- void sign_normalize ()

  Normalizes the signs.

- void normalize ()

  Normalizes signs and the inhomogeneous term.

- void strong_normalize ()

  Calls normalize, then divides out common factors.

Friends

- void swap (PPL::Congruence &x, PPL::Congruence &y)

  Specializes `std::swap`.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Congruence &c)

  Output operators.

- bool operator== (const Congruence &x, const Congruence &y)

  Returns `true` *if and only if* `x` *and* `y` *are equivalent.*

- bool operator!= (const Congruence &x, const Congruence &y)

  Returns `false` *if and only if* `x` *and* `y` *are equivalent.*

- Congruence operator/ (const Congruence &cg, const Coefficient_traits::const_reference k)

  Returns a copy of `cg`*, multiplying* `k` *into the copy's modulus.*

- Congruence operator/ (const Constraint &c, const Coefficient_traits::const_reference m)

  Creates a congruence from `c`*, with* `m` *as the modulus.*

9.6.1 Detailed Description

A linear congruence.

An object of the class Congruence is a congruence:

- $\mathrm{cg} = \sum_{i=0}^{n-1} a_i x_i + b = 0 \pmod{m}$

where n is the dimension of the space, a_i is the integer coefficient of variable x_i, b is the integer inhomogeneous term and m is the integer modulus; if $m = 0$, then cg represents the equality congruence $\sum_{i=0}^{n-1} a_i x_i + b = 0$ and, if $m \neq 0$, then the congruence cg is said to be a proper congruence.

How to build a congruence

Congruences $\pmod{1}$ are typically built by applying the congruence symbol '`%=`' to a pair of linear expressions. Congruences with modulus `m` are typically constructed by building a congruence $\pmod{1}$ using the given pair of linear expressions and then adding the modulus `m` using the modulus symbol is '`/`'.

The space dimension of a congruence 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 congruence $3x + 5y - z = 0$, having space dimension 3:

```
Congruence eq_cg((3*x + 5*y - z %= 0) / 0);
```

The following code builds the congruence $4x = 2y - 13 \pmod{1}$, having space dimension 2:

```
Congruence mod1_cg(4*x %= 2*y - 13);
```

The following code builds the congruence $4x = 2y - 13 \pmod{2}$, having space dimension 2:

```
Congruence mod2_cg((4*x %= 2*y - 13) / 2);
```

An unsatisfiable congruence on the zero-dimension space $\mathbb{R}^0$ can be specified as follows:

```
Congruence false_cg = Congruence::zero_dim_false();
```

Equivalent, but more involved ways are the following:

```
Congruence false_cg1((Linear_Expression::zero() %= 1) / 0);
Congruence false_cg2((Linear_Expression::zero() %= 1) / 2);
```

In contrast, the following code defines an unsatisfiable congruence having space dimension 3:

```
Congruence false_cg3((0*z %= 1) / 0);
```

How to inspect a congruence

Several methods are provided to examine a congruence and extract all the encoded information: its space dimension, its modulus and the value of its integer coefficients.

Example 2

The following code shows how it is possible to access the modulus as well as each of the coefficients. Given a congruence with linear expression `e` and modulus `m` (in this case $x - 5y + 3z = 4 \pmod 5$), we construct a new congruence with the same modulus `m` but where the linear expression is $2e$ ($2x - 10y + 6z = 8 \pmod 5$).

```
Congruence cg1((x - 5*y + 3*z %= 4) / 5);
cout << "Congruence cg1: " << cg1 << endl;
Coefficient m = cg1.modulus();
if (m == 0)
  cout << "Congruence cg1 is an equality." << endl;
else {
  Linear_Expression e;
  for (int i = cg1.space_dimension() - 1; i >= 0; --i)
    e += 2 * cg1.coefficient(Variable(i)) * Variable(i);
    e += 2 * cg1.inhomogeneous_term();
  Congruence cg2((e %= 0) / m);
  cout << "Congruence cg2: " << cg2 << endl;
}
```

The actual output could be the following:

```
Congruence cg1: A - 5*B + 3*C %= 4 / 5
Congruence cg2: 2*A - 10*B + 6*C %= 8 / 5
```

Note that, in general, the particular output obtained can be syntactically different from the (semantically equivalent) congruence considered.

9.6.2 Constructor & Destructor Documentation

9.6.2.1 Parma_Polyhedra_Library::Congruence::Congruence (const Constraint & *c*) `[explicit]`

Copy-constructs (modulo 0) from equality constraint `c`.

Exceptions:

std::invalid_argument Thrown if `c` is a relation.

9.6.3 Member Function Documentation

9.6.3.1 Coefficient_traits::const_reference Parma_Polyhedra_Library::Congruence::coefficient (Variable *v*) const `[inline]`

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`.

9.6.3.2 Congruence & Parma_Polyhedra_Library::Congruence::operator/= (const Coefficient_-traits::const_reference *k*) `[inline]`

Multiplies `k` into the modulus of `*this`.

If called with `*this` representing the congruence $e_1 = e_2 \pmod m$, then it returns with *this representing the congruence $e_1 = e_2 \pmod{mk}$.

9.6.3.3 bool Parma_Polyhedra_Library::Congruence::is_trivial_true () const

Returns `true` if and only if `*this` is a trivially true congruence.

Trivially true congruences are of one the following two forms:

- an equality: $\sum_{i=0}^{n-1} 0x_i + 0 == 0$; or
- a proper congruence: $\sum_{i=0}^{n-1} 0x_i + b\% = 0/m$, where n is the space dimension and m is the modulus.

9.6.3.4 bool Parma_Polyhedra_Library::Congruence::is_trivial_false () const

Returns `true` if and only if `*this` is a trivially false congruence.

Trivially false congruences have one of the following two forms:

- an equality: $\sum_{i=0}^{n-1} 0x_i + b == 0$ where $b \neq 0$; or
- a congruence: $\sum_{i=0}^{n-1} 0x_i + b\% = 0/m$, where $b \neq 0 \pmod{m}$.

9.6.3.5 bool Parma_Polyhedra_Library::Congruence::is_proper_congruence () const `[inline]`

Returns `true` if the modulus is greater than zero.

A congruence with a modulus of 0 is a linear equality.

9.6.3.6 bool Parma_Polyhedra_Library::Congruence::is_equality () const `[inline]`

Returns `true` if `*this` is an equality.

A modulus of zero denotes a linear equality.

9.6.3.7 void Parma_Polyhedra_Library::Congruence::sign_normalize () `[protected]`

Normalizes the signs.

The signs of the coefficients and the inhomogeneous term are normalized, leaving the first non-zero homogeneous coefficient positive.

9.6.3.8 void Parma_Polyhedra_Library::Congruence::normalize () `[protected]`

Normalizes signs and the inhomogeneous term.

Applies sign_normalize, then reduces the inhomogeneous term to the smallest possible positive number.

9.6.3.9 void Parma_Polyhedra_Library::Congruence::strong_normalize () `[protected]`

Calls normalize, then divides out common factors.

Strongly normalized Congruences have equivalent semantics if and only if their syntaxes (as output by operator<<) are equal.

9.6.4 Friends And Related Function Documentation

9.6.4.1 Congruence operator/ (const Congruence & *cg*, const Coefficient_traits::const_reference *k*) `[related]`

Returns a copy of `cg`, multiplying `k` into the copy's modulus.

If `cg` represents the congruence $e_1 = e_2 \pmod{m}$, then the result represents the congruence $e_1 = e_2 \pmod{mk}$.

9.7 Parma_Polyhedra_Library::Congruence_System Class Reference

A system of congruences.

Public Member Functions

- Congruence_System ()

  Default constructor: builds an empty system of congruences.

- Congruence_System (const Congruence &cg)

  Builds the singleton system containing only congruence `cg`*.*

- Congruence_System (const Constraint &c)

  If `c` *represents the constraint* $e_1 = e_2$*, builds the singleton system containing only constraint* $e_1 = e_2 \pmod{0}$*.*

- Congruence_System (const Constraint_System &cs)

  Builds a system containing copies of any equalities in `cs`*.*

- Congruence_System (const Congruence_System &cgs)

  Ordinary copy-constructor.

- ∼Congruence_System ()

  Destructor.

- Congruence_System & operator= (const Congruence_System &cgs)

  Assignment operator.

- dimension_type space_dimension () const

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

- bool is_equal_to (const Congruence_System &cgs) const

  Returns `true` *if and only if* `*this` *is exactly equal to* `cgs`*.*

- bool has_linear_equalities () const

  Returns `true` *if and only if* `*this` *contains one or more linear equalities.*

- void clear ()

  Removes all the congruences and sets the space dimension to 0.

- void insert (const Congruence &cg)

Inserts in `*this` a copy of the congruence `cg`, increasing the number of space dimensions if needed.

- void insert (const Constraint &c)

  Inserts in `*this` a copy of the equality constraint `c`, seen as a modulo 0 congruence, increasing the number of space dimensions if needed.

- void insert (const Congruence_System &cgs)

  Inserts in `*this` a copy of the congruences in `cgs`, increasing the number of space dimensions if needed.

- void recycling_insert (Congruence_System &cgs)

  Inserts into `*this` the congruences in `cgs`, increasing the number of space dimensions if needed.

- const_iterator begin () const

  Returns the const_iterator pointing to the first congruence, if `this` is not empty; otherwise, returns the past-the-end const_iterator.

- const_iterator end () const

  Returns the past-the-end const_iterator.

- bool OK () const

  Checks if all the invariants are satisfied.

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- dimension_type num_equalities () const

  Returns the number of equalities.

- dimension_type num_proper_congruences () const

  Returns the number of proper congruences.

- void swap (Congruence_System &cgs)

  Swaps `*this` with `y`.

- void add_unit_rows_and_columns (dimension_type dims)

  Adds `dims` rows and `dims` columns of zeroes to the matrix, initializing the added rows as in the unit congruence system.

- void concatenate (const Congruence_System &cgs)

  Concatenates copies of the congruences from `cgs` onto `*this`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Congruence_System can handle.

- static const Congruence_System & zero_dim_empty ()

  Returns the system containing only Congruence::zero_dim_false().

Protected Member Functions

- bool satisfies_all_congruences (const Grid_Generator &g) const

  Returns `true` *if* `g` *satisfies all the congruences.*

Friends

- void swap (Parma_Polyhedra_Library::Congruence_System &x, Parma_Polyhedra_-Library::Congruence_System &y)

  Specializes `std::swap`.

- bool operator== (const Congruence_System &x, const Congruence_System &y)

  Returns `true` *if and only if* `x` *and* `y` *are equivalent.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Congruence_System &cgs)

  Output operator.

Classes

- class const_iterator

  An iterator over a system of congruences.

9.7.1 Detailed Description

A system of congruences.

An object of the class Congruence_System is a system of congruences, i.e., a multiset of objects of the class Congruence. When inserting congruences in a system, space dimensions are automatically adjusted so that all the congruences in the system are defined on the same vector space.

In all the examples it is assumed that variables `x` and `y` are defined as follows:

```
Variable x(0);
Variable y(1);
```

Example 1

The following code builds a system of congruences corresponding to an integer grid in $\mathbb{R}^2$:

```
Congruence_System cgs;
cgs.insert(x %= 0);
cgs.insert(y %= 0);
```

Note that: the congruence system is created with space dimension zero; the first and second congruence insertions increase the space dimension to 1 and 2, respectively.

Example 2

By adding to the congruence system of the previous example, the congruence $x + y = 1 \pmod{2}$:

```
cgs.insert((x + y %= 1) / 2);
```

we obtain the grid containing just those integral points where the sum of the `x` and `y` values is odd.

Example 3

The following code builds a system of congruences corresponding to the grid in $\mathbb{Z}^2$ containing just the integral points on the `x` axis:

```
Congruence_System cgs;
cgs.insert(x %= 0);
cgs.insert((y %= 0) / 0);
```

Note:

After inserting a multiset of congruences in a congruence system, there are no guarantees that an *exact* copy of them can be retrieved: in general, only an *equivalent* congruence system will be available, where original congruences may have been reordered, removed (if they are trivial, duplicate or implied by other congruences), linearly combined, etc.

9.7.2 Constructor & Destructor Documentation

9.7.2.1 Parma_Polyhedra_Library::Congruence_System::Congruence_System (const Constraint & *c*) `[inline, explicit]`

If `c` represents the constraint $e_1 = e_2$, builds the singleton system containing only constraint $e_1 = e_2 \pmod{0}$.

Exceptions:

std::invalid_argument Thrown if `c` is not an equality constraint.

9.7.3 Member Function Documentation

9.7.3.1 void Parma_Polyhedra_Library::Congruence_System::insert (const Congruence & *cg*) `[inline]`

Inserts in `*this` a copy of the congruence `cg`, increasing the number of space dimensions if needed.

The copy of `cg` will be strongly normalized after being inserted.

9.7.3.2 void Parma_Polyhedra_Library::Congruence_System::insert (const Constraint & *c*)

Inserts in `*this` a copy of the equality constraint `c`, seen as a modulo 0 congruence, increasing the number of space dimensions if needed.

The modulo 0 congruence will be strongly normalized after being inserted.

Exceptions:

std::invalid_argument Thrown if `c` is a relation.

9.7.3.3 void Parma_Polyhedra_Library::Congruence_System::insert (const Congruence_System & *cgs*)

Inserts in `*this` a copy of the congruences in `cgs`, increasing the number of space dimensions if needed.

The inserted copies will be strongly normalized.

9.7.3.4 void Parma_Polyhedra_Library::Congruence_System::add_unit_rows_and_columns (dimension_type *dims*)

Adds `dims` rows and `dims` columns of zeroes to the matrix, initializing the added rows as in the unit congruence system.

Parameters:

dims The number of rows and columns to be added: must be strictly positive.

Turns the $r \times c$ matrix A into the $(r + dims) \times (c + dims)$ matrix $\left(\begin{smallmatrix} 0 & B \\ A & A \end{smallmatrix}\right)$ where B is the $dims \times dims$ unit matrix of the form $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. The matrix is expanded avoiding reallocation whenever possible.

9.7.3.5 void Parma_Polyhedra_Library::Congruence_System::concatenate (const Congruence_System & *cgs*)

Concatenates copies of the congruences from `cgs` onto `*this`.

The matrix for the new system of congruences is obtained by leaving the old system in the upper left-hand side and placing the congruences of `cgs` in the lower right-hand side, and padding with zeroes.

9.7.4 Friends And Related Function Documentation

9.7.4.1 std::ostream & operator<< (std::ostream & *s*, const Congruence_System & *cgs*) `[related]`

Output operator.

Writes `true` if `cgs` is empty. Otherwise, writes on `s` the congruences of `cgs`, all in one row and separated by ", ".

9.8 Parma_Polyhedra_Library::Congruence_System::const_iterator Class Reference

An iterator over a system of congruences.

Public Member Functions

- const_iterator ()

  Default constructor.

- const_iterator (const const_iterator &y)

Ordinary copy-constructor.

- ~const_iterator ()

  Destructor.

- const_iterator & operator= (const const_iterator &y)

  Assignment operator.

- const Congruence & operator * () const

  Dereference operator.

- const Congruence * operator → () const

  Indirect member selector.

- const_iterator & operator++ ()

  Prefix increment operator.

- const_iterator operator++ (int)

  Postfix increment operator.

- bool operator== (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are identical.*

- bool operator!= (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are different.*

9.8.1 Detailed Description

An iterator over a system of congruences.

A const_iterator is used to provide read-only access to each congruence contained in an object of Congruence_System.

Example

The following code prints the system of congruences defining the grid `gr`:

```
const Congruence_System& cgs = gr.congruences();
for (Congruence_System::const_iterator i = cgs.begin(),
       cgs_end = cgs.end(); i != cgs_end; ++i)
  cout << *i << endl;
```

9.9 Parma_Polyhedra_Library::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.*

- Coefficient_traits::const_reference coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- Coefficient_traits::const_reference inhomogeneous_term () const

  Returns the inhomogeneous term of `*this`.

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- bool is_tautological () const

  Returns `true` *if and only if* `*this` *is a tautology (i.e., an always true constraint).*

- bool is_inconsistent () const

  Returns `true` *if and only if* `*this` *is inconsistent (i.e., an always false constraint).*

- bool is_equivalent_to (const Constraint &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are equivalent constraints.*

- bool OK () const

  Checks if all the invariants are satisfied.

- void swap (Constraint &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Constraint can handle.

- static const Constraint & zero_dim_false ()

  The unsatisfiable (zero-dimension space) constraint $0 = 1$.

- static const Constraint & zero_dim_positivity ()

  The true (zero-dimension space) constraint $0 \leq 1$, *also known as* positivity constraint.

Friends

- Constraint operator== (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the constraint `e1 = e2`.

- Constraint operator== (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the constraint `e = n`.

- Constraint operator== (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the constraint `n = e`.

- Constraint operator>= (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the constraint `e1 >= e2`.

- Constraint operator>= (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the constraint `e >= n`.

- Constraint operator>= (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the constraint `n >= e`.

- Constraint operator<= (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the constraint `e1 <= e2`.

- Constraint operator<= (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the constraint `e <= n`.

- Constraint operator<= (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the constraint `n <= e`.

- Constraint operator> (const Linear_Expression &e1, const Linear_Expression &e2)

Returns the constraint `e1` $>$ `e2`.

- Constraint operator> (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the constraint `e` $>$ `n`.

- Constraint operator> (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the constraint `n` $>$ `e`.

- Constraint operator< (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the constraint `e1` $<$ `e2`.

- Constraint operator< (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the constraint `e` $<$ `n`.

- Constraint operator< (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the constraint `n` $<$ `e`.

Related Functions

(Note that these are not member functions.)

- bool operator== (const Constraint &x, const Constraint &y)

  Returns `true` *if and only if* `x` *is equivalent to* `y`.

- bool operator!= (const Constraint &x, const Constraint &y)

  Returns `true` *if and only if* `x` *is not equivalent to* `y`.

- void swap (Parma_Polyhedra_Library::Constraint &x, Parma_Polyhedra_Library::Constraint &y)

  Specializes `std::swap`.

- std::ostream & operator<< (std::ostream &s, const Constraint &c)

  Output operator.

- std::ostream & operator<< (std::ostream &s, const Constraint::Type &t)

  Output operator.

9.9.1 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(Linear_Expression::zero() == 1);
Constraint false_c2(Linear_Expression::zero() >= 1);
Constraint false_c3(Linear_Expression::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 $x - 5y + 3z \leq 4$), 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 {
  Linear_Expression 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.

9.9.2 Member Enumeration Documentation

9.9.2.1 enum Parma_Polyhedra_Library::Constraint::Type

The constraint type.

Enumerator:

EQUALITY The constraint is an equality.

NONSTRICT_INEQUALITY The constraint is a non-strict inequality.

STRICT_INEQUALITY The constraint is a strict inequality.

9.9.3 Member Function Documentation

9.9.3.1 Coefficient_traits::const_reference Parma_Polyhedra_Library::Constraint::coefficient (Variable *v*) const [inline]

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`.

9.9.3.2 bool Parma_Polyhedra_Library::Constraint::is_tautological () const

Returns `true` if and only if `*this` is a tautology (i.e., an always true constraint).

A tautology can have either one of the following forms:

- an equality: $\sum_{i=0}^{n-1} 0x_i + 0 = 0$; or
- a non-strict inequality: $\sum_{i=0}^{n-1} 0x_i + b \geq 0$, where $b \geq 0$; or
- a strict inequality: $\sum_{i=0}^{n-1} 0x_i + b > 0$, where $b > 0$.

9.9.3.3 bool Parma_Polyhedra_Library::Constraint::is_inconsistent () const

Returns `true` if and only if `*this` is inconsistent (i.e., an always false constraint).

An inconsistent constraint can have either one of the following forms:

- an equality: $\sum_{i=0}^{n-1} 0x_i + b = 0$, where $b \neq 0$; or
- a non-strict inequality: $\sum_{i=0}^{n-1} 0x_i + b \geq 0$, where $b < 0$; or
- a strict inequality: $\sum_{i=0}^{n-1} 0x_i + b > 0$, where $b \leq 0$.

9.9.3.4 bool Parma_Polyhedra_Library::Constraint::is_equivalent_to (const Constraint & *y*) const

Returns `true` if and only if `*this` and `y` are equivalent constraints.

Constraints having different space dimensions are not equivalent. Note that constraints having different types may nonetheless be equivalent, if they both are tautologies or inconsistent.

9.10 Parma_Polyhedra_Library::Constraint_System Class Reference

A system of constraints.

Public Member Functions

- Constraint_System ()

    Default constructor: builds an empty system of constraints.

- Constraint_System (const Constraint &c)

    Builds the singleton system containing only constraint `c`*.*

- Constraint_System (const Constraint_System &cs)

    Ordinary copy-constructor.

- ∼Constraint_System ()

    Destructor.

- Constraint_System & operator= (const Constraint_System &y)

    Assignment operator.

- dimension_type space_dimension () const

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

- bool has_strict_inequalities () const

    Returns `true` *if and only if* `*this` *contains one or more strict inequality constraints.*

- void clear ()

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

- void insert (const Constraint &c)

Inserts in `*this` *a copy of the constraint* `c`*, increasing the number of space dimensions if needed.*

- const_iterator begin () const

  Returns the const_iterator pointing to the first constraint, if `*this` *is not empty; otherwise, returns the past-the-end const_iterator.*

- const_iterator end () const

  Returns the past-the-end const_iterator.

- bool OK () const

  Checks if all the invariants are satisfied.

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- void swap (Constraint_System &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Constraint_System can handle.

- static const Constraint_System & zero_dim_empty ()

  Returns the singleton system containing only Constraint::zero_dim_false().

Friends

- bool operator== (const Polyhedron &x, const Polyhedron &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same polyhedron.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Constraint_System &cs)

  Output operator.

- void swap (Parma_Polyhedra_Library::Constraint_System &x, Parma_Polyhedra_Library::Constraint_System &y)

  Specializes `std::swap`.

Classes

- class const_iterator

  An iterator over a system of constraints.

9.10.1 Detailed Description

A system of constraints.

An object of the class Constraint_System is a system of constraints, i.e., a multiset of objects of the class Constraint. When inserting constraints in a system, space dimensions are automatically adjusted so that all the constraints in the system are defined on the same vector space.

In all the examples it is assumed that variables x and y are defined as follows:

```
Variable x(0);
Variable y(1);
```

Example 1

The following code builds a system of constraints corresponding to a square in $\mathbb{R}^2$:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 3);
cs.insert(y >= 0);
cs.insert(y <= 3);
```

Note that: the constraint system is created with space dimension zero; the first and third constraint insertions increase the space dimension to 1 and 2, respectively.

Example 2

By adding four strict inequalities to the constraint system of the previous example, we can remove just the four vertices from the square defined above.

```
cs.insert(x + y > 0);
cs.insert(x + y < 6);
cs.insert(x - y < 3);
cs.insert(y - x < 3);
```

Example 3

The following code builds a system of constraints corresponding to a half-strip in $\mathbb{R}^2$:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x - y <= 0);
cs.insert(x - y + 1 >= 0);
```

Note:

After inserting a multiset of constraints in a constraint system, there are no guarantees that an *exact* copy of them can be retrieved: in general, only an *equivalent* constraint system will be available, where original constraints may have been reordered, removed (if they are trivial, duplicate or implied by other constraints), linearly combined, etc.

9.10.2 Friends And Related Function Documentation

9.10.2.1 bool operator== (const Polyhedron & *x*, const Polyhedron & *y*) `[friend]`

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.

9.10.2.2 std::ostream & operator<< (std::ostream & *s*, const Constraint_System & *cs*) `[related]`

Output operator.

Writes `true` if `cs` is empty. Otherwise, writes on `s` the constraints of `cs`, all in one row and separated by ", ".

9.11 Parma_Polyhedra_Library::Constraint_System::const_iterator Class Reference

An iterator over a system of constraints.

Public Member Functions

- const_iterator ()

  Default constructor.

- const_iterator (const const_iterator &y)

  Ordinary copy-constructor.

- ∼const_iterator ()

  Destructor.

- const_iterator & operator= (const const_iterator &y)

  Assignment operator.

- const Constraint & operator ∗ () const

  Dereference operator.

- const Constraint ∗ operator → () const

  Indirect member selector.

- const_iterator & operator++ ()

  Prefix increment operator.

- const_iterator operator++ (int)

  Postfix increment operator.

- bool operator== (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are identical.*

- bool operator!= (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are different.*

9.11.1 Detailed Description

An iterator over a system of constraints.

A const_iterator is used to provide read-only access to each constraint contained in a Constraint_System object.

Example

The following code prints the system of constraints defining the polyhedron `ph`:

```
const Constraint_System& cs = ph.constraints();
for (Constraint_System::const_iterator i = cs.begin(),
       cs_end = cs.end(); i != cs_end; ++i)
  cout << *i << endl;
```

9.12 Parma_Polyhedra_Library::Determinate< PH > Class Template Reference

Wraps a PPL class into a determinate constraint system interface.

Public Member Functions

Constructors and Destructor

- Determinate (const PH &p)

  Injection operator: builds the determinate constraint system element corresponding to the base-level element `p`.

- Determinate (const Constraint_System &cs)

  Injection operator: builds the determinate constraint system element corresponding to the base-level element represented by `cs`.

- Determinate (const Congruence_System &cgs)

  corresponding to the base-level element represented by `cgs`.

- Determinate (const Determinate &y)

  Copy constructor.

- ∼Determinate ()

  Destructor.

Member Functions that Do Not Modify the Domain Element

- const PH & element () const

  Returns a const reference to the embedded element.

- bool is_top () const

  Returns `true` *if and only if* `*this` *is the top of the determinate constraint system (i.e., the whole vector space).*

- bool is_bottom () const

  Returns `true` *if and only if* `*this` *is the bottom of the determinate constraint system.*

- bool definitely_entails (const Determinate &y) const

  Returns `true` *if and only if* `*this` *entails* `y`.

- bool is_definitely_equivalent_to (const Determinate &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are equivalent.*

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns a lower bound to the size in bytes of the memory managed by `*this`.

- bool OK () const

  Checks if all the invariants are satisfied.

Member Functions that May Modify the Domain Element

- void upper_bound_assign (const Determinate &y)

  Assigns to `*this` *the upper bound of* `*this` *and* `y`.

- void meet_assign (const Determinate &y)

  Assigns to `*this` *the meet of* `*this` *and* `y`.

- void concatenate_assign (const Determinate &y)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`, *taken in this order.*

- PH & element ()

  Returns a reference to the embedded element.

- Determinate & operator= (const Determinate &y)

  Assignment operator.

- void swap (Determinate &y)

  Swaps `*this` *with* `y`.

Friends

- bool operator== (const Determinate< PH > &x, const Determinate< PH > &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same domain element.*

- bool operator!= (const Determinate< PH > &x, const Determinate< PH > &y)

  Returns `true` *if and only if* `x` *and* `y` *are different domain elements.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &, const Determinate< PH > &)

  Output operator.

- void swap (Parma_Polyhedra_Library::Determinate< PH > &x, Parma_Polyhedra_-Library::Determinate< PH > &y)

  Specializes `std::swap`.

9.12.1 Detailed Description

template<typename PH> class Parma_Polyhedra_Library::Determinate< PH >

Wraps a PPL class into a determinate constraint system interface.

9.13 Parma_Polyhedra_Library::From_Bounding_Box Struct Reference

A tag class.

9.13.1 Detailed Description

A tag class.

Tag class to differentiate the C_Polyhedron and NNC_Polyhedron constructors that build a polyhedron out of a bounding box.

9.14 Parma_Polyhedra_Library::From_Covering_Box Struct Reference

A tag class.

9.14.1 Detailed Description

A tag class.

Tag class to make the Grid covering box constructor unique.

9.15 Parma_Polyhedra_Library::Generator Class Reference

A line, ray, point or closure point.

Inherited by Parma_Polyhedra_Library::Grid_Generator `[private]`.

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.*

- Coefficient_traits::const_reference coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- Coefficient_traits::const_reference divisor () const

  If `*this` *is either a point or a closure point, returns its divisor.*

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- bool is_equivalent_to (const Generator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are equivalent generators.*

- bool OK () const

  Checks if all the invariants are satisfied.

- void swap (Generator &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static Generator line (const Linear_Expression &e)

  Shorthand for Generator Generator::line(const Linear_Expression& e).

- static Generator ray (const Linear_Expression &e)

  Shorthand for Generator Generator::ray(const Linear_Expression& e).

- static Generator point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_-traits::const_reference d=Coefficient_one())

  Shorthand for Generator Generator::point(const Linear_Expression& e, Coefficient_traits::const_-reference d).

- static Generator closure_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one())

  Shorthand for Generator Generator::closure_point(const Linear_Expression& e, Coefficient_-traits::const_reference d).

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Generator can handle.

- static const Generator & zero_dim_point ()

  Returns the origin of the zero-dimensional space $\mathbb{R}^0$.

- static const Generator & zero_dim_closure_point ()

  Returns, as a closure point, the origin of the zero-dimensional space $\mathbb{R}^0$.

Friends

- std::ostream & operator<< (std::ostream &s, const Generator &g)

  Output operator.

Related Functions

(Note that these are not member functions.)

- void swap (Parma_Polyhedra_Library::Generator &x, Parma_Polyhedra_Library::Generator &y)

  Specializes `std::swap`.

- bool operator== (const Generator &x, const Generator &y)

  Returns `true` *if and only if* `x` *is equivalent to* `y`.

- bool operator!= (const Generator &x, const Generator &y)

  Returns `true` *if and only if* `x` *is not equivalent to* `y`.

- std::ostream & operator<< (std::ostream &s, const Generator::Type &t)

  Output operator.

9.15.1 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 Coefficient 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

Closure 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;
  Linear_Expression 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.

9.15.2 Member Enumeration Documentation

9.15.2.1 enum Parma_Polyhedra_Library::Generator::Type

The generator type.

Enumerator:

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.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator.

9.15.3 Member Function Documentation

9.15.3.1 Generator line (const Linear_Expression & *e*) `[inline, static]`

Shorthand for Generator Generator::line(const Linear_Expression& e).

Exceptions:

std::invalid_argument Thrown if the homogeneous part of `e` represents the origin of the vector space.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator.

9.15.3.2 Generator ray (const Linear_Expression & *e*) `[inline, static]`

Shorthand for Generator Generator::ray(const Linear_Expression& e).

Exceptions:

std::invalid_argument Thrown if the homogeneous part of `e` represents the origin of the vector space.

9.15.3.3 Generator point (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Shorthand for Generator Generator::point(const Linear_Expression& e, Coefficient_traits::const_reference d).

Both `e` and `d` are optional arguments, with default values Linear_Expression::zero() and Coefficient_one(), respectively.

Exceptions:

std::invalid_argument Thrown if `d` is zero.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator.

9.15.3.4 Generator closure_point (const Linear_Expression & *e* = `Linear_-Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Shorthand for Generator Generator::closure_point(const Linear_Expression& e, Coefficient_traits::const_-reference d).

Both `e` and `d` are optional arguments, with default values Linear_Expression::zero() and Coefficient_one(), respectively.

Exceptions:

std::invalid_argument Thrown if `d` is zero.

9.15.3.5 Coefficient_traits::const_reference Parma_Polyhedra_Library::Generator::coefficient (Variable *v*) const `[inline]`

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`.

9.15.3.6 Coefficient_traits::const_reference Parma_Polyhedra_Library::Generator::divisor () const `[inline]`

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.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator.

9.15.3.7 bool Parma_Polyhedra_Library::Generator::is_equivalent_to (const Generator & *y*) const

Returns `true` if and only if `*this` and `y` are equivalent generators.

Generators having different space dimensions are not equivalent.

9.16 Parma_Polyhedra_Library::Generator_System Class Reference

A system of generators.

Inherited by Parma_Polyhedra_Library::Grid_Generator_System `[private]`.

Public Member Functions

- Generator_System ()

  Default constructor: builds an empty system of generators.

- Generator_System (const Generator &g)

  Builds the singleton system containing only generator `g`.

- Generator_System (const Generator_System &gs)

  Ordinary copy-constructor.

- ∼Generator_System ()

  Destructor.

- Generator_System & operator= (const Generator_System &y)

  Assignment operator.

- dimension_type space_dimension () const

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

- void clear ()

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

- void insert (const Generator &g)

  Inserts in `*this` *a copy of the generator* `g`*, increasing the number of space dimensions if needed.*

- const_iterator begin () const

  Returns the const_iterator pointing to the first generator, if `*this` *is not empty; otherwise, returns the past-the-end const_iterator.*

- const_iterator end () const

  Returns the past-the-end const_iterator.

- bool OK () const

  Checks if all the invariants are satisfied.

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- void swap (Generator_System &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Generator_System can handle.

- static const Generator_System & zero_dim_univ ()

  Returns the singleton system containing only Generator::zero_dim_point().

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Generator_System &gs)

  Output operator.

- void swap (Parma_Polyhedra_Library::Generator_System &x, Parma_Polyhedra_-Library::Generator_System &y)

  Specializes `std::swap`.

Classes

- class const_iterator

  An iterator over a system of generators.

9.16.1 Detailed Description

A system of generators.

An object of the class Generator_System is a system of generators, i.e., a multiset of objects of the class Generator (lines, rays, points and closure points). When inserting generators in a system, space dimensions are automatically adjusted so that all the generators in the system are defined on the same vector space. A system of generators which is meant to define a non-empty polyhedron must include at least one point: the reason is that lines, rays and closure points need a supporting point (lines and rays only specify directions while closure points only specify points in the topological closure of the NNC polyhedron).

In all the examples it is assumed that variables `x` and `y` are defined as follows:

```
Variable x(0);
Variable y(1);
```

Example 1

The following code defines the line having the same direction as the x axis (i.e., the first Cartesian axis) in $\mathbb{R}^2$:

```
Generator_System gs;
gs.insert(line(x + 0*y));
```

As said above, this system of generators corresponds to an empty polyhedron, because the line has no supporting point. To define a system of generators that does correspond to the x axis, we can add the following code which inserts the origin of the space as a point:

```
gs.insert(point(0*x + 0*y));
```

Since space dimensions are automatically adjusted, the following code obtains the same effect:

```
gs.insert(point(0*x));
```

In contrast, if we had added the following code, we would have defined a line parallel to the x axis through the point $(0,1)^{\mathrm{T}} \in \mathbb{R}^2$.

```
gs.insert(point(0*x + 1*y));
```

Example 2

The following code builds a ray having the same direction as the positive part of the x axis in $\mathbb{R}^2$:

```
Generator_System gs;
gs.insert(ray(x + 0*y));
```

To define a system of generators indeed corresponding to the set

$$\bigl\{\, (x,0)^{\mathrm{T}} \in \mathbb{R}^2 \bigm| x \geq 0 \,\bigr\},$$

one just has to add the origin:

```
gs.insert(point(0*x + 0*y));
```

Example 3

The following code builds a system of generators having four points and corresponding to a square in $\mathbb{R}^2$ (the same as Example 1 for the system of constraints):

```
Generator_System gs;
gs.insert(point(0*x + 0*y));
gs.insert(point(0*x + 3*y));
gs.insert(point(3*x + 0*y));
gs.insert(point(3*x + 3*y));
```

Example 4

By using closure points, we can define the *kernel* (i.e., the largest open set included in a given set) of the square defined in the previous example. Note that a supporting point is needed and, for that purpose, any inner point could be considered.

```
Generator_System gs;
gs.insert(point(x + y));
gs.insert(closure_point(0*x + 0*y));
gs.insert(closure_point(0*x + 3*y));
gs.insert(closure_point(3*x + 0*y));
gs.insert(closure_point(3*x + 3*y));
```

Example 5

The following code builds a system of generators having two points and a ray, corresponding to a half-strip in $\mathbb{R}^2$ (the same as Example 2 for the system of constraints):

```
Generator_System gs;
gs.insert(point(0*x + 0*y));
gs.insert(point(0*x + 1*y));
gs.insert(ray(x - y));
```

Note:

After inserting a multiset of generators in a generator system, there are no guarantees that an *exact* copy of them can be retrieved: in general, only an *equivalent* generator system will be available, where original generators may have been reordered, removed (if they are duplicate or redundant), etc.

9.16.2 Member Function Documentation

9.16.2.1 bool Parma_Polyhedra_Library::Generator_System::OK () const

Checks if all the invariants are satisfied.

Returns `true` if and only if `*this` is a valid Linear_System and each row in the system is a valid Generator.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator_System.

9.16.3 Friends And Related Function Documentation

9.16.3.1 std::ostream & operator<< (std::ostream & *s*, const Generator_System & *gs*) `[related]`

Output operator.

Writes `false` if `gs` is empty. Otherwise, writes on `s` the generators of `gs`, all in one row and separated by ", ".

9.17 Parma_Polyhedra_Library::Generator_System::const_iterator Class Reference

An iterator over a system of generators.

Inherited by Parma_Polyhedra_Library::Grid_Generator_System::const_iterator `[private]`.

Public Member Functions

- const_iterator ()

  Default constructor.

- const_iterator (const const_iterator &y)

  Ordinary copy-constructor.

- ~const_iterator ()

  Destructor.

- const_iterator & operator= (const const_iterator &y)

  Assignment operator.

- const Generator & operator $*$ () const

  Dereference operator.

- const Generator $*$ operator $\rightarrow$ () const

  Indirect member selector.

- const_iterator & operator++ ()

  Prefix increment operator.

- const_iterator operator++ (int)

  Postfix increment operator.

- bool operator== (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are identical.*

- bool operator!= (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are different.*

9.17.1 Detailed Description

An iterator over a system of generators.

A const_iterator is used to provide read-only access to each generator contained in an object of Generator_-System.

Example

The following code prints the system of generators of the polyhedron `ph`:

```
const Generator_System& gs = ph.generators();
for (Generator_System::const_iterator i = gs.begin(),
       gs_end = gs.end(); i != gs_end; ++i)
  cout << *i << endl;
```

The same effect can be obtained more concisely by using more features of the STL:

```
const Generator_System& gs = ph.generators();
copy(gs.begin(), gs.end(), ostream_iterator<Generator>(cout, "\n"));
```

9.18 Parma_Polyhedra_Library::GMP_Integer Class Reference

Unbounded integers as provided by the GMP library.

Related Functions

(Note that these are not member functions.)

- void swap (Parma_Polyhedra_Library::GMP_Integer &x, Parma_Polyhedra_Library::GMP_Integer &y)

Specializes `std::swap`.

Accessor Functions

- const mpz_class & raw_value (const GMP_Integer &x)

  Returns a const reference to the underlying integer value.

- mpz_class & raw_value (GMP_Integer &x)

  Returns a reference to the underlying integer value.

Memory Size Inspection Functions

- memory_size_type total_memory_in_bytes (const GMP_Integer &x)

  Returns the total size in bytes of the memory occupied by `x`*.*

- memory_size_type external_memory_in_bytes (const GMP_Integer &x)

  Returns the size in bytes of the memory managed by `x`*.*

Arithmetic Operators

- void neg_assign (GMP_Integer &x)

  Assigns to `x` *its negation.*

- void neg_assign (GMP_Integer &x, const GMP_Integer &y)

  Assigns to `x` *the negation of* `y`*.*

- void gcd_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)

  Assigns to `x` *the greatest common divisor of* `y` *and* `z`*.*

- void gcdext_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z, GMP_-Integer &s, GMP_Integer &t)

  Extended GCD.

- void lcm_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)

  Assigns to `x` *the least common multiple of* `y` *and* `z`*.*

- void add_mul_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)

  Assigns to `x` *the value* `x + y * z`*.*

- void sub_mul_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)

  Assigns to `x` *the value* `x - y * z`*.*

- void exact_div_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)

  If `z` *divides* `y`*, assigns to* `x` *the quotient of the integer division of* `y` *and* `z`*.*

- void sqrt_assign (GMP_Integer &x, const GMP_Integer &y)

  Assigns to `x` *the integer square root of* `y`*.*

- int cmp (const GMP_Integer &x, const GMP_Integer &y)

  Returns a negative, zero or positive value depending on whether `x` *is lower than, equal to or greater than* `y`*, respectively.*

9.18.1 Detailed Description

Unbounded integers as provided by the GMP library.

GMP_Integer is an alias for the `mpz_class` type defined in the C++ interface of the GMP library. For more information, see `http://www.swox.com/gmp/`

9.18.2 Friends And Related Function Documentation

9.18.2.1 void gcdext_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*, GMP_Integer & *s*, GMP_Integer & *t*) `[related]`

Extended GCD.

Assigns to `x` the greatest common divisor of `y` and `z`, and to `s` and `t` the values such that `y` ∗ `s` + `z` ∗ `t` = `x`.

9.18.2.2 void exact_div_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) `[related]`

If `z` divides `y`, assigns to `x` the quotient of the integer division of `y` and `z`.

The behavior is undefined if `z` does not divide `y`.

9.19 Parma_Polyhedra_Library::Grid Class Reference

A grid.

Public Member Functions

- Grid (dimension_type num_dimensions=0, const Degenerate_Element kind=UNIVERSE)

  Builds a grid having the specified properties.

- Grid (const Congruence_System &cgs)

  Builds a grid, copying a system of congruences.

- Grid (Congruence_System &cgs)

  Builds a grid, recycling a system of congruences.

- Grid (const Constraint_System &cs)

  Builds a grid, copying a system of constraints.

- Grid (Constraint_System &cs)

  Builds a grid, recycling a system of constraints.

- Grid (const Grid_Generator_System &const_gs)

  Builds a grid, copying a system of generators.

- Grid (Grid_Generator_System &gs)

  Builds a grid, recycling a system of generators.

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

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

- template<typename Box> Grid (const Box &box, From_Covering_Box dummy)

  Builds a grid out of a generic, interval-based covering box.

- Grid (const Grid &y)

  Ordinary copy-constructor.

- Grid & operator= (const Grid &y)

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

Member Functions that Do Not Modify the Grid

- dimension_type space_dimension () const

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

- dimension_type affine_dimension () const

  Returns 0*, if `*this` is empty; otherwise, returns the affine dimension of `*this`.*

- const Congruence_System & congruences () const

  Returns the system of congruences.

- const Congruence_System & minimized_congruences () const

  Returns the system of congruences in reduced form.

- const Grid_Generator_System & generators () const

  Returns the system of generators.

- const Grid_Generator_System & minimized_generators () const

  Returns the minimized system of generators.

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between `*this` and `cg`.

- Poly_Gen_Relation relation_with (const Grid_Generator &g) const

  Returns the relations holding between `*this` and `g`.

- bool is_empty () const

  Returns `true` if and only if `*this` is an empty grid.

- bool is_universe () const

  Returns `true` if and only if `*this` is a universe grid.

- 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 Grid &y) const

  Returns `true` if and only if `*this` and `y` are disjoint.

- bool is_discrete () const

  Returns `true` if and only if `*this` is discrete.

- bool is_bounded () const

  Returns `true` *if and only if* `*this` *is bounded.*

- bool bounds_from_above (const Linear_Expression &expr) const

  Returns `true` *if and only if* `expr` *is bounded in* `*this`.

- bool bounds_from_below (const Linear_Expression &expr) const

  Returns `true` *if and only if* `expr` *is bounded in* `*this`.

- bool maximize (const Linear_Expression &expr, Coefficient &sup_n, Coefficient &sup_d, bool &maximum) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from above in* `*this`, *in which case the supremum value is computed.*

- bool maximize (const Linear_Expression &expr, Coefficient &sup_n, Coefficient &sup_d, bool &maximum, Grid_Generator &point) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from above in* `*this`, *in which case the supremum value and a point where* `expr` *reaches it are computed.*

- bool minimize (const Linear_Expression &expr, Coefficient &inf_n, Coefficient &inf_d, bool &minimum) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from below in* `*this`, *in which case the infimum value is computed.*

- bool minimize (const Linear_Expression &expr, Coefficient &inf_n, Coefficient &inf_d, bool &minimum, Grid_Generator &point) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from below in* `*this`, *in which case the infimum value and a point where* `expr` *reaches it are computed.*

- bool contains (const Grid &y) const

  Returns `true` *if and only if* `*this` *contains* `y`.

- bool strictly_contains (const Grid &y) const

  Returns `true` *if and only if* `*this` *strictly contains* `y`.

- template<typename Box> void shrink_bounding_box (Box &box) const

  Uses `*this` *to shrink a generic, interval-based bounding box.*

- template<typename Box> void get_covering_box (Box &box) const

  Writes the covering box for `*this` *into* `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 Grid

- void add_congruence (const Congruence &cg)

  Adds a copy of congruence `cg` *to* `*this`.

- void add_congruence (const Constraint &c)

  Adds constraint `c` *to* `*this`.

- bool add_congruence_and_minimize (const Congruence &c)

  Adds a copy of congruence `cg` *to the system of congruences of* `this`, *reducing the result.*

- bool add_congruence_and_minimize (const Constraint &c)

  Adds a copy of constraint `c` *to* `*this`*, reducing the result.*

- void add_generator (const Grid_Generator &g)

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

- bool add_generator_and_minimize (const Grid_Generator &g)

  Adds a copy of generator `g` *to the system of generators of* `this`*, reducing the result.*

- void add_congruences (const Congruence_System &cgs)

  Adds a copy of each congruence in `cgs` *to* `*this`.

- void add_congruences (const Constraint_System &cs)

  Adds a copy of each equality constraint in `cs` *to* `*this`.

- void add_recycled_congruences (Congruence_System &cgs)

  Adds the congruences in `cgs` *to *this.*

- void add_recycled_congruences (Constraint_System &cs)

  Adds the equality constraints in `cs` *to* `*this`.

- bool add_congruences_and_minimize (const Congruence_System &cgs)

  Adds a copy of the congruences in `cgs` *to the system of congruences of* `*this`*, reducing the result.*

- bool add_congruences_and_minimize (const Constraint_System &cs)

  Adds a copy of each equality constraint in `cs` *to* `*this`*, reducing the result.*

- bool add_recycled_congruences_and_minimize (Congruence_System &cgs)

  Adds the congruences in `cgs` *to the system of congruences of* `this`*, reducing the result.*

- bool add_recycled_congruences_and_minimize (Constraint_System &cs)

  Adds the equalities in `cs` *to* `*this`*, reducing the result.*

- void add_constraint (const Constraint &c)

  Adds constraint `c` *to* `*this`.

- bool add_constraint_and_minimize (const Constraint &c)

  Adds constraint `c` *to* `*this`*, reducing the result.*

- void add_constraints (const Constraint_System &cs)

  Adds copies of the equality constraints in `cs` *to* `*this`.

- bool add_constraints_and_minimize (const Constraint_System &cs)

  Adds copies of the equality constraints in `cs` *to* `*this`*, reducing the result.*

- void add_recycled_constraints (Constraint_System &cs)

  Adds the equality constraints in `cs` *to* `*this`.

- bool add_recycled_constraints_and_minimize (Constraint_System &cs)
- void add_generators (const Grid_Generator_System &gs)

  Adds a copy of the generators in `gs` *to the system of generators of* `*this`.

- void add_recycled_generators (Grid_Generator_System &gs)

  Adds the generators in `gs` *to the system of generators of* `this`.

- bool add_generators_and_minimize (const Grid_Generator_System &gs)

  Adds a copy of the generators in `gs` *to the system of generators of* `*this`*, reducing the result.*

- bool add_recycled_generators_and_minimize (Grid_Generator_System &gs)

  Adds the generators in `gs` *to the system of generators of* `this`*, reducing the result.*

- void intersection_assign (const Grid &y)

  Assigns to `*this` *the intersection of* `*this` *and* `y`*. The result is not guaranteed to be reduced.*

- bool intersection_assign_and_minimize (const Grid &y)

  Assigns to `*this` *the intersection of* `*this` *and* `y`*, reducing the result.*

- void join_assign (const Grid &y)

  Assigns to `*this` *the join of* `*this` *and* `y`*.*

- bool join_assign_and_minimize (const Grid &y)

  Assigns to `*this` *the join of* `*this` *and* `y`*, reducing the result.*

- void upper_bound_assign (const Grid &y)

  Same as join_assign(y).

- bool join_assign_if_exact (const Grid &y)

  If the join of `*this` *and* `y` *is exact it is assigned to* `this` *and* `true` *is returned, otherwise* `false` *is returned.*

- bool upper_bound_assign_if_exact (const Grid &y)

  Same as join_assign_if_exact(y).

- void grid_difference_assign (const Grid &y)

  Assigns to `*this` *the grid-difference of* `*this` *and* `y`*.*

- void difference_assign (const Grid &y)

  Same as grid_difference_assign(y).

- void affine_image (Variable var, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

  Assigns to `*this` *the affine image of* `this` *under the function mapping variable* `var` *to the affine expression specified by* `expr` *and* `denominator`*.*

- void affine_preimage (Variable var, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

  Assigns to `*this` *the affine preimage of* `*this` *under the function mapping variable* `var` *to the affine expression specified by* `expr` *and* `denominator`*.*

- void generalized_affine_image (Variable var, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one(), Coefficient_traits::const_reference modulus=Coefficient_one())

  Assigns to `*this` *the image of* `*this` *with respect to the generalized affine relation* $\mathrm{var}' = \frac{\mathrm{expr}}{\mathrm{denominator}} \pmod{\mathrm{modulus}}$.

- void generalized_affine_preimage (Variable var, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one(), Coefficient_traits::const_reference modulus=Coefficient_one())

  Assigns to `*this` *the preimage of* `*this` *with respect to the generalized affine relation* $\mathrm{var}' = \frac{\mathrm{expr}}{\mathrm{denominator}} \pmod{\mathrm{modulus}}$.

- void generalized_affine_image (const Linear_Expression &lhs, const Linear_Expression &rhs, Coefficient_traits::const_reference modulus=Coefficient_one())

  Assigns to `*this` *the image of* `*this` *with respect to the generalized affine relation* $\mathrm{lhs}' = \mathrm{rhs} \pmod{\mathrm{modulus}}$.

- void generalized_affine_preimage (const Linear_Expression &lhs, const Linear_Expression &rhs, Coefficient_traits::const_reference modulus=Coefficient_one())

  Assigns to `*this` *the preimage of* `*this` *with respect to the generalized affine relation* $\mathrm{lhs}' = \mathrm{rhs} \pmod{\mathrm{modulus}}$.

- void time_elapse_assign (const Grid &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 widening_assign (const Grid &y, unsigned *tp=NULL)

  Assigns to `*this` *the result of computing the Grid widening between* `*this` *and* `y`.

- void limited_extrapolation_assign (const Grid &y, const Congruence_System &cgs, unsigned *tp=NULL)

  Improves the result of the Grid widening computation by also enforcing those congruences in `cgs` *that are satisfied by all the points of* `*this`.

Member Functions that May Modify the Dimension of the Vector Space

- void add_space_dimensions_and_embed (dimension_type m)

  Adds `m` *new space dimensions and embeds the old grid in the new vector space.*

- void add_space_dimensions_and_project (dimension_type m)

  Adds `m` *new space dimensions to the grid and does not embed it in the new vector space.*

- void concatenate_assign (const Grid &y)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`*, taken in this order.*

- void remove_space_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified dimensions from the vector space.

- void remove_higher_space_dimensions (dimension_type new_dimension)

  Removes the higher dimensions of the vector space so that the resulting space will have dimension `new_dimension`.

- template<typename Partial_Function> void map_space_dimensions (const Partial_Function &pfunc)

  Remaps the dimensions of the vector space according to a partial function.

- void expand_space_dimension (Variable var, dimension_type m)

  Creates `m` *copies of the space dimension corresponding to* `var`.

- void fold_space_dimensions (const Variables_Set &to_be_folded, Variable var)

  Folds the space dimensions in `to_be_folded` *into* `var`.

Miscellaneous Member Functions

- ∼Grid ()

  Destructor.

- void swap (Grid &y)

  Swaps `*this` *with grid* `y`. *(*`*this` *and* `y` *can be dimension-incompatible.).*

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension all kinds of Grid can handle.

Friends

- bool operator== (const Grid &x, const Grid &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same grid.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Grid &gr)

  Output operator.

- bool operator!= (const Grid &x, const Grid &y)

  Returns `true` *if and only if* `x` *and* `y` *are different grids.*

- void swap (Parma_Polyhedra_Library::Grid &x, Parma_Polyhedra_Library::Grid &y)

  Specializes `std::swap`.

9.19.1 Detailed Description

A grid.

An object of the class Grid represents a rational grid.

A grid can be specified as either a finite system of congruences or a finite system of generators (see Section Rational Grids) and it is always possible to obtain either representation. That is, if we know the system of congruences, we can obtain from this the system of generators that define the same grid and vice versa. These systems can contain redundant members, or they can be in the minimal form. Most operators on grids 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 grid turns out to be empty.

A key attributes of any grid is its space dimension (the dimension $n \in \mathbb{N}$ of the enclosing vector space):

- all grids, the empty ones included, are endowed with a space dimension;
- most operations working on a grid and another object (another grid, a congruence, a generator, a set of variables, etc.) will throw an exception if the grid and the object are not dimension-compatible (see Section Space Dimensions and Dimension-compatibility for Grids);
- the only ways in which the space dimension of a grid can be changed are with *explicit* calls to operators provided for that purpose, and with standard copy, assignment and swap operators.

Note that two different grids can be defined on the zero-dimension space: the empty grid and the universe grid R^0.

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 grid corresponding to the even integer pairs in $\mathbb{R}^2$, given as a system of congruences:

```
Congruence_System cgs;
cgs.insert((x %= 0) / 2);
cgs.insert((y %= 0) / 2);
Grid gr(cgs);
```

The following code builds the same grid as above, but starting from a system of generators specifying three of the points:

```
Grid_Generator_System gs;
gs.insert(grid_point(0*x + 0*y));
gs.insert(grid_point(0*x + 2*y));
gs.insert(grid_point(2*x + 0*y));
Grid gr(gs);
```

Example 2

The following code builds a grid corresponding to a line in $\mathbb{R}^2$ by adding a single congruence to the universe grid:

```
Congruence_System cgs;
cgs.insert(x - y == 0);
Grid gr(cgs);
```

The following code builds the same grid as above, but starting from a system of generators specifying a point and a line:

```
Grid_Generator_System gs;
gs.insert(grid_point(0*x + 0*y));
gs.insert(grid_line(x + y));
Grid gr(gs);
```

Example 3

The following code builds a grid corresponding to the integral points on the line $x = y$ in $\mathbb{R}^2$ constructed by adding an equality and congruence to the universe grid:

```
Congruence_System cgs;
cgs.insert(x - y == 0);
cgs.insert(x %= 0);
Grid gr(cgs);
```

The following code builds the same grid as above, but starting from a system of generators specifying a point and a parameter:

```
Grid_Generator_System gs;
gs.insert(grid_point(0*x + 0*y));
gs.insert(parameter(x + y));
Grid gr(gs);
```

Example 4

The following code builds the grid corresponding to a plane by creating the universe grid in $\mathbb{R}^2$:

```
Grid gr(2);
```

The following code builds the same grid as above, but starting from the empty grid in $\mathbb{R}^2$ and inserting the appropriate generators (a point, and two lines).

```
Grid gr(2, EMPTY);
gr.add_generator(grid_point(0*x + 0*y));
gr.add_generator(grid_line(x));
gr.add_generator(grid_line(y));
```

Note that a generator system must contain a point when describing a grid. To ensure that this is always the case it is required that the first generator inserted in an empty grid is a point (otherwise, an exception is thrown).

Example 5

The following code shows the use of the function `add_space_dimensions_and_embed`:

```
Grid gr(1);
gr.add_congruence(x == 2);
gr.add_space_dimensions_and_embed(1);
```

We build the universe grid in the 1-dimension space $\mathbb{R}$. Then we add a single equality congruence, thus obtaining the grid corresponding to the singleton set $\{2\} \subseteq \mathbb{R}$. After the last line of code, the resulting grid is

$$\bigl\{\, (2, y)^{\mathrm{T}} \in \mathbb{R}^2 \bigm| y \in \mathbb{R} \,\bigr\}.$$

Example 6

The following code shows the use of the function `add_space_dimensions_and_project`:

```
Grid gr(1);
gr.add_congruence(x == 2);
gr.add_space_dimensions_and_project(1);
```

The first two lines of code are the same as in Example 4 for `add_space_dimensions_and_-embed`. After the last line of code, the resulting grid is the singleton set $\bigl\{(2, 0)^{\mathrm{T}}\bigr\} \subseteq \mathbb{R}^2$.

Example 7

The following code shows the use of the function `affine_image`:

```
Grid gr(2, EMPTY);
gr.add_generator(grid_point(0*x + 0*y));
gr.add_generator(grid_point(4*x + 0*y));
gr.add_generator(grid_point(0*x + 2*y));
Linear_Expression expr = x + 3;
gr.affine_image(x, expr);
```

In this example the starting grid is all the pairs of x and y in $\mathbb{R}^2$ where x is an integer multiple of 4 and y is an integer multiple of 2. The considered variable is x and the affine expression is $x + 3$. The resulting grid is the given grid translated 3 integers to the right (all the pairs (x, y) where x is -1 plus an integer multiple of 4 and y is an integer multiple of 2). Moreover, if the affine transformation for the same variable `x` is instead $x + y$:

```
Linear_Expression expr = x + y;
```

the resulting grid is every second integral point along the $x = y$ line, with this line of points repeated at every fourth integral value along the x axis. Instead, if we do not use an invertible transformation for the same variable; for example, the affine expression y:

```
Linear_Expression expr = y;
```

the resulting grid is every second point along the $x = y$ line.

Example 8

The following code shows the use of the function `affine_preimage`:

```
Grid gr(2, EMPTY);
gr.add_generator(grid_point(0*x + 0*y));
gr.add_generator(grid_point(4*x + 0*y));
gr.add_generator(grid_point(0*x + 2*y));
Linear_Expression expr = x + 3;
gr.affine_preimage(x, expr);
```

In this example the starting grid, `var` and the affine expression and the denominator are the same as in Example 6, while the resulting grid is similar but translated 3 integers to the left (all the pairs (x, y) where x is -3 plus an integer multiple of 4 and y is an integer multiple of 2).. Moreover, if the affine transformation for `x` is $x + y$

```
Linear_Expression expr = x + y;
```

the resulting grid is a similar grid to the result in Example 6, only the grid is slanted along $x = -y$. Instead, if we do not use an invertible transformation for the same variable `x`, for example, the affine expression y:

```
Linear_Expression expr = y;
```

the resulting grid is every fourth line parallel to the x axis.

Example 9

For this example we also use the variables:

```
Variable z(2);
Variable w(3);
```

The following code shows the use of the function `remove_space_dimensions`:

```
Grid_Generator_System gs;
gs.insert(grid_point(3*x + y +0*z + 2*w));
Grid gr(gs);
Variables_Set to_be_removed;
to_be_removed.insert(y);
to_be_removed.insert(z);
gr.remove_space_dimensions(to_be_removed);
```

The starting grid is the singleton set $\left\{(3,1,0,2)^{\mathrm{T}}\right\} \subseteq \mathbb{R}^4$, while the resulting grid is $\left\{(3,2)^{\mathrm{T}}\right\} \subseteq \mathbb{R}^2$. Be careful when removing space dimensions *incrementally*: since dimensions are automatically renamed after each application of the `remove_space_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);
gr.remove_space_dimensions(to_be_removed1);
set<Variable> to_be_removed2;
to_be_removed2.insert(z);
gr.remove_space_dimensions(to_be_removed2);
```

In this case, the result is the grid $\left\{(3,0)^{\mathrm{T}}\right\} \subseteq \mathbb{R}^2$: when removing the set of dimensions `to_-be_removed2` we are actually removing variable w of the original grid. For the same reason, the operator `remove_space_dimensions` is not idempotent: removing twice the same non-empty set of dimensions is never the same as removing them just once.

9.19.2 Constructor & Destructor Documentation

9.19.2.1 Parma_Polyhedra_Library::Grid::Grid (dimension_type *num_dimensions* = `0`, const Degenerate_Element *kind* = `UNIVERSE`) `[explicit]`

Builds a grid having the specified properties.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the grid;

kind Specifies whether the universe or the empty grid has to be built.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.2 Parma_Polyhedra_Library::Grid::Grid (const Congruence_System & *cgs*) `[inline, explicit]`

Builds a grid, copying a system of congruences.

The grid inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the grid.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.3 Parma_Polyhedra_Library::Grid::Grid (Congruence_System & *cgs*) `[inline, explicit]`

Builds a grid, recycling a system of congruences.

The grid inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the grid. Its data-structures will be recycled to build the grid.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.4 Parma_Polyhedra_Library::Grid::Grid (const Constraint_System & *cs*) `[explicit]`

Builds a grid, copying a system of constraints.

The grid inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the grid.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.5 Parma_Polyhedra_Library::Grid::Grid (Constraint_System & *cs*) `[explicit]`

Builds a grid, recycling a system of constraints.

The grid inherits the space dimension of the constraint system.

Parameters:

cs The system of constraints defining the grid.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.6 Parma_Polyhedra_Library::Grid::Grid (const Grid_Generator_System & *const_gs*) `[inline, explicit]`

Builds a grid, copying a system of generators.

The grid inherits the space dimension of the generator system.

Parameters:

const_gs The system of generators defining the grid.

Exceptions:

std::invalid_argument Thrown if the system of generators is not empty but has no points.

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.7 Parma_Polyhedra_Library::Grid::Grid (Grid_Generator_System & *gs*) `[inline, explicit]`

Builds a grid, recycling a system of generators.

The grid inherits the space dimension of the generator system.

Parameters:

gs The system of generators defining the grid. Its data-structures will be recycled to build the grid.

Exceptions:

std::invalid_argument Thrown if the system of generators is not empty but has no points.

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

9.19.2.8 template<typename Box> Parma_Polyhedra_Library::Grid::Grid (const Box & *box*, From_Bounding_Box *dummy*)

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

Parameters:

box The bounding box representing the grid to be built. The box can contain only point and universe intervals;

dummy A dummy tag to make this constructor syntactically unique.

Exceptions:

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

std::invalid_argument Thrown if `box` contains at least one interval with: a topologically open bound, a single bound, or two bounds which have space between them.

The template class Box must provide the following methods.

```
dimension_type space_dimension() const
```

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

```
bool is_empty() const
```

returns `true` if and only if the bounding box describes the empty set.

```
bool get_lower_bound(dimension_type k, bool closed,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the `k`-th space 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 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,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the `k`-th space 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 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.

9.19.2.9 template<typename Box> Parma_Polyhedra_Library::Grid::Grid (const Box & *box*, From_Covering_Box *dummy*)

Builds a grid out of a generic, interval-based covering box.

The covering box is a set of upper and lower values for each dimension. When a covering box is tiled onto empty space the corners of the tiles form a rectilinear grid.

A box interval with only one bound fixes the values of all grid points in the dimension associated with the box to the value of the bound. A box interval which has upper and lower bounds of equal value allows all grid points with any value in the dimension associated with the interval. The presence of a universe interval results in the empty grid. The empty box produces the empty grid of the same dimension as the box.

Parameters:

box The covering box representing the grid to be built;

dummy A dummy tag to make this constructor syntactically unique.

Exceptions:

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

std::invalid_argument Thrown if `box` contains any topologically open bounds.

The template class Box must provide the following methods.

```
dimension_type space_dimension() const
```

returns the dimension of the vector space enclosing the grid represented by the covering box.

```
bool is_empty() const
```

returns `true` if and only if the covering box describes the empty set.

```
bool get_lower_bound(dimension_type k, bool closed,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the `k`-th space 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 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,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the k-th space 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 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.

9.19.3 Member Function Documentation

9.19.3.1 bool Parma_Polyhedra_Library::Grid::is_disjoint_from (const Grid & *y*) const

Returns true if and only if *this and y are disjoint.

Exceptions:

std::invalid_argument Thrown if x and y are dimension-incompatible.

9.19.3.2 bool Parma_Polyhedra_Library::Grid::is_discrete () const

Returns true if and only if *this is discrete.

A grid is discrete if it can be defined by a generator system which contains only points and parameters. This includes the empty grid and any grid in dimension zero.

9.19.3.3 bool Parma_Polyhedra_Library::Grid::bounds_from_above (const Linear_Expression & *expr*) const [inline]

Returns true if and only if expr is bounded in *this.

This method is the same as bounds_from_below.

Exceptions:

std::invalid_argument Thrown if expr and *this are dimension-incompatible.

9.19.3.4 bool Parma_Polyhedra_Library::Grid::bounds_from_below (const Linear_Expression & *expr*) const [inline]

Returns true if and only if expr is bounded in *this.

This method is the same as bounds_from_above.

Exceptions:

std::invalid_argument Thrown if expr and *this are dimension-incompatible.

9.19.3.5 bool Parma_Polyhedra_Library::Grid::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*) const [inline]

Returns true if and only if *this is not empty and expr is bounded from above in *this, in which case the supremum value is computed.

Parameters:

expr The linear expression to be maximized subject to *this;

sup_n The numerator of the supremum value;

sup_d The denominator of the supremum value;

maximum `true` if the supremum value can be reached in `this`. Always `true` when `this` bounds `expr`. Present for interface compatibility with class Polyhedron, where closure points can result in a value of false.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded by `*this`, `false` is returned and `sup_n`, `sup_d` and `maximum` are left untouched.

9.19.3.6 bool Parma_Polyhedra_Library::Grid::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Grid_Generator & *point*) const `[inline]`

Returns `true` if and only if `*this` is not empty and `expr` is bounded from above in `*this`, in which case the supremum value and a point where `expr` reaches it are computed.

Parameters:

expr The linear expression to be maximized subject to `*this`;

sup_n The numerator of the supremum value;

sup_d The denominator of the supremum value;

maximum `true` if the supremum value can be reached in `this`. Always `true` when `this` bounds `expr`. Present for interface compatibility with class Polyhedron, where closure points can result in a value of false;

point When maximization succeeds, will be assigned a point where `expr` reaches its supremum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded by `*this`, `false` is returned and `sup_n`, `sup_d`, `maximum` and `point` are left untouched.

9.19.3.7 bool Parma_Polyhedra_Library::Grid::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*) const `[inline]`

Returns `true` if and only if `*this` is not empty and `expr` is bounded from below in `*this`, in which case the infimum value is computed.

Parameters:

expr The linear expression to be minimized subject to `*this`;

inf_n The numerator of the infimum value;

inf_d The denominator of the infimum value;

minimum `true` if the is the infimum value can be reached in `this`. Always `true` when `this` bounds `expr`. Present for interface compatibility with class Polyhedron, where closure points can result in a value of false.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from below, `false` is returned and `inf_n`, `inf_d` and `minimum` are left untouched.

9.19.3.8 bool Parma_Polyhedra_Library::Grid::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Grid_Generator & *point*) const `[inline]`

Returns `true` if and only if `*this` is not empty and `expr` is bounded from below in `*this`, in which case the infimum value and a point where `expr` reaches it are computed.

Parameters:

expr The linear expression to be minimized subject to `*this`;

inf_n The numerator of the infimum value;

inf_d The denominator of the infimum value;

minimum `true` if the is the infimum value can be reached in `this`. Always `true` when `this` bounds `expr`. Present for interface compatibility with class Polyhedron, where closure points can result in a value of false;

point When minimization succeeds, will be assigned a point where `expr` reaches its infimum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from below, `false` is returned and `inf_n`, `inf_d`, `minimum` and `point` are left untouched.

9.19.3.9 bool Parma_Polyhedra_Library::Grid::contains (const Grid & *y*) const

Returns `true` if and only if `*this` contains `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.10 bool Parma_Polyhedra_Library::Grid::strictly_contains (const Grid & *y*) const `[inline]`

Returns `true` if and only if `*this` strictly contains `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.11 template<typename Box> void Parma_Polyhedra_Library::Grid::shrink_bounding_-box (Box & *box*) const

Uses `*this` to shrink a generic, interval-based bounding box.

Parameters:

box The bounding box to be shrunk.

Exceptions:

std::invalid_argument Thrown if `*this` and `box` are dimension-incompatible, or if `box` contains any topologically open bounds.

The template class Box must provide the following methods

```
dimension_type space_dimension() const
```

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

```
bool get_lower_bound(dimension_type k, bool closed,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the `k`-th space 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 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,
                     Coefficient& n, Coefficient& d) const
```

Let I be the interval corresponding to the `k`-th space 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 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.

```
set_empty()
```

Causes the box to become empty, i.e., to represent the empty set.

```
raise_lower_bound(dimension_type k, bool closed,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space dimension with $[n/d, +\infty)$. `closed` is always passed as `true`.

```
lower_upper_bound(dimension_type k, bool closed,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space dimension with $(-\infty, n/d]$. `closed` is always passed as `true`.

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)`.

9.19.3.12 template<typename Box> void Parma_Polyhedra_Library::Grid::get_covering_box (Box & *box*) const

Writes the covering box for `*this` into `box`.

The covering box is a set of upper and lower values for each dimension. When the covering box written into `box` is tiled onto empty space the corners of the tiles form the sparsest rectilinear grid that includes `*this`.

The value of the lower bound of each interval of the resulting `box` are as close as possible to the origin, with positive values taking preference when the lowest positive value equals the lowest negative value.

If all the points have a single value in a particular dimension of the grid then there is only a lower bound on the interval produced in `box`, and the lower bound denotes the single value for the dimension. If the coordinates of the points in a particular dimension include every value then the upper and lower bounds of the associated interval in `box` are set equal. The empty grid produces the empty `box`. The zero dimension universe grid produces the zero dimension universe box.

Parameters:

box The Box into which the covering box is written.

Exceptions:

std::invalid_argument Thrown if `*this` and `box` are dimension-incompatible.

The template class Box must provide the following methods

```
Box(dimension_type space_dimension)
```

Creates a universe box of space_dimension dimensions.

```
dimension_type space_dimension() const
```

returns the dimension of the vector space enclosing the grid represented by the covering box.

```
set_empty()
```

Causes the box to become empty, i.e., to represent the empty set.

```
raise_lower_bound(dimension_type k, bool closed,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space dimension with $[n/d, +\infty)$. `closed` is always passed as `true`.

```
lower_upper_bound(dimension_type k, bool closed,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space dimension with $(-\infty, n/d]$. `closed` is always passed as `true`.

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)`.

9.19.3.13 bool Parma_Polyhedra_Library::Grid::OK (bool *check_not_empty* = `false`) const

Checks if all the invariants are satisfied.

Returns:

`true` if and only if `*this` satisfies all the invariants and either `check_not_empty` is `false` or `*this` is not empty.

Parameters:

check_not_empty `true` if and only if, in addition to checking the invariants, `*this` must be checked to be 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.

9.19.3.14 void Parma_Polyhedra_Library::Grid::add_congruence (const Congruence & *cg*)

Adds a copy of congruence `cg` to `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

9.19.3.15 void Parma_Polyhedra_Library::Grid::add_congruence (const Constraint & *c*)

Adds constraint `c` to `*this`.

The addition can only affect `*this` if `c` is an equality.

Exceptions:

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

9.19.3.16 bool Parma_Polyhedra_Library::Grid::add_congruence_and_minimize (const Congruence & *c*)

Adds a copy of congruence `cg` to the system of congruences of `this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

9.19.3.17 bool Parma_Polyhedra_Library::Grid::add_congruence_and_minimize (const Constraint & *c*)

Adds a copy of constraint `c` to `*this`, reducing the result.

The addition can only affect `*this` if `c` is an equality.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

9.19.3.18 void Parma_Polyhedra_Library::Grid::add_generator (const Grid_Generator & *g*)

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

Exceptions:

std::invalid_argument Thrown if `*this` and generator `g` are dimension-incompatible, or if `*this` is an empty grid and `g` is not a point.

9.19.3.19 bool Parma_Polyhedra_Library::Grid::add_generator_and_minimize (const Grid_-Generator & *g*)

Adds a copy of generator `g` to the system of generators of `this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and generator `g` are dimension-incompatible, or if `*this` is an empty grid and `g` is not a point.

9.19.3.20 void Parma_Polyhedra_Library::Grid::add_congruences (const Congruence_System & *cgs*)

Adds a copy of each congruence in `cgs` to `*this`.

Parameters:

cgs Contains the congruences that will be added to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

9.19.3.21 void Parma_Polyhedra_Library::Grid::add_congruences (const Constraint_System & *cs*)

Adds a copy of each equality constraint in `cs` to `*this`.

Parameters:

cs The congruences that will be considered for addition to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

9.19.3.22 void Parma_Polyhedra_Library::Grid::add_recycled_congruences (Congruence_System & *cgs*)

Adds the congruences in `cgs` to *this.

Parameters:

cgs The congruence system that will be recycled, adding its congruences to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cgs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.23 void Parma_Polyhedra_Library::Grid::add_recycled_congruences (Constraint_System & *cs*)

Adds the equality constraints in `cs` to `*this`.

Parameters:

cs The constraint system from which constraints will be considered for addition to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.24 bool Parma_Polyhedra_Library::Grid::add_congruences_and_minimize (const Congruence_System & *cgs*)

Adds a copy of the congruences in `cgs` to the system of congruences of `*this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cgs Contains the congruences that will be added to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

9.19.3.25 bool Parma_Polyhedra_Library::Grid::add_congruences_and_minimize (const Constraint_System & *cs*)

Adds a copy of each equality constraint in `cs` to `*this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cs Contains the constraints that will be added to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

9.19.3.26 bool Parma_Polyhedra_Library::Grid::add_recycled_congruences_and_minimize (Congruence_System & *cgs*)

Adds the congruences in `cgs` to the system of congruences of `this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cgs The congruence system that will be recycled, adding its congruences to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cgs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.27 bool Parma_Polyhedra_Library::Grid::add_recycled_congruences_and_minimize (Constraint_System & *cs*)

Adds the equalities in `cs` to `*this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cs The constraint system that will be recycled, adding its equalities to the system of congruences of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.28 void Parma_Polyhedra_Library::Grid::add_constraint (const Constraint & *c*)

Adds constraint `c` to `*this`.

The addition can only affect `*this` if `c` is an equality.

Exceptions:

std::invalid_argument Thrown if `*this` and `c` are dimension-incompatible.

9.19.3.29 bool Parma_Polyhedra_Library::Grid::add_constraint_and_minimize (const Constraint & *c*)

Adds constraint `c` to `*this`, reducing the result.

The addition can only affect `*this` if `c` is an equality.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `c` are dimension-incompatible.

9.19.3.30 void Parma_Polyhedra_Library::Grid::add_constraints (const Constraint_System & *cs*)

Adds copies of the equality constraints in `cs` to `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

9.19.3.31 bool Parma_Polyhedra_Library::Grid::add_constraints_and_minimize (const Constraint_System & *cs*)

Adds copies of the equality constraints in `cs` to `*this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

9.19.3.32 void Parma_Polyhedra_Library::Grid::add_recycled_constraints (Constraint_System & *cs*)

Adds the equality constraints in `cs` to `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.33 bool Parma_Polyhedra_Library::Grid::add_recycled_constraints_and_minimize (Constraint_System & *cs*)

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Warning:

The only assumption that can be made about `cs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.34 void Parma_Polyhedra_Library::Grid::add_generators (const Grid_Generator_System & *gs*)

Adds a copy of the generators in `gs` to the system of generators of `*this`.

Parameters:

gs Contains the generators that will be added to the system of generators of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `gs` are dimension-incompatible, or if `*this` is empty and the system of generators `gs` is not empty, but has no points.

9.19.3.35 void Parma_Polyhedra_Library::Grid::add_recycled_generators (Grid_Generator_-System & *gs*)

Adds the generators in `gs` to the system of generators of `this`.

Parameters:

gs The generator system that will be recycled, adding its generators to the system of generators of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `gs` are dimension-incompatible, or if `*this` is empty and the system of generators `gs` is not empty, but has no points.

Warning:

The only assumption that can be made about `gs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.36 bool Parma_Polyhedra_Library::Grid::add_generators_and_minimize (const Grid_Generator_System & *gs*)

Adds a copy of the generators in `gs` to the system of generators of `*this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

gs Contains the generators that will be added to the system of generators of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `gs` are dimension-incompatible, or if `this` is empty and the system of generators `gs` is not empty, but has no points.

9.19.3.37 bool Parma_Polyhedra_Library::Grid::add_recycled_generators_and_minimize (Grid_Generator_System & *gs*)

Adds the generators in `gs` to the system of generators of `this`, reducing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

gs The generator system that will be recycled, adding its generators to the system of generators of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `gs` are dimension-incompatible, or if `this` is empty and the system of generators `gs` is not empty, but has no points.

Warning:

The only assumption that can be made about `gs` upon successful or exceptional return is that it can be safely destroyed.

9.19.3.38 void Parma_Polyhedra_Library::Grid::intersection_assign (const Grid & *y*)

Assigns to `*this` the intersection of `*this` and `y`. The result is not guaranteed to be reduced.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.39 bool Parma_Polyhedra_Library::Grid::intersection_assign_and_minimize (const Grid & *y*)

Assigns to `*this` the intersection of `*this` and `y`, reducing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.40 void Parma_Polyhedra_Library::Grid::join_assign (const Grid & *y*)

Assigns to `*this` the join of `*this` and `y`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.41 bool Parma_Polyhedra_Library::Grid::join_assign_and_minimize (const Grid & *y*)

Assigns to `*this` the join of `*this` and `y`, reducing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.42 bool Parma_Polyhedra_Library::Grid::join_assign_if_exact (const Grid & *y*)

If the join of `*this` and `y` is exact it is assigned to `this` and `true` is returned, otherwise `false` is returned.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.43 void Parma_Polyhedra_Library::Grid::grid_difference_assign (const Grid & *y*)

Assigns to `*this` the grid-difference of `*this` and `y`.

The grid difference between grids x and y is the smallest grid containing all the points from x and y that are only in x.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.19.3.44 void Parma_Polyhedra_Library::Grid::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the affine image of `this` under the function mapping variable `var` to 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 space dimension of `*this`.

9.19.3.45 void Parma_Polyhedra_Library::Grid::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the affine preimage of `*this` under the function mapping variable `var` to 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 space dimension of `*this`.

9.19.3.46 void Parma_Polyhedra_Library::Grid::generalized_affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`, Coefficient_traits::const_reference *modulus* = `Coefficient_one()`)

Assigns to `*this` the image of `*this` with respect to the generalized affine relation $\mathrm{var}' = \frac{\mathrm{expr}}{\mathrm{denominator}} \pmod{\mathrm{modulus}}$.

Parameters:

var The left hand side variable of the generalized affine relation;

expr The numerator of the right hand side affine expression;

denominator The denominator of the right hand side affine expression. Optional argument with an automatic value of one;

modulus The modulus of the congruence lhs = rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of one.

Exceptions:

std::invalid_argument Thrown if `denominator` is zero or if `expr` and `*this` are dimension-incompatible or if `var` is not a space dimension of `this`.

9.19.3.47 void Parma_Polyhedra_Library::Grid::generalized_affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`, Coefficient_traits::const_reference *modulus* = `Coefficient_one()`)

Assigns to `*this` the preimage of `*this` with respect to the generalized affine relation $\mathrm{var}' = \frac{\mathrm{expr}}{\mathrm{denominator}} \pmod{\mathrm{modulus}}$.

Parameters:

var The left hand side variable of the generalized affine relation;

expr The numerator of the right hand side affine expression;

denominator The denominator of the right hand side affine expression. Optional argument with an automatic value of one;

modulus The modulus of the congruence lhs = rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of one.

Exceptions:

std::invalid_argument Thrown if `denominator` is zero or if `expr` and `*this` are dimension-incompatible or if `var` is not a space dimension of `this`.

9.19.3.48 void Parma_Polyhedra_Library::Grid::generalized_affine_image (const Linear_Expression & *lhs*, const Linear_Expression & *rhs*, Coefficient_traits::const_reference *modulus* = `Coefficient_one()`)

Assigns to `*this` the image of `*this` with respect to the generalized affine relation $\mathrm{lhs}' = \mathrm{rhs} \pmod{\mathrm{modulus}}$.

Parameters:

lhs The left hand side affine expression.

rhs The right hand side affine expression.

modulus The modulus of the congruence lhs = rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of one.

Exceptions:

std::invalid_argument Thrown if `*this` is dimension-incompatible with `lhs` or `rhs`.

9.19.3.49 void Parma_Polyhedra_Library::Grid::generalized_affine_preimage (const Linear_-Expression & *lhs*, const Linear_Expression & *rhs*, Coefficient_traits::const_reference *modulus* = `Coefficient_one()`)

Assigns to `*this` the preimage of `*this` with respect to the generalized affine relation $\mathrm{lhs}' = \mathrm{rhs} \pmod{\mathrm{modulus}}$.

Parameters:

lhs The left hand side affine expression;

rhs The right hand side affine expression;

modulus The modulus of the congruence lhs = rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of one.

Exceptions:

std::invalid_argument Thrown if `*this` is dimension-incompatible with `lhs` or `rhs`.

9.19.3.50 void Parma_Polyhedra_Library::Grid::time_elapse_assign (const Grid & *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 dimension-incompatible.

9.19.3.51 void Parma_Polyhedra_Library::Grid::widening_assign (const Grid & *y*, unsigned ∗ *tp* = `NULL`)

Assigns to `*this` the result of computing the Grid widening between `*this` and `y`.

Parameters:

y A grid 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 dimension-incompatible.

9.19.3.52 void Parma_Polyhedra_Library::Grid::limited_extrapolation_assign (const Grid & *y*, const Congruence_System & *cgs*, unsigned ∗ *tp* = `NULL`)

Improves the result of the Grid widening computation by also enforcing those congruences in `cgs` that are satisfied by all the points of `*this`.

Parameters:

y A grid that *must* be contained in `*this`;

cgs The system of congruences used to improve the widened grid;

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 dimension-incompatible.

9.19.3.53 void Parma_Polyhedra_Library::Grid::add_space_dimensions_and_embed (dimension_type *m*)

Adds `m` new space dimensions and embeds the old grid in the new vector space.

Parameters:

m The number of dimensions to add.

Exceptions:

std::length_error Thrown if adding `m` new space dimensions would cause the vector space to exceed dimension `max_space_dimension()`.

The new space dimensions will be those having the highest indexes in the new grid, which is characterized by a system of congruences in which the variables which are the new dimensions can have any value. For instance, when starting from the grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

$$\{ (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{L} \}.$$

9.19.3.54 void Parma_Polyhedra_Library::Grid::add_space_dimensions_and_project (dimension_type *m*)

Adds `m` new space dimensions to the grid and does not embed it in the new vector space.

Parameters:

m The number of space dimensions to add.

Exceptions:

std::length_error Thrown if adding `m` new space dimensions would cause the vector space to exceed dimension `max_space_dimension()`.

The new space dimensions will be those having the highest indexes in the new grid, which is characterized by a system of congruences in which the variables running through the new dimensions are all constrained to be equal to 0. For instance, when starting from the grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

$$\{ (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{L} \}.$$

9.19.3.55 void Parma_Polyhedra_Library::Grid::concatenate_assign (const Grid & *y*)

Assigns to `*this` the concatenation of `*this` and `y`, taken in this order.

Exceptions:

std::length_error Thrown if the concatenation would cause the vector space to exceed dimension `max_space_dimension()`.

9.19.3.56 void Parma_Polyhedra_Library::Grid::remove_space_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions from the vector space.

Parameters:

to_be_removed The set of Variable objects corresponding to the space 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`.

9.19.3.57 void Parma_Polyhedra_Library::Grid::remove_higher_space_dimensions (dimension_-type *new_dimension*)

Removes the higher dimensions of the vector space 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`.

9.19.3.58 template<typename Partial_Function> void Parma_Polyhedra_Library::Grid::map_-space_dimensions (const Partial_Function & *pfunc*)

Remaps the dimensions of the vector space according to a partial function.

If `pfunc` maps only some of the dimensions of `*this` then the rest will be projected away.

If the highest dimension mapped to by `pfunc` is higher than the highest dimension in `*this` then the number of dimensions in `this` will be increased to the highest dimension mapped to by `pfunc`.

Parameters:

pfunc The partial function specifying the destiny of each space dimension.

The template class Partial_Function 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. The `max_in_codomain()` method is called at most once.

```
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. This method is called at most n times, where n is the dimension of the vector space enclosing the grid.

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

9.19.3.59 void Parma_Polyhedra_Library::Grid::expand_space_dimension (Variable *var*, dimension_type *m*)

Creates `m` copies of the space dimension corresponding to `var`.

Parameters:

var The variable corresponding to the space dimension to be replicated;

m The number of replicas to be created.

Exceptions:

std::invalid_argument Thrown if `var` does not correspond to a dimension of the vector space.

std::length_error Thrown if adding `m` new space dimensions would cause the vector space to exceed dimension `max_space_dimension()`.

If `*this` has space dimension n, with $n > 0$, and `var` has space dimension $k \leq n$, then the k-th space dimension is expanded to `m` new space dimensions $n, n+1, \dots, n+m-1$.

9.19.3.60 void Parma_Polyhedra_Library::Grid::fold_space_dimensions (const Variables_Set & *to_be_folded*, Variable *var*)

Folds the space dimensions in `to_be_folded` into `var`.

Parameters:

to_be_folded The set of Variable objects corresponding to the space dimensions to be folded;

var The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions:

std::invalid_argument Thrown if `*this` is dimension-incompatible with `var` or with one of the Variable objects contained in `to_be_folded`. Also thrown if `var` is contained in `to_be_folded`.

If `*this` has space dimension n, with $n > 0$, `var` has space dimension $k \leq n$, `to_be_folded` is a set of variables whose maximum space dimension is also less than or equal to n, and `var` is not a member of `to_be_folded`, then the space dimensions corresponding to variables in `to_be_folded` are folded into the k-th space dimension.

9.19.4 Friends And Related Function Documentation

9.19.4.1 bool operator== (const Grid & *x*, const Grid & *y*) `[friend]`

Returns `true` if and only if `x` and `y` are the same grid.

Note that `x` and `y` may be dimension-incompatible grids: in those cases, the value `false` is returned.

9.19.4.2 std::ostream & operator<< (std::ostream & *s*, const Grid & *gr*) `[related]`

Output operator.

Writes a textual representation of `gr` on `s`: `false` is written if `gr` is an empty grid; `true` is written if `gr` is a universe grid; a minimized system of congruences defining `gr` is written otherwise, all congruences in one row separated by ", "s.

9.19.4.3 bool operator!= (const Grid & *x*, const Grid & *y*) `[related]`

Returns `true` if and only if `x` and `y` are different grids.

Note that `x` and `y` may be dimension-incompatible grids: in those cases, the value `true` is returned.

9.20 Parma_Polyhedra_Library::Grid_Certificate Class Reference

The convergence certificate for the Grid widening operator.

Public Member Functions

- Grid_Certificate ()
  Default constructor.
- Grid_Certificate (const Grid &gr)
  Constructor: computes the certificate for `gr`.
- Grid_Certificate (const Grid_Certificate &y)
  Copy constructor.
- ∼Grid_Certificate ()
  Destructor.
- int compare (const Grid_Certificate &y) const
  The comparison function for certificates.
- int compare (const Grid &gr) const
  Compares `*this` *with the certificate for grid* `gr`.

Classes

- struct Compare
  A total ordering on Grid certificates.

9.20.1 Detailed Description

The convergence certificate for the Grid widening operator.

Convergence certificates are used to instantiate the BHZ03 framework so as to define widening operators for the finite powerset domain.

Note:

Each convergence certificate has to be used together with a compatible widening operator. In particular, Grid_Certificate can certify the Grid widening.

9.20.2 Member Function Documentation

9.20.2.1 int Parma_Polyhedra_Library::Grid_Certificate::compare (const Grid_Certificate & *y*) const

The comparison function for certificates.

Returns:

-1, 0 or 1 depending on whether `*this` is smaller than, equal to, or greater than `y`, respectively.

9.21 Parma_Polyhedra_Library::Grid_Certificate::Compare Struct Reference

A total ordering on Grid certificates.

Public Member Functions

- bool operator() (const Grid_Certificate &x, const Grid_Certificate &y) const

  Returns `true` *if and only if* `x` *comes before* `y`.

9.21.1 Detailed Description

A total ordering on Grid certificates.

This binary predicate defines a total ordering on Grid certificates which is used when storing information about sets of grids.

9.22 Parma_Polyhedra_Library::Grid_Generator Class Reference

A line, parameter or point.

Inherits Parma_Polyhedra_Library::Generator.

Public Types

- enum Type { LINE, PARAMETER, POINT }

  The generator type.

Public Member Functions

- 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_parameter () const

  Returns `true` *if and only if* `*this` *is a parameter.*

- bool is_line_or_parameter () const

  Returns `true` *if and only if* `*this` *is a line or a parameter.*

- bool is_point () const

  Returns `true` *if and only if* `*this` *is a point.*

- bool is_parameter_or_point () const

  Returns `true` *if and only if* `*this` *row represents a parameter or a point.*

- Grid_Generator & operator= (const Grid_Generator &g)

  Assignment operator.

- Grid_Generator & operator= (const Generator &g)

  Assignment operator.

- Coefficient_traits::const_reference divisor () const

  Returns the divisor of `*this`.

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- bool is_equivalent_to (const Grid_Generator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are equivalent generators.*

- bool is_equal_to (const Grid_Generator &y) const

  Returns `true` *if* `*this` *is exactly equal to* `y`.

- bool all_homogeneous_terms_are_zero () const

  Returns `true` *if and only if all the homogeneous terms of* `*this` *are* 0.

- void scale_to_divisor (Coefficient_traits::const_reference d)

  Scales `*this` *to be represented with a divisor of* `d` *(if *this is a parameter or point).*

- bool OK () const

  Checks if all the invariants are satisfied.

- void swap (Grid_Generator &y)

  Swaps `*this` *with* `y`.

- void coefficient_swap (Grid_Generator &y)

  Swaps `*this` *with* `y`*, leaving* `*this` *with the original capacity.*

Static Public Member Functions

- static Grid_Generator line (const Linear_Expression &e)

  Returns the line of direction `e`.

- static Grid_Generator parameter (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one())

  Shorthand for Grid_Generator Grid_Generator::parameter(const Linear_Expression& e, Coefficient_-traits::const_reference d).

- static Grid_Generator point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_-traits::const_reference d=Coefficient_one())

  Returns the point at `e` */* `d`.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Grid_Generator &g)

  Output operator.

- void swap (Parma_Polyhedra_Library::Grid_Generator &x, Parma_Polyhedra_Library::Grid_-Generator &y)

  Specializes `std::swap`.

- Grid_Generator grid_line (const Linear_Expression &e)

  Shorthand for Grid_Generator Grid_Generator::line(const Linear_Expression& e).

- Grid_Generator grid_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_-traits::const_reference d=Coefficient_one())

  Shorthand for Grid_Generator Grid_Generator::point(const Linear_Expression& e, Coefficient_-traits::const_reference d).

- bool operator== (const Grid_Generator &x, const Grid_Generator &y)

  Returns `true` *if and only if* `x` *is equivalent to* `y`.

- bool operator!= (const Grid_Generator &x, const Grid_Generator &y)

  Returns `true` *if and only if* `x` *is not equivalent to* `y`.

- std::ostream & operator<< (std::ostream &s, const Grid_Generator::Type &t)

  Output operator.

9.22.1 Detailed Description

A line, parameter or point.

An object of the class Grid_Generator is one of the following:

- a line $\boldsymbol{l} = (a_0, \ldots, a_{n-1})^{\mathrm{T}}$;
- a parameter $\boldsymbol{q} = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$;
- a point $\boldsymbol{p} = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$;

where n is the dimension of the space and, for points and parameters, $d > 0$ is the divisor.

How to build a grid generator.

Each type of generator is built by applying the corresponding function (`line`, `parameter` or `point`) to a linear expression; 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 parameters, an optional Coefficient 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:

```
Grid_Generator l = grid_line(x - y - z);
```

By definition, the origin of the space is not a line, so that the following code throws an exception:

```
Grid_Generator l = grid_line(0*x);
```

Example 2

The following code builds the parameter as the vector $\boldsymbol{p} = (1, -1, -1)^{\mathrm{T}} \in \mathbb{R}^3$ which has the same direction as the line in Example 1:

```
Grid_Generator q = parameter(x - y - z);
```

Note that, unlike lines, for parameters, the length as well as the direction of the vector represented by the code is significant. Thus `q` is *not* the same as the parameter `q1` defined by

```
Grid_Generator q1 = parameter(2x - 2y - 2z);
```

By definition, the origin of the space is not a parameter, so that the following code throws an exception:

```
Grid_Generator q = parameter(0*x);
```

Example 3

The following code builds the point $\boldsymbol{p} = (1, 0, 2)^{\mathrm{T}} \in \mathbb{R}^3$:

```
Grid_Generator p = grid_point(1*x + 0*y + 2*z);
```

The same effect can be obtained by using the following code:

```
Grid_Generator p = grid_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:

```
Grid_Generator origin3 = grid_point(0*x + 0*y + 0*z);
Grid_Generator origin3_alt = grid_point(0*z);
```

Note however that the following code would have defined a different point, namely $\mathbf{0} \in \mathbb{R}^2$:

```
Grid_Generator origin2 = grid_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.

```
Grid_Generator origin0 = Generator::zero_dim_point();
Grid_Generator origin0_alt = grid_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 `grid_point` (the divisor):

```
Grid_Generator p = grid_point(2*x + 0*y + 4*z, 2);
```

Obviously, the divisor can be used to specify points having some non-integer (but rational) coordinates. For instance, the point $\boldsymbol{p1} = (-1.5, 3.2, 2.1)^{\mathrm{T}} \in \mathbb{R}^3$ can be specified by the following code:

```
Grid_Generator p1 = grid_point(-15*x + 32*y + 21*z, 10);
```

If a zero divisor is provided, an exception is thrown.

Example 5

Parameters, like points can have a divisor. For instance, the parameter $\boldsymbol{q} = (1, 0, 2)^{\mathrm{T}} \in \mathbb{R}^3$ can be defined:

```
Grid_Generator q = parameter(2*x + 0*y + 4*z, 2);
```

Also, the divisor can be used to specify parameters having some non-integer (but rational) coordinates. For instance, the parameter $\boldsymbol{q} = (-1.5, 3.2, 2.1)^{\mathrm{T}} \in \mathbb{R}^3$ can be defined:

```
Grid_Generator q = parameter(-15*x + 32*y + 21*z, 10);
```

If a zero divisor is provided, an exception is thrown.

How to inspect a grid generator

Several methods are provided to examine a grid generator and extract all the encoded information: its space dimension, its type and the value of its integer coefficients and the value of the denominator.

Example 6

The following code shows how it is possible to access each single coefficient of a grid generator. If `g1` is a point having coordinates $(a_0, \ldots, a_{n-1})^{\mathrm{T}}$, we construct the parameter `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;
  Linear_Expression e;
  for (int i = g1.space_dimension() - 1; i >= 0; i--)
    e += (i + 1) * g1.coefficient(Variable(i)) * Variable(i);
  Grid_Generator g2 = parameter(e, g1.divisor());
  cout << "Parameter g2: " << g2 << endl;
}
else
  cout << "Grid Generator g1 is not a point." << endl;
```

Therefore, for the point

```
Grid_Generator g1 = grid_point(2*x - y + 3*z, 2);
```

we would obtain the following output:

```
Point g1: p((2*A - B + 3*C)/2)
Parameter g2: parameter((2*A - 2*B + 9*C)/2)
```

When working with points and parameters, be careful not to confuse the notion of *coefficient* with the notion of *coordinate*: these are equivalent only when the divisor is 1.

9.22.2 Member Enumeration Documentation

9.22.2.1 enum Parma_Polyhedra_Library::Grid_Generator::Type

The generator type.

Enumerator:

LINE The generator is a line.

PARAMETER The generator is a parameter.

POINT The generator is a point.

Reimplemented from Parma_Polyhedra_Library::Generator.

9.22.3 Member Function Documentation

9.22.3.1 static Grid_Generator Parma_Polyhedra_Library::Grid_Generator::line (const Linear_Expression & *e*) `[static]`

Returns the line of direction `e`.

Exceptions:

std::invalid_argument Thrown if the homogeneous part of `e` represents the origin of the vector space.

Reimplemented from Parma_Polyhedra_Library::Generator.

9.22.3.2 Grid_Generator parameter (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Shorthand for Grid_Generator Grid_Generator::parameter(const Linear_Expression& e, Coefficient_traits::const_reference d).

Both `e` and `d` are optional arguments, with default values Linear_Expression::zero() and Coefficient_one(), respectively.

Exceptions:

std::invalid_argument Thrown if `d` is zero.

9.22.3.3 static Grid_Generator Parma_Polyhedra_Library::Grid_Generator::point (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[static]`

Returns the point at `e` / `d`.

Both `e` and `d` are optional arguments, with default values Linear_Expression::zero() and Coefficient_one(), respectively.

Exceptions:

std::invalid_argument Thrown if `d` is zero.

Reimplemented from Parma_Polyhedra_Library::Generator.

9.22.3.4 Coefficient_traits::const_reference Parma_Polyhedra_Library::Grid_Generator::divisor () const `[inline]`

Returns the divisor of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` is a line.

Reimplemented from Parma_Polyhedra_Library::Generator.

9.22.3.5 bool Parma_Polyhedra_Library::Grid_Generator::is_equivalent_to (const Grid_Generator & *y*) const

Returns `true` if and only if `*this` and `y` are equivalent generators.

Generators having different space dimensions are not equivalent.

9.22.3.6 void Parma_Polyhedra_Library::Grid_Generator::scale_to_divisor (Coefficient_traits::const_reference *d*)

Scales `*this` to be represented with a divisor of `d` (if *this is a parameter or point).

Exceptions:

std::invalid_argument Thrown if `d` is zero.

9.22.3.7 void Parma_Polyhedra_Library::Grid_Generator::coefficient_swap (Grid_Generator & *y*)

Swaps `*this` with `y`, leaving `*this` with the original capacity.

All up to and including the last element of the smaller of `this` and `y` are swapped. The parameter divisor element of `y` is swapped with the divisor element of `*this`.

9.23 Parma_Polyhedra_Library::Grid_Generator_System Class Reference

A system of grid generators.

Inherits Parma_Polyhedra_Library::Generator_System.

Public Member Functions

- Grid_Generator_System ()

  Default constructor: builds an empty system of generators.

- Grid_Generator_System (const Grid_Generator_System &gs)

  Ordinary copy-constructor.

- Grid_Generator_System (dimension_type dim)

  Builds an empty system of generators of dimension `dim`.

- Grid_Generator_System (const Grid_Generator &g)

  Builds the singleton system containing only generator `g`.

- dimension_type space_dimension () const

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

- void clear ()

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

- void insert (const Grid_Generator &g)

  Inserts into `*this` *a copy of the generator* `g`*, increasing the number of space dimensions if needed.*

- void recycling_insert (Grid_Generator &g)

  Inserts into `*this` *the generator* `g`*, increasing the number of space dimensions if needed.*

- void recycling_insert (Grid_Generator_System &gs)

  Inserts into `*this` *the generators in* `gs`*, increasing the number of space dimensions if needed.*

- const_iterator begin () const

  Returns the const_iterator pointing to the first generator, if `this` *is not empty; otherwise, returns the past-the-end const_iterator.*

- const_iterator end () const

  Returns the past-the-end const_iterator.

- void swap (Grid_Generator_System &y)

Swaps `*this` *with* `y`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- void affine_image (dimension_type v, const Linear_Expression &expr, Coefficient_traits::const_reference denominator)

  Assigns to a given variable an affine expression.

- dimension_type num_generators () const

  Returns the number of generators in the system.

- dimension_type num_parameters () const

  Returns the number of parameters in the system.

- dimension_type num_lines () const

  Returns the number of lines in the system.

- bool has_points () const

  Returns `true` *if and only if* `*this` *contains one or more points.*

- bool is_equal_to (const Grid_Generator_System y) const

  Returns `true` *if* `*this` *is identical to* `y`.

- Grid_Generator & operator[ ] (dimension_type k)

  Returns the `k-` *th generator of the system.*

- const Grid_Generator & operator[ ] (dimension_type k) const

  Returns a constant reference to the `k-` *th generator of the system.*

- bool OK () const

  Checks if all the invariants are satisfied.

- void add_universe_rows_and_columns (dimension_type dims)

  Adds `dims` *rows and* `dims` *columns of zeroes to the matrix, initializing the added rows as in the universe system.*

- void remove_space_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified dimensions from the generator system.

- void remove_higher_space_dimensions (dimension_type new_dimension)

  Removes the higher dimensions of the system so that the resulting system will have dimension `new_dimension`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Grid_Generator_System can handle.

Friends

- bool operator== (const Grid_Generator_System &x, const Grid_Generator_System &y)

  Returns `true` *if and only if* `x` *and* `y` *are identical.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Grid_Generator_System &gs)

  Output operator.

- void swap (Parma_Polyhedra_Library::Grid_Generator_System &x, Parma_Polyhedra_-Library::Grid_Generator_System &y)

  Specializes `std::swap`.

Classes

- class const_iterator

  An iterator over a system of grid generators.

9.23.1 Detailed Description

A system of grid generators.

An object of the class Grid_Generator_System is a system of grid generators, i.e., a multiset of objects of the class Grid_Generator (lines, parameters and points). When inserting generators in a system, space dimensions are automatically adjusted so that all the generators in the system are defined on the same vector space. A system of grid generators which is meant to define a non-empty grid must include at least one point: the reason is that lines and parameters need a supporting point (lines only specify directions while parameters only specify direction and distance.

In all the examples it is assumed that variables `x` and `y` are defined as follows:

```
Variable x(0);
Variable y(1);
```

Example 1

The following code defines the line having the same direction as the x axis (i.e., the first Cartesian axis) in $\mathbb{R}^2$:

```
Grid_Generator_System gs;
gs.insert(grid_line(x + 0*y));
```

As said above, this system of generators corresponds to an empty grid, because the line has no supporting point. To define a system of generators that does correspond to the x axis, we can add the following code which inserts the origin of the space as a point:

```
gs.insert(grid_point(0*x + 0*y));
```

Since space dimensions are automatically adjusted, the following code obtains the same effect:

```
gs.insert(grid_point(0*x));
```

In contrast, if we had added the following code, we would have defined a line parallel to the x axis through the point $(0,1)^{\mathrm{T}} \in \mathbb{R}^2$.

```
gs.insert(grid_point(0*x + 1*y));
```

Example 2

The following code builds a system of generators corresponding to the grid consisting of all the integral points on the x axes; that is, all points satisfying the congruence relation

$$\bigl\{\, (x,0)^{\mathrm{T}} \in \mathbb{R}^2 \bigm| x \pmod{1}\, 0 \,\bigr\},$$

```
Grid_Generator_System gs;
gs.insert(parameter(x + 0*y));
gs.insert(grid_point(0*x + 0*y));
```

Example 3

The following code builds a system of generators having three points corresponding to a non-relational grid consisting of all points whose coordinates are integer multiple of 3.

```
Grid_Generator_System gs;
gs.insert(grid_point(0*x + 0*y));
gs.insert(grid_point(0*x + 3*y));
gs.insert(grid_point(3*x + 0*y));
```

Example 4

By using parameters instead of two of the points we can define the same grid as that defined in the previous example. Note that there has to be at least one point and, for this purpose, any point in the grid could be considered. Thus the following code builds two identical grids from the grid generator systems `gs` and `gs1`.

```
Grid_Generator_System gs;
gs.insert(grid_point(0*x + 0*y));
gs.insert(parameter(0*x + 3*y));
gs.insert(parameter(3*x + 0*y));
Grid_Generator_System gs1;
gs1.insert(grid_point(3*x + 3*y));
gs1.insert(parameter(0*x + 3*y));
gs1.insert(parameter(3*x + 0*y));
```

Example 5

The following code builds a system of generators having one point and a parameter corresponding to all the integral points that lie on $x + y = 2$ in $\mathbb{R}^2$

```
Grid_Generator_System gs;
gs.insert(grid_point(1*x + 1*y));
gs.insert(parameter(1*x - 1*y));
```

Note:

After inserting a multiset of generators in a grid generator system, there are no guarantees that an *exact* copy of them can be retrieved: in general, only an *equivalent* grid generator system will be available, where original generators may have been reordered, removed (if they are duplicate or redundant), etc.

9.23.2 Member Function Documentation

9.23.2.1 void Parma_Polyhedra_Library::Grid_Generator_System::insert (const Grid_Generator & *g*)

Inserts into `*this` a copy of the generator `g`, increasing the number of space dimensions if needed.

If `g` is an all-zero parameter then the only action is to ensure that the space dimension of `*this` is at least the space dimension of `g`.

9.23.2.2 void Parma_Polyhedra_Library::Grid_Generator_System::affine_image (dimension_type *v*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator*)

Assigns to a given variable an affine expression.

Parameters:

v Index of the column to which the affine transformation is assigned;

expr The numerator of the affine transformation: $\sum_{i=0}^{n-1} a_i x_i + b$;

denominator The denominator of the affine transformation;

We allow affine transformations (see the Section Operations on Rational Grids)to have rational coefficients. Since the coefficients of linear expressions are integers we also provide an integer `denominator` that will be used as denominator of the affine transformation. The denominator is required to be a positive integer and its default value is 1.

The affine transformation assigns to each element of `v` -th column the follow expression:

$$\frac{\sum_{i=0}^{n-1} a_i x_i + b}{\mathrm{denominator}}.$$

`expr` is a constant parameter and unaltered by this computation.

Reimplemented from Parma_Polyhedra_Library::Generator_System.

9.23.2.3 bool Parma_Polyhedra_Library::Grid_Generator_System::OK () const

Checks if all the invariants are satisfied.

Returns `true` if and only if `*this` is a valid Linear_System and each row in the system is a valid Grid_-Generator.

Reimplemented from Parma_Polyhedra_Library::Generator_System.

9.23.2.4 void Parma_Polyhedra_Library::Grid_Generator_System::add_universe_rows_and_-columns (dimension_type *dims*)

Adds `dims` rows and `dims` columns of zeroes to the matrix, initializing the added rows as in the universe system.

Parameters:

dims The number of rows and columns to be added: must be strictly positive.

Turns the $r \times c$ matrix A into the $(r + dims) \times (c + dims)$ matrix $\left(\begin{smallmatrix} A & 0 \\ 0 & B \end{smallmatrix}\right)$ where B is the $dims \times dims$ unit matrix of the form $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$. The matrix is expanded avoiding reallocation whenever possible.

9.23.2.5 void Parma_Polyhedra_Library::Grid_Generator_System::remove_space_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions from the generator system.

Exceptions:

std::invalid_argument Thrown if the highest space dimension of the variables in `to_be_removed` is higher than the space dimension of `*this`.

9.23.2.6 void Parma_Polyhedra_Library::Grid_Generator_System::remove_higher_space_dimensions (dimension_type *new_dimension*)

Removes the higher dimensions of the system so that the resulting system will have dimension `new_dimension`.

Exceptions:

std::invalid_argument Thrown if the `new_dimension` is higher than the space dimension of `*this`.

9.23.3 Friends And Related Function Documentation

9.23.3.1 std::ostream & operator<< (std::ostream & *s*, const Grid_Generator_System & *gs*) `[related]`

Output operator.

Writes `false` if `gs` is empty. Otherwise, writes on `s` the generators of `gs`, all in one row and separated by ", ".

9.24 Parma_Polyhedra_Library::Grid_Generator_System::const_iterator Class Reference

An iterator over a system of grid generators.

Inherits Parma_Polyhedra_Library::Generator_System::const_iterator.

Public Member Functions

- const_iterator ()

  Default constructor.

- const_iterator (const const_iterator &y)

  Ordinary copy-constructor.

- ∼const_iterator ()

  Destructor.

- const_iterator & operator= (const const_iterator &y)

  Assignment operator.

- const Grid_Generator & operator ∗ () const

  Dereference operator.

- const Grid_Generator ∗ operator → () const

  Indirect member selector.

- const_iterator & operator++ ()

  Prefix increment operator.

- const_iterator operator++ (int)

  Postfix increment operator.

- bool operator== (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are identical.*

- bool operator!= (const const_iterator &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are different.*

9.24.1 Detailed Description

An iterator over a system of grid generators.

A const_iterator is used to provide read-only access to each generator contained in an object of Grid_-Generator_System.

Example

The following code prints the system of generators of the grid `gr`:

```
const Grid_Generator_System& gs = gr.generators();
for (Grid_Generator_System::const_iterator i = gs.begin(),
      gs_end = gs.end(); i != gs_end; ++i)
  cout << *i << endl;
```

The same effect can be obtained more concisely by using more features of the STL:

```
const Generator_System& gs = gr.generators();
copy(gs.begin(), gs.end(), ostream_iterator<Grid_Generator>(cout, "\n"));
```

9.25 Parma_Polyhedra_Library::H79_Certificate Class Reference

A convergence certificate for the H79 widening operator.

Public Member Functions

- H79_Certificate ()

    Default constructor.

- template<typename PH> H79_Certificate (const PH &ph)

    Constructor: computes the certificate for `ph`.

- H79_Certificate (const Polyhedron &ph)

    Constructor: computes the certificate for `ph`.

- H79_Certificate (const H79_Certificate &y)

    Copy constructor.

- ~H79_Certificate ()

    Destructor.

- int compare (const H79_Certificate &y) const

    The comparison function for certificates.

- template<typename PH> int compare (const PH &ph) const

    Compares `*this` *with the certificate for polyhedron* `ph`.

- int compare (const Polyhedron &ph) const

    Compares `*this` *with the certificate for polyhedron* `ph`.

Classes

- struct Compare

    A total ordering on H79 certificates.

9.25.1 Detailed Description

A convergence certificate for the H79 widening operator.

Convergence certificates are used to instantiate the BHZ03 framework so as to define widening operators for the finite powerset domain.

Note:

The convergence of the H79 widening can also be certified by BHRZ03_Certificate.

9.25.2 Member Function Documentation

9.25.2.1 int Parma_Polyhedra_Library::H79_Certificate::compare (const H79_Certificate & *y*) const

The comparison function for certificates.

Returns:

-1, 0 or 1 depending on whether `*this` is smaller than, equal to, or greater than `y`, respectively.

Compares `*this` with `y`, using a total ordering which is a refinement of the limited growth ordering relation for the H79 widening.

9.26 Parma_Polyhedra_Library::H79_Certificate::Compare Struct Reference

A total ordering on H79 certificates.

Public Member Functions

- bool operator() (const H79_Certificate &x, const H79_Certificate &y) const

  Returns `true` *if and only if* `x` *comes before* `y`.

9.26.1 Detailed Description

A total ordering on H79 certificates.

This binary predicate defines a total ordering on H79 certificates which is used when storing information about sets of polyhedra.

9.27 Parma_Polyhedra_Library::Linear_Expression Class Reference

A linear expression.

Public Member Functions

- Linear_Expression ()

  Default constructor: returns a copy of Linear_Expression::zero().

- Linear_Expression (const Linear_Expression &e)

  Ordinary copy-constructor.

- ~Linear_Expression ()

  Destructor.

- Linear_Expression (Coefficient_traits::const_reference n)

  Builds the linear expression corresponding to the inhomogeneous term `n`.

- Linear_Expression (const Constraint &c)

  Builds the linear expression corresponding to constraint `c`.

- Linear_Expression (const Generator &g)

  Builds the linear expression corresponding to generator `g` *(for points and closure points, the divisor is not copied).*

- Linear_Expression (const Congruence &cg)

Builds the linear expression corresponding to congruence `cg`.

- dimension_type space_dimension () const

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

- Coefficient_traits::const_reference coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- Coefficient_traits::const_reference inhomogeneous_term () const

  Returns the inhomogeneous term of `*this`.

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- bool OK () const

  Checks if all the invariants are satisfied.

- void swap (Linear_Expression &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Linear_Expression can handle.

- static const Linear_Expression & zero ()

  Returns the (zero-dimension space) constant 0.

Friends

- Linear_Expression operator+ (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the linear expression `e1 + e2`.

- Linear_Expression operator+ (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the linear expression `n + e`.

- Linear_Expression operator+ (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `e + n`.

- Linear_Expression operator- (const Linear_Expression &e)

  Returns the linear expression `- e`.

- Linear_Expression operator- (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the linear expression `e1 - e2`.

- Linear_Expression operator- (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the linear expression `n - e`.

- Linear_Expression operator- (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `e - n`.

- Linear_Expression operator * (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the linear expression `n * e`.

- Linear_Expression operator * (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `e * n`.

- Linear_Expression & operator+= (Linear_Expression &e1, const Linear_Expression &e2)

  Returns the linear expression `e1 + e2` *and assigns it to* `e1`.

- Linear_Expression & operator+= (Linear_Expression &e, Variable v)

  Returns the linear expression `e + v` *and assigns it to* `e`.

- Linear_Expression & operator+= (Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `e + n` *and assigns it to* `e`.

- Linear_Expression & operator-= (Linear_Expression &e1, const Linear_Expression &e2)

  Returns the linear expression `e1 - e2` *and assigns it to* `e1`.

- Linear_Expression & operator-= (Linear_Expression &e, Variable v)

  Returns the linear expression `e - v` *and assigns it to* `e`.

- Linear_Expression & operator-= (Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `e - n` *and assigns it to* `e`.

- Linear_Expression & operator *= (Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the linear expression `n * e` *and assigns it to* `e`.

- std::ostream & operator<< (std::ostream &s, const Linear_Expression &e)

  Output operator.

- Congruence operator%= (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the congruence $e1 = e2 \pmod{1}$.

- Congruence operator%= (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the congruence $e = n \pmod{1}$.

Related Functions

(Note that these are not member functions.)

- Linear_Expression (Variable v)

Builds the linear expression corresponding to the variable `v`.

- Linear_Expression operator+ (Variable v, Variable w)

  Returns the linear expression `v + w`.

- Linear_Expression operator+ (Variable v, const Linear_Expression &e)

  Returns the linear expression `v + e`.

- Linear_Expression operator+ (const Linear_Expression &e, Variable v)

  Returns the linear expression `e + v`.

- Linear_Expression operator+ (const Linear_Expression &e)

  Returns the linear expression `e`.

- Linear_Expression operator- (Variable v, Variable w)

  Returns the linear expression `v - w`.

- Linear_Expression operator- (Variable v, const Linear_Expression &e)

  Returns the linear expression `v - e`.

- Linear_Expression operator- (const Linear_Expression &e, Variable v)

  Returns the linear expression `e - v`.

- void swap (Parma_Polyhedra_Library::Linear_Expression &x, Parma_Polyhedra_Library::Linear_-Expression &y)

  Specializes `std::swap`.

9.27.1 Detailed Description

A linear expression.

An object of the class Linear_Expression represents the linear expression

$$\sum_{i=0}^{n-1} a_i x_i + b$$

where n is the dimension of the vector 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 Coefficient: available operators include unary negation, binary addition and subtraction, as well as multiplication by a Coefficient. 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 Coefficient have space dimension zero.

Example

The following code builds the linear expression $4x - 2y - z + 14$, having space dimension 3:

```
Linear_Expression e = 4*x - 2*y - z + 14;
```

Another way to build the same linear expression is:

```
Linear_Expression e1 = 4*x;
Linear_Expression e2 = 2*y;
Linear_Expression e3 = z;
Linear_Expression e = Linear_Expression(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 in the fifth line.

9.27.2 Constructor & Destructor Documentation

9.27.2.1 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (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 $\bowtie \in \{=, \geq, >\}$, this 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.

9.27.2.2 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (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 $d = 1$), this 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.

9.27.2.3 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const Congruence & *cg*) `[explicit]`

Builds the linear expression corresponding to congruence `cg`.

Given the congruence $cg = \bigl(\sum_{i=0}^{n-1} a_i x_i + b = 0 \pmod{m}\bigr)$, this builds the linear expression $\sum_{i=0}^{n-1} a_i x_i + b$.

9.27.3 Friends And Related Function Documentation

9.27.3.1 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (Variable *v*) `[related]`

Builds the linear expression corresponding to the variable `v`.

Exceptions:

std::length_error Thrown if the space dimension of `v` exceeds `Linear_Expression::max_space_dimension()`.

9.27.3.2 Linear_Expression & operator+= (Linear_Expression & *e*, Variable *v*) `[friend]`

Returns the linear expression `e` + `v` and assigns it to `e`.

Exceptions:

std::length_error Thrown if the space dimension of `v` exceeds `Linear_Expression::max_-space_dimension()`.

9.27.3.3 Linear_Expression & operator-= (Linear_Expression & *e*, Variable *v*) `[friend]`

Returns the linear expression `e` - `v` and assigns it to `e`.

Exceptions:

std::length_error Thrown if the space dimension of `v` exceeds `Linear_Expression::max_-space_dimension()`.

9.28 Parma_Polyhedra_Library::LP_Problem Class Reference

A Linear Programming problem.

Public Member Functions

- LP_Problem ()

  Default constructor: builds a trivial LP problem.

- LP_Problem (const Constraint_System &cs, const Linear_Expression &obj=Linear_-Expression::zero(), Optimization_Mode mode=MAXIMIZATION)

  Builds an LP problem from the constraint system `cs`*, the objective function* `obj` *and optimization mode* `mode`*.*

- LP_Problem (const LP_Problem &y)

  Ordinary copy-constructor.

- ∼LP_Problem ()

  Destructor.

- LP_Problem & operator= (const LP_Problem &y)

  Assignment operator.

- dimension_type space_dimension () const

  Returns the space dimension of the current LP problem.

- const Constraint_System & constraints () const

  Returns the constraints defining the current feasible region.

- const Linear_Expression & objective_function () const

  Returns the current objective function.

- Optimization_Mode optimization_mode () const

  Returns the current optimization mode.

- void clear ()

  Resets `*this` *to be equal to the trivial LP problem.*

- void add_constraint (const Constraint &c)

  Adds a copy of constraint `c` *to the current LP problem, increasing the number of space dimensions if needed.*

- void add_constraints (const Constraint_System &cs)

  Adds a copy of the constraints in `cs` *to the current LP problem, increasing the number of space dimensions if needed.*

- void set_objective_function (const Linear_Expression &obj)

  Sets the objective function to `obj`.

- void set_optimization_mode (Optimization_Mode mode)

  Sets the optimization mode to `mode`.

- bool is_satisfiable () const

  Checks satisfiability of `*this`.

- LP_Problem_Status solve () const

  Optimizes the current LP problem using the primal simplex algorithm.

- void evaluate_objective_function (const Generator &evaluating_point, Coefficient &num, Coefficient &den) const

  Sets `num` *and* `den` *so that* $\frac{num}{den}$ *is the result of evaluating the objective function on* `evaluating_point`.

- const Generator & feasible_point () const

  Returns a feasible point for `*this`*, if it exists.*

- const Generator & optimizing_point () const

  Returns an optimal point for `*this`*, if it exists.*

- void optimal_value (Coefficient &num, Coefficient &den) const

  Sets `num` *and* `den` *so that* $\frac{num}{den}$ *is the solution of the optimization problem.*

- bool OK () const

  Checks if all the invariants are satisfied.

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

- void swap (LP_Problem &y)

  Swaps `*this` *with* `y`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a LP_Problem can handle.

9.28.1 Detailed Description

A Linear Programming problem.

9.28.2 Constructor & Destructor Documentation

9.28.2.1 Parma_Polyhedra_Library::LP_Problem::LP_Problem () `[inline]`

Default constructor: builds a trivial LP problem.

The trivial LP problem requires to maximize the objective function 0 on the zero-dimensional vector space under no constraints at all: the origin of the vector space is the optimal solution.

9.28.2.2 Parma_Polyhedra_Library::LP_Problem::LP_Problem (const Constraint_System & *cs*, const Linear_Expression & *obj* = `Linear_Expression::zero()`, Optimization_Mode *mode* = `MAXIMIZATION`) `[inline, explicit]`

Builds an LP problem from the constraint system `cs`, the objective function `obj` and optimization mode `mode`.

Parameters:

cs The constraint system defining the feasible region for the LP problem.

obj The objective function for the LP problem (optional argument with default value 0).

mode The optimization mode (optional argument with default value `MAXIMIZATION`).

Exceptions:

std::invalid_argument Thrown if the constraint system contains any strict inequality or if the space dimension of the objective function is strictly greater than the space dimension of the constraint system.

9.28.3 Member Function Documentation

9.28.3.1 void Parma_Polyhedra_Library::LP_Problem::add_constraint (const Constraint & *c*) `[inline]`

Adds a copy of constraint `c` to the current LP problem, increasing the number of space dimensions if needed.

Exceptions:

std::invalid_argument Thrown if the constraint `c` is a strict inequality.

9.28.3.2 void Parma_Polyhedra_Library::LP_Problem::add_constraints (const Constraint_System & *cs*) [inline]

Adds a copy of the constraints in `cs` to the current LP problem, increasing the number of space dimensions if needed.

Exceptions:

std::invalid_argument Thrown if the constraint system `cs` contains any strict inequality.

9.28.3.3 void Parma_Polyhedra_Library::LP_Problem::set_objective_function (const Linear_Expression & *obj*) [inline]

Sets the objective function to `obj`.

Exceptions:

std::invalid_argument Thrown if the space dimension of `obj` is strictly greater than the space dimension of `*this`.

9.28.3.4 bool Parma_Polyhedra_Library::LP_Problem::is_satisfiable () const

Checks satisfiability of `*this`.

Returns:

`true` if and only if the LP problem is satisfiable.

9.28.3.5 LP_Problem_Status Parma_Polyhedra_Library::LP_Problem::solve () const [inline]

Optimizes the current LP problem using the primal simplex algorithm.

Returns:

An LP_Problem_Status flag indicating the outcome of the optimization attempt (unfeasible, unbounded or optimized problem).

9.28.3.6 void Parma_Polyhedra_Library::LP_Problem::evaluate_objective_function (const Generator & *evaluating_point*, Coefficient & *num*, Coefficient & *den*) const

Sets `num` and `den` so that $\frac{num}{den}$ is the result of evaluating the objective function on `evaluating_point`.

Parameters:

evaluating_point The point on which the objective function will be evaluated.

num On exit will contain the numerator of the evaluated value.

den On exit will contain the denominator of the evaluated value.

Exceptions:

std::invalid_argument Thrown if `*this` and `evaluating_point` are dimension-incompatible or if the generator `evaluating_point` is not a point.

9.28.3.7 const Generator & Parma_Polyhedra_Library::LP_Problem::feasible_point () const `[inline]`

Returns a feasible point for `*this`, if it exists.

Exceptions:

std::domain_error Thrown if the LP problem is not satisfiable.

9.28.3.8 const Generator & Parma_Polyhedra_Library::LP_Problem::optimizing_point () const `[inline]`

Returns an optimal point for `*this`, if it exists.

Exceptions:

std::domain_error Thrown if `*this` doesn't not have an optimizing point, i.e., if the LP problem is unbounded or not satisfiable.

9.28.3.9 void Parma_Polyhedra_Library::LP_Problem::optimal_value (Coefficient & *num*, Coefficient & *den*) const `[inline]`

Sets `num` and `den` so that $\frac{num}{den}$ is the solution of the optimization problem.

Exceptions:

std::domain_error Thrown if `*this` doesn't not have an optimizing point, i.e., if the LP problem is unbounded or not satisfiable.

9.29 Parma_Polyhedra_Library::NNC_Polyhedron Class Reference

A not necessarily closed convex polyhedron.

Inherits Parma_Polyhedra_Library::Polyhedron.

Public Member Functions

- NNC_Polyhedron (dimension_type num_dimensions=0, Degenerate_Element kind=UNIVERSE)

  Builds either the universe or the empty NNC polyhedron.

- NNC_Polyhedron (const Constraint_System &cs)

  Builds an NNC polyhedron from a system of constraints.

- NNC_Polyhedron (Constraint_System &cs)

  Builds an NNC polyhedron recycling a system of constraints.

- NNC_Polyhedron (const Generator_System &gs)

  Builds an NNC polyhedron from a system of generators.

- NNC_Polyhedron (Generator_System &gs)

  Builds an NNC polyhedron recycling a system of generators.

- NNC_Polyhedron (const Congruence_System &cgs)

  Builds an NNC polyhedron from a system of congruences.

- NNC_Polyhedron (Congruence_System &cgs)

  Builds an NNC polyhedron recycling a system of congruences.

- 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 & operator= (const C_Polyhedron &y)

  Assigns to `*this` *the C polyhedron* `y`*.*

- ∼NNC_Polyhedron ()

  Destructor.

- bool poly_hull_assign_if_exact (const NNC_Polyhedron &y)

  If the poly-hull of `*this` *and* `y` *is exact it is assigned to* `*this` *and* `true` *is returned, otherwise* `false` *is returned.*

- bool upper_bound_assign_if_exact (const NNC_Polyhedron &y)

  Same as poly_hull_assign_if_exact(y).

9.29.1 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.

9.29.2 Constructor & Destructor Documentation

9.29.2.1 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (dimension_type *num_dimensions* = `0`, Degenerate_Element *kind* = `UNIVERSE`) `[inline, 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.

Exceptions:

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

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

9.29.2.2 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (const Constraint_System & *cs*) `[inline, explicit]`

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.

9.29.2.3 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Constraint_System & *cs*) `[inline, explicit]`

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.

9.29.2.4 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (const Generator_System & *gs*) `[inline, explicit]`

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.

Exceptions:

std::invalid_argument Thrown if the system of generators is not empty but has no points.

9.29.2.5 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Generator_System & *gs*) `[inline, explicit]`

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.

9.29.2.6 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (const Congruence_-System & *cgs*) `[explicit]`

Builds an NNC polyhedron from a system of congruences.

The polyhedron inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

9.29.2.7 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Congruence_System & *cgs*) `[explicit]`

Builds an NNC polyhedron recycling a system of congruences.

The polyhedron inherits the space dimension of the congruence system.

Parameters:

cgs The system of congruences defining the polyhedron. It is not declared `const` because its data-structures will be recycled to build the polyhedron.

9.29.2.8 template<typename Box> Parma_Polyhedra_Library::NNC_Polyhedron::NNC_-Polyhedron (const Box & *box*, From_Bounding_Box *dummy*) `[inline]`

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.

Exceptions:

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

9.29.3 Member Function Documentation

9.29.3.1 bool Parma_Polyhedra_Library::NNC_Polyhedron::poly_hull_assign_if_exact (const NNC_Polyhedron & *y*)

If the poly-hull of `*this` and `y` is exact it is assigned to `*this` and `true` is returned, otherwise `false` is returned.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

9.30 Parma_Polyhedra_Library::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

- static Poly_Con_Relation nothing ()

  The assertion that says nothing.

- static Poly_Con_Relation is_disjoint ()

  The polyhedron and the set of points satisfying the constraint are disjoint.

- static Poly_Con_Relation strictly_intersects ()

  The polyhedron intersects the set of points satisfying the constraint, but it is not included in it.

- static Poly_Con_Relation is_included ()

  The polyhedron is included in the set of points satisfying the constraint.

- static Poly_Con_Relation saturates ()

  The polyhedron is included in the set of points saturating the constraint.

Friends

- 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.

9.30.1 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.

9.31 Parma_Polyhedra_Library::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

- static Poly_Gen_Relation nothing ()
  The assertion that says nothing.
- static Poly_Gen_Relation subsumes ()
  Adding the generator would not change the polyhedron.

Friends

- 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.

9.31.1 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.

9.32 Parma_Polyhedra_Library::Polyhedra_Powerset< PH > Class Template Reference

The powerset construction instantiated on PPL polyhedra.

Inherits Parma_Polyhedra_Library::Powerset< Parma_Polyhedra_Library::Determinate< PH > >.

Public Member Functions

Constructors

- Polyhedra_Powerset (dimension_type num_dimensions=0, Degenerate_Element kind=UNIVERSE)

  Builds a universe (top) or empty (bottom) Polyhedra_Powerset.

- Polyhedra_Powerset (const Polyhedra_Powerset &y)

  Ordinary copy-constructor.

- Polyhedra_Powerset (const PH &ph)

  If `ph` *is nonempty, builds a powerset containing only* `ph`*. Builds the empty powerset otherwise.*

- template<typename QH> Polyhedra_Powerset (const Polyhedra_Powerset< QH > &y)

  Copy-constructor allowing a source powerset with elements of a different polyhedron kind.

- Polyhedra_Powerset (const Constraint_System &cs)

  Creates a Polyhedra_Powerset with a single polyhedron with the same information contents as `cs`.

- Polyhedra_Powerset (const Congruence_System &cgs)

  with the same information contents as `cgs`.

Member Functions that Do Not Modify the Powerset of Polyhedra

- dimension_type space_dimension () const

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

- bool geometrically_covers (const Polyhedra_Powerset &y) const

Returns true *if and only if* *this *geometrically covers* y*, i.e., if any point (in some element) of* y *is also a point (of some element) of* *this*.*

- bool geometrically_equals (const Polyhedra_Powerset &y) const

  Returns true *if and only if* *this *is geometrically equal to* y*, i.e., if (the elements of)* *this *and* y *contain the same set of points.*

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by *this*.*

- memory_size_type external_memory_in_bytes () const

  Returns a lower bound to the size in bytes of the memory managed by *this*.*

- bool OK () const

  Checks if all the invariants are satisfied.

Space Dimension Preserving Member Functions that May Modify the Powerset of Polyhedra

- void add_disjunct (const PH &ph)

  Adds to *this *the disjunct* ph*.*

- void add_constraint (const Constraint &c)

  Intersects *this *with constraint* c*.*

- bool add_constraint_and_minimize (const Constraint &c)

  Intersects *this *with the constraint* c*, minimizing the result.*

- void add_constraints (const Constraint_System &cs)

  Intersects *this *with the constraints in* cs*.*

- bool add_constraints_and_minimize (const Constraint_System &cs)

  Intersects *this *with the constraints in* cs*, minimizing the result.*

- void pairwise_reduce ()

  Assign to *this *the result of (recursively) merging together the pairs of polyhedra whose poly-hull is the same as their set-theoretical union.*

- template<typename Widening> void BGP99_extrapolation_assign (const Polyhedra_Powerset &y, Widening wf, unsigned max_disjuncts)

  Assigns to *this *the result of applying the BGP99 extrapolation operator to* *this *and* y*, using the widening function* wf *and the cardinality threshold* max_disjuncts*.*

- template<typename Cert, typename Widening> void BHZ03_widening_assign (const Polyhedra_Powerset &y, Widening wf)

  Assigns to *this *the result of computing the BHZ03-widening between* *this *and* y*, using the widening function* wf *certified by the convergence certificate* Cert*.*

Member Functions that May Modify the Dimension of the Vector Space

- Polyhedra_Powerset & operator= (const Polyhedra_Powerset &y)

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

- template<typename QH> Polyhedra_Powerset & operator= (const Polyhedra_Powerset< QH > &y)

  Assignment operator allowing a source powerset with elements of a different polyhedron kind (`*this` *and* `y` *can be dimension-incompatible).*

- void swap (Polyhedra_Powerset &y)

  Swaps `*this` *with* `y`*.*

- void add_space_dimensions_and_embed (dimension_type m)

  Adds `m` *new dimensions to the vector space containing* `*this` *and embeds each polyhedron in* `*this` *in the new space.*

- void add_space_dimensions_and_project (dimension_type m)

  Adds `m` *new dimensions to the vector space containing* `*this` *without embedding the polyhedra in* `*this` *in the new space.*

- void intersection_assign (const Polyhedra_Powerset &y)

  Assigns to `*this` *the intersection of* `*this` *and* `y`*.*

- void poly_difference_assign (const Polyhedra_Powerset &y)

  Assigns to `*this` *the difference of* `*this` *and* `y`*.*

- void concatenate_assign (const Polyhedra_Powerset &y)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`*.*

- void time_elapse_assign (const Polyhedra_Powerset &y)

  Assigns to `*this` *the result of computing the time-elapse between* `*this` *and* `y`*.*

- void remove_space_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified space dimensions.

- void remove_higher_space_dimensions (dimension_type new_dimension)

  Removes the higher space dimensions so that the resulting space will have dimension `new_-dimension`*.*

- template<typename Partial_Function> void map_space_dimensions (const Partial_Function &pfunc)

  Remaps the dimensions of the vector space according to a partial function.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Polyhedra_Powerset<PH> can handle.

Related Functions

(Note that these are not member functions.)

- Widening_Function< PH > widen_fun_ref (void(PH::*wm)(const PH &, unsigned *))

  Wraps a widening method into a function object.

- Limited_Widening_Function< PH, CS > widen_fun_ref (void(PH::∗lwm)(const PH &, const CS &, unsigned ∗), const CS &cs)

  Wraps a limited widening method into a function object.

- std::pair< PH, Polyhedra_Powerset< NNC_Polyhedron > > linear_partition (const PH &p, const PH &q)

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

- bool check_containment (const NNC_Polyhedron &ph, const Polyhedra_Powerset< NNC_-Polyhedron > &ps)

  Returns `true` *if and only if the union of the NNC polyhedra in* `ps` *contains the NNC polyhedron* `ph`*.*

- bool check_containment (const PH &ph, const Polyhedra_Powerset< PH > &ps)

  Returns `true` *if and only if the union of the objects in* `ps` *contains* `ph`*.*

- void swap (Parma_Polyhedra_Library::Polyhedra_Powerset< PH > &x, Parma_Polyhedra_-Library::Polyhedra_Powerset< PH > &y)

  Specializes `std::swap`*.*

9.32.1 Detailed Description

template<typename PH> class Parma_Polyhedra_Library::Polyhedra_Powerset< PH >

The powerset construction instantiated on PPL polyhedra.

9.32.2 Constructor & Destructor Documentation

9.32.2.1 template<typename PH> Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::Polyhedra_Powerset (dimension_type *num_dimensions* = `0`, Degenerate_Element *kind* = `UNIVERSE`) `[inline, explicit]`

Builds a universe (top) or empty (bottom) Polyhedra_Powerset.

Parameters:

num_dimensions The number of dimensions of the vector space enclosing the powerset;

kind Specifies whether the universe or the empty powerset has to be built.

9.32.3 Member Function Documentation

9.32.3.1 template<typename PH> bool Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::geometrically_covers (const Polyhedra_Powerset< PH > & *y*) const `[inline]`

Returns `true` if and only if `*this` geometrically covers `y`, i.e., if any point (in some element) of `y` is also a point (of some element) of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

Warning:

This may be *really* expensive!

9.32.3.2 template<typename PH> bool Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::geometrically_equals (const Polyhedra_Powerset< PH > & *y*) const [inline]

Returns `true` if and only if `*this` is geometrically equal to `y`, i.e., if (the elements of) `*this` and `y` contain the same set of points.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

Warning:

This may be *really* expensive!

9.32.3.3 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::add_disjunct (const PH & *ph*)

Adds to `*this` the disjunct `ph`.

Exceptions:

std::invalid_argument Thrown if `*this` and `ph` are dimension-incompatible.

9.32.3.4 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::add_constraint (const Constraint & *c*)

Intersects `*this` with constraint `c`.

Exceptions:

std::invalid_argument Thrown if `*this` and constraint `c` are topology-incompatible or dimension-incompatible.

9.32.3.5 template<typename PH> bool Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::add_constraint_and_minimize (const Constraint & *c*)

Intersects `*this` with the constraint `c`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Exceptions:

std::invalid_argument Thrown if `*this` and `c` are topology-incompatible or dimension-incompatible.

9.32.3.6 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::add_constraints (const Constraint_System & *cs*)

Intersects `*this` with the constraints in `cs`.

Parameters:

cs The constraints to intersect with.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

9.32.3.7 template<typename PH> bool Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::add_constraints_and_minimize (const Constraint_System & *cs*)

Intersects `*this` with the constraints in `cs`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cs The constraints to intersect with.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

9.32.3.8 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::pairwise_reduce ()

Assign to `*this` the result of (recursively) merging together the pairs of polyhedra whose poly-hull is the same as their set-theoretical union.

On exit, for all the pairs $\mathcal{P}$, $\mathcal{Q}$ of different polyhedra in `*this`, we have $\mathcal{P} \uplus \mathcal{Q} \neq \mathcal{P} \cup \mathcal{Q}$.

9.32.3.9 template<typename PH> template<typename Widening> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::BGP99_extrapolation_assign (const Polyhedra_Powerset< PH > & *y*, Widening *wf*, unsigned *max_disjuncts*)

Assigns to `*this` the result of applying the BGP99 extrapolation operator to `*this` and `y`, using the widening function `wf` and the cardinality threshold `max_disjuncts`.

Parameters:

y A finite powerset of polyhedra. It *must* definitely entail `*this`;

wf The widening function to be used on polyhedra objects. It is obtained from the corresponding widening method by using the helper function Parma_Polyhedra_Library::widen_fun_ref. Legal values are, e.g., `widen_fun_ref(&Polyhedron::H79_widening_assign)` and `widen_fun_ref(&Polyhedron::limited_H79_extrapolation_assign, cs);`

max_disjuncts The maximum number of disjuncts occurring in the powerset `*this` *before* starting the computation. If this number is exceeded, some of the disjuncts in `*this` are collapsed (i.e., joined together).

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

For a description of the extrapolation operator, see [BGP99] and [BHZ03b].

9.32.3.10 template<typename PH> template<typename Cert, typename Widening> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::BHZ03_widening_assign (const Polyhedra_Powerset< PH > & *y*, Widening *wf*)

Assigns to `*this` the result of computing the BHZ03-widening between `*this` and `y`, using the widening function `wf` certified by the convergence certificate `Cert`.

Parameters:

y The finite powerset of polyhedra computed in the previous iteration step. It *must* definitely entail `*this`;

wf The widening function to be used on polyhedra objects. It is obtained from the corresponding widening method by using the helper function widen_fun_ref. Legal values are, e.g., `widen_fun_ref(&Polyhedron::H79_widening_assign)` and `widen_fun_ref(&Polyhedron::limited_H79_extrapolation_assign, cs)`.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

Warning:

In order to obtain a proper widening operator, the template parameter `Cert` should be a finite convergence certificate for the base-level widening function `wf`; otherwise, an extrapolation operator is obtained. For a description of the methods that should be provided by `Cert`, see BHRZ03_Certificate or H79_Certificate.

9.32.3.11 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::intersection_assign (const Polyhedra_Powerset< PH > & *y*) `[inline]`

Assigns to `*this` the intersection of `*this` and `y`.

The result is obtained by intersecting each polyhedron in `*this` with each polyhedron in `y` and collecting all these intersections.

9.32.3.12 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::poly_difference_assign (const Polyhedra_Powerset< PH > & *y*)

Assigns to `*this` the difference of `*this` and `y`.

The result is obtained by computing the poly-difference of each polyhedron in `*this` with each polyhedron in `y` and collecting all these differences.

9.32.3.13 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::concatenate_assign (const Polyhedra_Powerset< PH > & *y*)

Assigns to `*this` the concatenation of `*this` and `y`.

The result is obtained by computing the pairwise concatenation of each polyhedron in `*this` with each polyhedron in `y`.

9.32.3.14 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::time_elapse_assign (const Polyhedra_Powerset< PH > & *y*) `[inline]`

Assigns to `*this` the result of computing the time-elapse between `*this` and `y`.

The result is obtained by computing the pairwise time elapse of each polyhedron in `*this` with each polyhedron in `y`.

9.32.3.15 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::remove_space_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified space dimensions.

Parameters:

to_be_removed The set of Variable objects corresponding to the space 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`.

9.32.3.16 template<typename PH> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::remove_higher_space_dimensions (dimension_type *new_dimension*)

Removes the higher space 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`.

9.32.3.17 template<typename PH> template<typename Partial_Function> void Parma_Polyhedra_Library::Polyhedra_Powerset< PH >::map_space_dimensions (const Partial_Function & *pfunc*)

Remaps the dimensions of the vector space according to a partial function.

See also Polyhedron::map_space_dimensions.

9.32.4 Friends And Related Function Documentation

9.32.4.1 template<typename PH> Widening_Function< PH > widen_fun_ref (void(PH::*)(const PH &, unsigned *) *wm*) `[related]`

Wraps a widening method into a function object.

Parameters:

wm The widening method.

9.32.4.2 template<typename PH> Limited_Widening_Function< PH, CS > widen_fun_ref (void(PH::*)(const PH &, const CS &, unsigned *) *lwm*, const CS & *cs*) `[related]`

Wraps a limited widening method into a function object.

Parameters:

lwm The limited widening method.

cs The constraint system limiting the widening.

9.32.4.3 template<typename PH> std::pair< PH, Polyhedra_Powerset< NNC_Polyhedron > > linear_partition (const PH & *p*, const PH & *q*) `[related]`

Partitions `q` with respect to `p`.

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

- `r.first` is the intersection of `p` and `q`;
- `r.second` has the property that all its elements are 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`).

9.32.4.4 template<typename PH> bool check_containment (const PH & *ph*, const Polyhedra_Powerset< PH > & *ps*) `[related]`

Returns `true` if and only if the union of the objects in `ps` contains `ph`.

Note:

It is assumed that the template parameter PH can be converted without precision loss into an NNC_Polyhedron; otherwise, an incorrect result might be obtained.

9.33 Parma_Polyhedra_Library::Polyhedron Class Reference

The base class for convex polyhedra.

Inherited by Parma_Polyhedra_Library::C_Polyhedron, and Parma_Polyhedra_Library::NNC_Polyhedron.

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`.

- dimension_type affine_dimension () const

  Returns 0*, if* `*this` *is empty; otherwise, returns the affine dimension of* `*this`.

- const Constraint_System & constraints () const

  Returns the system of constraints.

- const Constraint_System & minimized_constraints () const

  Returns the system of constraints, with no redundant constraint.

- const Generator_System & generators () const

  Returns the system of generators.

- const Generator_System & 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 Linear_Expression &expr) const

  Returns `true` *if and only if* `expr` *is bounded from above in* `*this`.

- bool bounds_from_below (const Linear_Expression &expr) const

  Returns `true` *if and only if* `expr` *is bounded from below in* `*this`.

- bool maximize (const Linear_Expression &expr, Coefficient &sup_n, Coefficient &sup_d, bool &maximum) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from above in* `*this`*, in which case the supremum value is computed.*

- bool maximize (const Linear_Expression &expr, Coefficient &sup_n, Coefficient &sup_d, bool &maximum, Generator &point) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from above in* `*this`*, in which case the supremum value and a point where* `expr` *reaches it are computed.*

- bool minimize (const Linear_Expression &expr, Coefficient &inf_n, Coefficient &inf_d, bool &minimum) const

Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from below in* `*this`*, in which case the infimum value is computed.*

- bool minimize (const Linear_Expression &expr, Coefficient &inf_n, Coefficient &inf_d, bool &minimum, Generator &point) const

  Returns `true` *if and only if* `*this` *is not empty and* `expr` *is bounded from below in* `*this`*, in which case the infimum value and a point where* `expr` *reaches it are computed.*

- 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_COMPLEXITY) const

  Uses `*this` *to shrink a generic, interval-based bounding box. Assigns to* `box` *the intersection of* `box` *with the smallest bounding box containing* `*this`*.*

- 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_congruence (const Congruence &cg)

  Adds a copy of congruence `cg` *to the system of congruences of* `this` *(without minimizing the result).*

- void add_constraints (const Constraint_System &cs)

  Adds a copy of the constraints in `cs` *to the system of constraints of* `*this` *(without minimizing the result).*

- void add_recycled_constraints (Constraint_System &cs)

  Adds the constraints in `cs` *to the system of constraints of* `*this` *(without minimizing the result).*

- bool add_constraints_and_minimize (const Constraint_System &cs)

  Adds a copy of the constraints in `cs` *to the system of constraints of* `*this`*, minimizing the result.*

- bool add_recycled_constraints_and_minimize (Constraint_System &cs)

  Adds the constraints in `cs` *to the system of constraints of* `*this`*, minimizing the result.*

- void add_generators (const Generator_System &gs)

  Adds a copy of the generators in `gs` *to the system of generators of* `*this` *(without minimizing the result).*

- void add_recycled_generators (Generator_System &gs)

  Adds the generators in `gs` *to the system of generators of* `*this` *(without minimizing the result).*

- bool add_generators_and_minimize (const Generator_System &gs)

  Adds a copy of the generators in `gs` *to the system of generators of* `*this`*, minimizing the result.*

- bool add_recycled_generators_and_minimize (Generator_System &gs)

  Adds the generators in `gs` *to the system of generators of* `*this`*, minimizing the result.*

- void add_congruences (const Congruence_System &cgs)

  Adds to `*this` *constraints equivalent to the congruences in* `cgs` *(without 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 of* `*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 upper_bound_assign (const Polyhedron &y)

  Same as poly_hull_assign(y).

- 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 difference_assign (const Polyhedron &y)

  Same as poly_difference_assign(y).

- void affine_image (Variable var, const Linear_Expression &expr, Coefficient_traits::const_-reference denominator=Coefficient_one())

  Assigns to `*this` *the affine image of* `*this` *under the function mapping variable* `var` *to the affine expression specified by* `expr` *and* `denominator`*.*

- void affine_preimage (Variable var, const Linear_Expression &expr, Coefficient_traits::const_-reference denominator=Coefficient_one())

  Assigns to `*this` *the affine preimage of* `*this` *under the function mapping variable* `var` *to the affine expression specified by* `expr` *and* `denominator`*.*

- void generalized_affine_image (Variable var, const Relation_Symbol relsym, const Linear_-Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

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

- void generalized_affine_preimage (Variable var, const Relation_Symbol relsym, const Linear_-Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one())

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

- void generalized_affine_image (const Linear_Expression &lhs, const Relation_Symbol relsym, const Linear_Expression &rhs)

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

- void generalized_affine_preimage (const Linear_Expression &lhs, const Relation_Symbol relsym, const Linear_Expression &rhs)

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

- void bounded_affine_image (Variable var, const Linear_Expression &lb_expr, const Linear_Expression &ub_expr, Coefficient_traits::const_reference denominator=Coefficient_one())

  Assigns to `*this` *the image of* `*this` *with respect to the bounded affine relation* $\frac{\mathrm{lb\_expr}}{\mathrm{denominator}} \leq \mathrm{var}' \leq \frac{\mathrm{ub\_expr}}{\mathrm{denominator}}$.

- void bounded_affine_preimage (Variable var, const Linear_Expression &lb_expr, const Linear_Expression &ub_expr, Coefficient_traits::const_reference denominator=Coefficient_one())

  Assigns to `*this` *the preimage of* `*this` *with respect to the bounded affine relation* $\frac{\mathrm{lb\_expr}}{\mathrm{denominator}} \leq \mathrm{var}' \leq \frac{\mathrm{ub\_expr}}{\mathrm{denominator}}$.

- 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 Constraint_System &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 Constraint_System &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 Constraint_System &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 Constraint_System &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_space_dimensions_and_embed (dimension_type m)

Adds `m` *new space dimensions and embeds the old polyhedron in the new vector space.*

- void add_space_dimensions_and_project (dimension_type m)

  Adds `m` *new space dimensions to the polyhedron and does not embed it in the new vector space.*

- void concatenate_assign (const Polyhedron &y)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`*, taken in this order.*

- void remove_space_dimensions (const Variables_Set &to_be_removed)

  Removes all the specified dimensions from the vector space.

- void remove_higher_space_dimensions (dimension_type new_dimension)

  Removes the higher dimensions of the vector space so that the resulting space will have dimension `new_dimension`*.*

- template<typename Partial_Function> void map_space_dimensions (const Partial_Function &pfunc)

  Remaps the dimensions of the vector space according to a partial function.

- void expand_space_dimension (Variable var, dimension_type m)

  Creates `m` *copies of the space dimension corresponding to* `var`*.*

- void fold_space_dimensions (const Variables_Set &to_be_folded, Variable var)

  Folds the space dimensions in `to_be_folded` *into* `var`*.*

Miscellaneous Member Functions

- ~Polyhedron ()

  Destructor.

- void swap (Polyhedron &y)

  Swaps `*this` *with polyhedron* `y`*. (*`*this` *and* `y` *can be dimension-incompatible.).*

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`*.*

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`*.*

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension all kinds of Polyhedron can handle.

Protected Member Functions

- Polyhedron (Topology topol, dimension_type num_dimensions, Degenerate_Element kind)

  Builds a polyhedron having the specified properties.

- Polyhedron (const Polyhedron &y)

Ordinary copy-constructor.

- Polyhedron (Topology topol, const Constraint_System &cs)

  Builds a polyhedron from a system of constraints.

- Polyhedron (Topology topol, Constraint_System &cs)

  Builds a polyhedron recycling a system of constraints.

- Polyhedron (Topology topol, const Generator_System &gs)

  Builds a polyhedron from a system of generators.

- Polyhedron (Topology topol, Generator_System &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 different polyhedra.

- void swap (Parma_Polyhedra_Library::Polyhedron &x, Parma_Polyhedra_Library::Polyhedron &y)

  Specializes `std::swap`.

9.33.1 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);
- the topology of a polyhedron cannot be changed; rather, there are constructors for each of the two derived classes that will build a new polyhedron with the topology of that class from another polyhedron from either class and any topology;
- the only ways in which the space dimension of a polyhedron can be changed 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:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 3);
cs.insert(y >= 0);
cs.insert(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:

```
Generator_System gs;
gs.insert(point(0*x + 0*y));
gs.insert(point(0*x + 3*y));
gs.insert(point(3*x + 0*y));
gs.insert(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:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x - y <= 0);
cs.insert(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:

```
Generator_System gs;
gs.insert(point(0*x + 0*y));
gs.insert(point(0*x + y));
gs.insert(ray(x - y));
C_Polyhedron ph(gs);
```

Example 3

The following code builds the polyhedron corresponding to a 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, 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 Minkowski'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_space_dimensions_and_embed`:

```
C_Polyhedron ph(1);
ph.add_constraint(x == 2);
ph.add_space_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_space_dimensions_and_project`:

```
C_Polyhedron ph(1);
ph.add_constraint(x == 2);
ph.add_space_dimensions_and_project(1);
```

The first two lines of code are the same as in Example 4 for `add_space_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, 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));
Linear_Expression expr = x + 4;
ph.affine_image(x, expr);
```

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 $x + y$:

```
Linear_Expression expr = 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:

```
Linear_Expression expr = 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);
Linear_Expression expr = x + 4;
ph.affine_preimage(x, expr);
```

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 $x + y$

```
Linear_Expression expr = 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:

```
Linear_Expression expr = 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_space_dimensions:

```
Generator_System gs;
gs.insert(point(3*x + y +0*z + 2*w));
C_Polyhedron ph(gs);
Variables_Set to_be_removed;
to_be_removed.insert(y);
to_be_removed.insert(z);
ph.remove_space_dimensions(to_be_removed);
```

The starting polyhedron is the singleton set $\{(3, 1, 0, 2)^T\} \subseteq \mathbb{R}^4$, while the resulting polyhedron is $\{(3, 2)^T\} \subseteq \mathbb{R}^2$. Be careful when removing space dimensions *incrementally*: since dimensions are automatically renamed after each application of the remove_space_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_space_dimensions(to_be_removed1);
set<Variable> to_be_removed2;
to_be_removed2.insert(z);
ph.remove_space_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_space_dimensions` is not idempotent: removing twice the same non-empty set of dimensions is never the same as removing them just once.

9.33.2 Constructor & Destructor Documentation

9.33.2.1 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, dimension_type *num_dimensions*, Degenerate_Element *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.

9.33.2.2 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, const Constraint_-System & *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 `topol`.

9.33.2.3 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, Constraint_System & *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 `topol`.

9.33.2.4 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, const Generator_System & *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 the topology of `gs` is incompatible with `topol`, or if the system of generators is not empty but has no points.

9.33.2.5 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, Generator_System & *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 the topology of `gs` is incompatible with `topol`, or if the system of generators is not empty but has no points.

9.33.2.6 template<typename Box> Parma_Polyhedra_Library::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,
                     Coefficient& n, Coefficient& d) const
```

Let I the interval corresponding to the `k`-th space 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,
                     Coefficient& n, Coefficient& d) const
```

Let I the interval corresponding to the `k`-th space 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.

9.33.3 Member Function Documentation

9.33.3.1 Poly_Con_Relation Parma_Polyhedra_Library::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.

9.33.3.2 Poly_Gen_Relation Parma_Polyhedra_Library::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.

9.33.3.3 bool Parma_Polyhedra_Library::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.

9.33.3.4 bool Parma_Polyhedra_Library::Polyhedron::bounds_from_above (const Linear_-Expression & *expr*) const [inline]

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.

9.33.3.5 bool Parma_Polyhedra_Library::Polyhedron::bounds_from_below (const Linear_-Expression & *expr*) const [inline]

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.

9.33.3.6 bool Parma_Polyhedra_Library::Polyhedron::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*) const [inline]

Returns `true` if and only if `*this` is not empty and `expr` is bounded from above in `*this`, in which case the supremum value is computed.

Parameters:

expr The linear expression to be maximized subject to `*this`;

sup_n The numerator of the supremum value;

sup_d The denominator of the supremum value;

maximum `true` if and only if the supremum is also the maximum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from above, `false` is returned and `sup_n`, `sup_d` and `maximum` are left untouched.

9.33.3.7 bool Parma_Polyhedra_Library::Polyhedron::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *point*) const [inline]

Returns `true` if and only if `*this` is not empty and `expr` is bounded from above in `*this`, in which case the supremum value and a point where `expr` reaches it are computed.

Parameters:

expr The linear expression to be maximized subject to `*this`;

sup_n The numerator of the supremum value;

sup_d The denominator of the supremum value;

maximum `true` if and only if the supremum is also the maximum value;

point When maximization succeeds, will be assigned the point or closure point where `expr` reaches its supremum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from above, `false` is returned and `sup_n`, `sup_d`, `maximum` and `point` are left untouched.

9.33.3.8 bool Parma_Polyhedra_Library::Polyhedron::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*) const `[inline]`

Returns `true` if and only if `*this` is not empty and `expr` is bounded from below in `*this`, in which case the infimum value is computed.

Parameters:

expr The linear expression to be minimized subject to `*this`;

inf_n The numerator of the infimum value;

inf_d The denominator of the infimum value;

minimum `true` if and only if the infimum is also the minimum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from below, `false` is returned and `inf_n`, `inf_d` and `minimum` are left untouched.

9.33.3.9 bool Parma_Polyhedra_Library::Polyhedron::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *point*) const `[inline]`

Returns `true` if and only if `*this` is not empty and `expr` is bounded from below in `*this`, in which case the infimum value and a point where `expr` reaches it are computed.

Parameters:

expr The linear expression to be minimized subject to `*this`;

inf_n The numerator of the infimum value;

inf_d The denominator of the infimum value;

minimum `true` if and only if the infimum is also the minimum value;

point When minimization succeeds, will be assigned a point or closure point where `expr` reaches its infimum value.

Exceptions:

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `*this` is empty or `expr` is not bounded from below, `false` is returned and `inf_n`, `inf_d`, `minimum` and `point` are left untouched.

9.33.3.10 bool Parma_Polyhedra_Library::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.

9.33.3.11 bool Parma_Polyhedra_Library::Polyhedron::strictly_contains (const Polyhedron & *y*) const `[inline]`

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.

9.33.3.12 template<typename Box> void Parma_Polyhedra_Library::Polyhedron::shrink_bounding_box (Box & *box*, Complexity_Class *complexity* = `ANY_COMPLEXITY`) const

Uses `*this` to shrink a generic, interval-based bounding box. Assigns to `box` the intersection of `box` with the smallest bounding box containing `*this`.

Parameters:

box The bounding box to be shrunk;

complexity The complexity class of the algorithm to be used.

If the polyhedron `*this` or `box` is empty, then the empty box is returned.

If `*this` and `box` are non-empty, then, for each space dimension k with variable var, let u be the upper and l the lower bound of the smallest interval containing `*this`.

If l is infinite, then `box` is unaltered; if l is finite, then the `box` interval for space dimension k is (destructively) intersected with $[l, +\text{infty})$ if a point of `*this` satisfies var $==$ l and with $(l, +\text{infty})$ otherwise.

Similarly, if u is infinite, then `box` is unaltered; if u is finite, then the `box` interval for space dimension k is (destructively) intersected with $(-\text{infty}, u]$ if a point of `*this` satisfies var $==$ u and with $(-\text{infty}, u)$ otherwise.

The template class Box must provide the following methods, whose return values, if any, are 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,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space 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,
                  Coefficient_traits::const_reference n,
                  Coefficient_traits::const_reference d)
```

intersects the interval corresponding to the `k`-th space 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)`.

9.33.3.13 bool Parma_Polyhedra_Library::Polyhedron::OK (bool *check_not_empty* = `false`) const

Checks if all the invariants are satisfied.

Returns:

`true` if and only if `*this` satisfies all the invariants and either `check_not_empty` is `false` or `*this` is not empty.

Parameters:

check_not_empty `true` if and only if, in addition to checking the invariants, `*this` must be checked to be 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.

9.33.3.14 void Parma_Polyhedra_Library::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.

9.33.3.15 bool Parma_Polyhedra_Library::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.

9.33.3.16 void Parma_Polyhedra_Library::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.

9.33.3.17 bool Parma_Polyhedra_Library::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.

9.33.3.18 void Parma_Polyhedra_Library::Polyhedron::add_congruence (const Congruence & *cg*)

Adds a copy of congruence `cg` to the system of congruences of `this` (without minimizing the result).

Exceptions:

std::invalid_argument Thrown if `*this` and congruence `cg` are topology-incompatible or dimension-incompatible.

9.33.3.19 void Parma_Polyhedra_Library::Polyhedron::add_constraints (const Constraint_System & *cs*)

Adds a copy of the constraints in `cs` to the system of constraints of `*this` (without minimizing the result).

Parameters:

cs Contains the constraints that will be added to the system of constraints of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

9.33.3.20 void Parma_Polyhedra_Library::Polyhedron::add_recycled_constraints (Constraint_System & *cs*)

Adds the constraints in `cs` to the system of constraints of `*this` (without minimizing the result).

Parameters:

cs The constraint system that will be recycled, adding its constraints to the system of constraints of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

Warning:

The only assumption that can be made on `cs` upon successful or exceptional return is that it can be safely destroyed.

9.33.3.21 bool Parma_Polyhedra_Library::Polyhedron::add_constraints_and_minimize (const Constraint_System & *cs*)

Adds a copy of the constraints in `cs` to the system of constraints of `*this`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cs Contains the constraints that will be added to the system of constraints of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

9.33.3.22 bool Parma_Polyhedra_Library::Polyhedron::add_recycled_constraints_and_minimize (Constraint_System & *cs*)

Adds the constraints in `cs` to the system of constraints of `*this`, minimizing the result.

Returns:

`false` if and only if the result is empty.

Parameters:

cs The constraint system that will be recycled, adding its constraints to the system of constraints of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cs` are topology-incompatible or dimension-incompatible.

Warning:

The only assumption that can be made on `cs` upon successful or exceptional return is that it can be safely destroyed.

9.33.3.23 void Parma_Polyhedra_Library::Polyhedron::add_generators (const Generator_System & *gs*)

Adds a copy of the generators in `gs` to the system of generators of `*this` (without minimizing the result).

Parameters:

gs Contains the generators that will be added to the system of generators of `*this`.

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.

9.33.3.24 void Parma_Polyhedra_Library::Polyhedron::add_recycled_generators (Generator_System & *gs*)

Adds the generators in `gs` to the system of generators of `*this` (without minimizing the result).

Parameters:

gs The generator system that will be recycled, adding its generators to the system of generators of `*this`.

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.

Warning:

The only assumption that can be made on `gs` upon successful or exceptional return is that it can be safely destroyed.

9.33.3.25 bool Parma_Polyhedra_Library::Polyhedron::add_generators_and_minimize (const Generator_System & *gs*)

Adds a copy of 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 Contains the generators that will be added to the system of generators of `*this`.

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.

9.33.3.26 bool Parma_Polyhedra_Library::Polyhedron::add_recycled_generators_and_minimize (Generator_System & *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 generator system that will be recycled, adding its generators to the system of generators of `*this`.

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.

Warning:

The only assumption that can be made on `gs` upon successful or exceptional return is that it can be safely destroyed.

9.33.3.27 void Parma_Polyhedra_Library::Polyhedron::add_congruences (const Congruence_-System & *cgs*)

Adds to `*this` constraints equivalent to the congruences in `cgs` (without minimizing the result).

Parameters:

cgs Contains the congruences that will be added to the system of constraints of `*this`.

Exceptions:

std::invalid_argument Thrown if `*this` and `cgs` are topology-incompatible or dimension-incompatible.

9.33.3.28 void Parma_Polyhedra_Library::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.

9.33.3.29 bool Parma_Polyhedra_Library::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.

9.33.3.30 void Parma_Polyhedra_Library::Polyhedron::poly_hull_assign (const Polyhedron & *y*)

Assigns to `*this` the poly-hull 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.

9.33.3.31 bool Parma_Polyhedra_Library::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.

9.33.3.32 void Parma_Polyhedra_Library::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.

9.33.3.33 void Parma_Polyhedra_Library::Polyhedron::affine_image (Variable *var*, const Linear_-Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the affine image of `*this` under the function mapping variable `var` to 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 space dimension of `*this`.

9.33.3.34 void Parma_Polyhedra_Library::Polyhedron::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the affine preimage of `*this` under the function mapping variable `var` to 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 space dimension of `*this`.

9.33.3.35 void Parma_Polyhedra_Library::Polyhedron::generalized_affine_image (Variable *var*, const Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

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

Parameters:

var The left hand side variable of the generalized affine relation;

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 space dimension of `*this` or if `*this` is a C_Polyhedron and `relsym` is a strict relation symbol.

9.33.3.36 void Parma_Polyhedra_Library::Polyhedron::generalized_affine_preimage (Variable *var*, const Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`)

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

Parameters:

var The left hand side variable of the generalized affine relation;

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 space dimension of `*this` or if `*this` is a C_Polyhedron and `relsym` is a strict relation symbol.

9.33.3.37 void Parma_Polyhedra_Library::Polyhedron::generalized_affine_image (const Linear_Expression & *lhs*, const Relation_Symbol *relsym*, const Linear_Expression & *rhs*)

Assigns to `*this` the image of `*this` with respect to the generalized affine relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, 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.

9.33.3.38 void Parma_Polyhedra_Library::Polyhedron::generalized_affine_preimage (const Linear_Expression & *lhs*, const Relation_Symbol *relsym*, const Linear_Expression & *rhs*)

Assigns to `*this` the preimage of `*this` with respect to the generalized affine relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, 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.

9.33.3.39 void Parma_Polyhedra_Library::Polyhedron::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_-reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the image of `*this` with respect to the bounded affine relation $\frac{\mathrm{lb\_expr}}{\mathrm{denominator}} \leq \mathrm{var}' \leq \frac{\mathrm{ub\_expr}}{\mathrm{denominator}}$.

Parameters:

var The variable updated by the affine relation;

lb_expr The numerator of the lower bounding affine expression;

ub_expr The numerator of the upper bounding affine expression;

denominator The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1.)

Exceptions:

std::invalid_argument Thrown if `denominator` is zero or if `lb_expr` (resp., `ub_expr`) and `*this` are dimension-incompatible or if `var` is not a space dimension of `*this`.

9.33.3.40 void Parma_Polyhedra_Library::Polyhedron::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_-reference *denominator* = `Coefficient_one()`)

Assigns to `*this` the preimage of `*this` with respect to the bounded affine relation $\frac{\mathrm{lb\_expr}}{\mathrm{denominator}} \leq \mathrm{var}' \leq \frac{\mathrm{ub\_expr}}{\mathrm{denominator}}$.

Parameters:

var The variable updated by the affine relation;

lb_expr The numerator of the lower bounding affine expression;

ub_expr The numerator of the upper bounding affine expression;

denominator The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1.)

Exceptions:

std::invalid_argument Thrown if `denominator` is zero or if `lb_expr` (resp., `ub_expr`) and `*this` are dimension-incompatible or if `var` is not a space dimension of `*this`.

9.33.3.41 void Parma_Polyhedra_Library::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.

9.33.3.42 void Parma_Polyhedra_Library::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.

9.33.3.43 void Parma_Polyhedra_Library::Polyhedron::limited_BHRZ03_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *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.

9.33.3.44 void Parma_Polyhedra_Library::Polyhedron::bounded_BHRZ03_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *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.

9.33.3.45 void Parma_Polyhedra_Library::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.

9.33.3.46 void Parma_Polyhedra_Library::Polyhedron::limited_H79_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *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.

9.33.3.47 void Parma_Polyhedra_Library::Polyhedron::bounded_H79_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *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.

9.33.3.48 void Parma_Polyhedra_Library::Polyhedron::add_space_dimensions_and_embed (dimension_type *m*)

Adds `m` new space dimensions and embeds the old polyhedron in the new vector space.

Parameters:

m The number of dimensions to add.

Exceptions:

std::length_error Thrown if adding `m` new space dimensions would cause the vector space to exceed dimension `max_space_dimension()`.

The new space 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 space dimension, the result will be the polyhedron

$$\bigl\{\, (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x, y)^{\mathrm{T}} \in \mathcal{P} \,\bigr\}.$$

9.33.3.49 void Parma_Polyhedra_Library::Polyhedron::add_space_dimensions_and_project (dimension_type *m*)

Adds `m` new space dimensions to the polyhedron and does not embed it in the new vector space.

Parameters:

m The number of space dimensions to add.

Exceptions:

std::length_error Thrown if adding `m` new space dimensions would cause the vector space to exceed dimension `max_space_dimension()`.

The new space 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 space dimension, the result will be the polyhedron

$$\bigl\{\, (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \bigm| (x, y)^{\mathrm{T}} \in \mathcal{P} \,\bigr\}.$$

9.33.3.50 void Parma_Polyhedra_Library::Polyhedron::concatenate_assign (const Polyhedron & *y*)

Assigns to `*this` the concatenation of `*this` and `y`, taken in this order.

Exceptions:

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible.

std::length_error Thrown if the concatenation would cause the vector space to exceed dimension `max_space_dimension()`.

9.33.3.51 void Parma_Polyhedra_Library::Polyhedron::remove_space_dimensions (const Variables_Set & *to_be_removed*)

Removes all the specified dimensions from the vector space.

Parameters:

to_be_removed The set of Variable objects corresponding to the space 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`.

9.33.3.52 void Parma_Polyhedra_Library::Polyhedron::remove_higher_space_dimensions (dimension_type *new_dimension*)

Removes the higher dimensions of the vector space 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`.

9.33.3.53 template<typename Partial_Function> void Parma_Polyhedra_Library::Polyhedron::map_space_dimensions (const Partial_Function & *pfunc*)

Remaps the dimensions of the vector space according to a partial function.

Parameters:

pfunc The partial function specifying the destiny of each space dimension.

The template class Partial_Function 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. The `max_in_codomain()` method is called at most once.

```
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. This method is called at most n times, where n is the dimension of the vector space enclosing the polyhedron.

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

9.33.3.54 void Parma_Polyhedra_Library::Polyhedron::expand_space_dimension (Variable *var*, dimension_type *m*)

Creates m copies of the space dimension corresponding to var.

Parameters:

var The variable corresponding to the space dimension to be replicated;

m The number of replicas to be created.

Exceptions:

std::invalid_argument Thrown if var does not correspond to a dimension of the vector space.

std::length_error Thrown if adding m new space dimensions would cause the vector space to exceed dimension max_space_dimension().

If *this has space dimension n, with $n > 0$, and var has space dimension $k \leq n$, then the k-th space dimension is expanded to m new space dimensions $n, n+1, \ldots, n+m-1$.

9.33.3.55 void Parma_Polyhedra_Library::Polyhedron::fold_space_dimensions (const Variables_-Set & *to_be_folded*, Variable *var*)

Folds the space dimensions in to_be_folded into var.

Parameters:

to_be_folded The set of Variable objects corresponding to the space dimensions to be folded;

var The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions:

std::invalid_argument Thrown if *this is dimension-incompatible with var or with one of the Variable objects contained in to_be_folded. Also thrown if var is contained in to_be_-folded.

If *this has space dimension n, with $n > 0$, var has space dimension $k \leq n$, to_be_folded is a set of variables whose maximum space dimension is also less than or equal to n, and var is not a member of to_be_folded, then the space dimensions corresponding to variables in to_be_folded are folded into the k-th space dimension.

9.33.3.56 void Parma_Polyhedra_Library::Polyhedron::swap (Polyhedron & *y*) [inline]

Swaps *this with polyhedron y. (*this and y can be dimension-incompatible.).

Exceptions:

std::invalid_argument Thrown if x and y are topology-incompatible.

9.33.4 Friends And Related Function Documentation

9.33.4.1 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 ", ".

9.33.4.2 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.

9.34 Parma_Polyhedra_Library::Powerset< D > Class Template Reference

The powerset construction on a base-level domain.

Public Types

- typedef omega_iterator iterator

  Alias for a read-only *bidirectional iterator on the disjuncts of a Powerset element.*

- typedef omega_const_iterator const_iterator

  A bidirectional const_iterator on the disjuncts of a Powerset element.

- typedef std::reverse_iterator< iterator > reverse_iterator

  The reverse iterator type built from Powerset::iterator.

- typedef std::reverse_iterator< const_iterator > const_reverse_iterator

  The reverse iterator type built from Powerset::const_iterator.

Public Member Functions

Constructors and Destructor

- Powerset ()

  Default constructor: builds the bottom of the powerset constraint system (i.e., the empty powerset).

- Powerset (const Powerset &y)

  Copy constructor.

- Powerset (const D &d)

  If `d` *is not bottom, builds a powerset containing only* `d`*. Builds the empty powerset otherwise.*

- ~Powerset ()

  Destructor.

Member Functions that Do Not Modify the Powerset Element

- 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).*

- bool is_top () const

  Returns `true` *if and only if* `*this` *is the top element of the powerset constraint system (i.e., it represents the universe).*

- bool is_bottom () const

  Returns `true` *if and only if* `*this` *is the bottom element of the powerset constraint system (i.e., it represents the empty set).*

- memory_size_type total_memory_in_bytes () const

  Returns a lower bound to the total size in bytes of the memory occupied by `*this`.

- memory_size_type external_memory_in_bytes () const

  Returns a lower bound to the size in bytes of the memory managed by `*this`.

- bool OK (bool disallow_bottom=false) const

  Checks if all the invariants are satisfied.

Member Functions for the Direct Manipulation of Disjuncts

- void omega_reduce () const

  Drops from the sequence of disjuncts in `*this` *all the non-maximal elements so that* `*this` *is non-redundant.*

- size_type size () const

  Returns the number of disjuncts.

- bool empty () const

  Returns `true` *if and only if there are no disjuncts.*

- iterator begin ()

  Returns an iterator pointing to the first disjunct, if `*this` *is not empty; otherwise, returns the past-the-end iterator.*

- iterator end ()

  Returns the past-the-end iterator.

- const_iterator begin () const

  Returns a const_iterator pointing to the first disjunct, if `*this` *is not empty; otherwise, returns the past-the-end const_iterator.*

- const_iterator end () const

  Returns the past-the-end const_iterator.

- reverse_iterator rbegin ()

  Returns a reverse_iterator pointing to the last disjunct, if `*this` *is not empty; otherwise, returns the before-the-start reverse_iterator.*

- reverse_iterator rend ()

  Returns the before-the-start reverse_iterator.

- const_reverse_iterator rbegin () const

  Returns a const_reverse_iterator pointing to the last disjunct, if `*this` *is not empty; otherwise, returns the before-the-start const_reverse_iterator.*

- const_reverse_iterator rend () const

  Returns the before-the-start const_reverse_iterator.

- void add_disjunct (const D &d)

  Adds to `*this` *the disjunct* `d`.

- iterator drop_disjunct (iterator position)

  Drops the disjunct in `*this` *pointed to by* `position`*, returning an iterator to the disjunct following* `position`.

- void drop_disjuncts (iterator first, iterator last)

  Drops all the disjuncts from `first` *to* `last` *(excluded).*

- void clear ()

  Drops all the disjuncts, making `*this` *an empty powerset.*

Member Functions that May Modify the Powerset Element

- Powerset & operator= (const Powerset &y)

  The assignment operator.

- void swap (Powerset &y)

  Swaps `*this` *with* `y`.

- void least_upper_bound_assign (const Powerset &y)

  Assigns to `*this` *the least upper bound of* `*this` *and* `y`.

- 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`.

- void collapse ()

  If `*this` *is not empty (i.e., it is not the bottom element), it is reduced to a singleton obtained by computing an upper-bound of all the disjuncts.*

Protected Types

- typedef std::list< D > Sequence

  A powerset is implemented as a sequence of elements.

- typedef Sequence::iterator Sequence_iterator

  Alias for the low-level iterator on the disjuncts.

- typedef Sequence::const_iterator Sequence_const_iterator

  Alias for the low-level const_iterator on the disjuncts.

Protected Member Functions

- bool is_omega_reduced () const

  Returns `true` *if and only if* `*this` *does not contain non-maximal elements.*

- void collapse (unsigned max_disjuncts)

  Upon return, `*this` *will contain at most* `max_disjuncts` *elements; the set of disjuncts in positions greater than or equal to* `max_disjuncts`*, will be replaced at that position by their upper-bound.*

- iterator add_non_bottom_disjunct (const D &d, iterator first, iterator last)

  Adds to `*this` *the disjunct* `d`*, assuming* `d` *is not the bottom element and ensuring partial Omega-reduction.*

- void add_non_bottom_disjunct (const D &d)

  Adds to `*this` *the disjunct* `d`*, assuming* `d` *is not the bottom element.*

- template<typename Binary_Operator_Assign> void pairwise_apply_assign (const Powerset &y, Binary_Operator_Assign op_assign)

  Assigns to `*this` *the result of applying* `op_assign` *pairwise to the elements in* `*this` *and* `y`*.*

Protected Attributes

- Sequence sequence

  The sequence container holding powerset's elements.

- bool reduced

  If `true`*,* `*this` *is Omega-reduced.*

Related Functions

(Note that these are not member functions.)

- bool operator== (const Powerset< D > &x, const Powerset< D > &y)

  Returns `true` *if and only if* `x` *and* `y` *are equivalent.*

- bool operator!= (const Powerset< D > &x, const Powerset< D > &y)

  Returns `true` *if and only if* `x` *and* `y` *are not equivalent.*

- std::ostream & operator<< (std::ostream &s, const Powerset< D > &x)

  Output operator.

- void swap (Parma_Polyhedra_Library::Powerset< D > &x, Parma_Polyhedra_Library::Powerset< D > &y)

  Specializes `std::swap`*.*

9.34.1 Detailed Description

template<typename D> class Parma_Polyhedra_Library::Powerset< D >

The powerset construction on a base-level domain.

This class offers a generic implementation of a *powerset* domain as defined in Section The Powerset Construction.

Besides invoking the available methods on the disjuncts of a Powerset, this class also provides bidirectional iterators that allow for a direct inspection of these disjuncts. For a consistent handling of Omega-reduction, all the iterators are *read-only*, meaning that the disjuncts cannot be overwritten. Rather, by using the class `iterator`, it is possible to drop one or more disjuncts (possibly so as to later add back modified versions). As an example of iterator usage, the following templatic function drops from powerset `ps` all the disjuncts that would have become redundant by the addition of an external element `d`.

```
template <typename D>
void
drop_subsumed(Powerset<D>& ps, const D& d) {
  for (typename Powerset<D>::iterator i = ps.begin(),
         ps_end = ps.end(), i != ps_end; )
    if (i->definitely_entails(d))
      i = ps.drop_disjunct(i);
    else
      ++i;
}
```

The template class D must provide the following methods.

```
    memory_size_type total_memory_in_bytes() const
```

Returns a lower bound on the total size in bytes of the memory occupied by the instance of D.

```
    bool is_top() const
```

Returns `true` if and only if the instance of D is the top element of the domain.

```
    bool is_bottom() const
```

Returns `true` if and only if the instance of D is the bottom element of the domain.

```
    bool definitely_entails(const D& y) const
```

Returns `true` if the instance of D definitely entails `y`. Returns `false` if the instance may not entail `y` (i.e., if the instance does not entail `y` or if entailment could not be decided).

```
    void upper_bound_assign(const D& y)
```

Assigns to the instance of D an upper bound of the instance and `y`.

```
    void meet_assign(const D& y)
```

Assigns to the instance of D the meet of the instance and `y`.

```
    bool OK() const
```

Returns `true` if the instance of D is in a consistent state, else returns `false`.

The following operators on the template class D must be defined.

```
operator<<(std::ostream& s, const D& x)
```

Writes a textual representation of the instance of D on `s`.

```
operator==(const D& x, const D& y)
```

Returns `true` if and only if `x` and `y` are equivalent D's.

```
operator!=(const D& x, const D& y)
```

Returns `true` if and only if `x` and `y` are different D's.

9.34.2 Member Typedef Documentation

9.34.2.1 template<typename D> typedef std::list<D> Parma_Polyhedra_Library::Powerset< D >::Sequence `[protected]`

A powerset is implemented as a sequence of elements.

The particular sequence employed must support efficient deletion in any position and efficient back insertion.

9.34.2.2 template<typename D> typedef omega_iterator Parma_Polyhedra_Library::Powerset< D >::iterator

Alias for a *read-only* bidirectional iterator on the disjuncts of a Powerset element.

By using this iterator type, the disjuncts cannot be overwritten, but they can be removed using methods `drop_disjunct(iterator position)` and `drop_disjuncts(iterator first, iterator last)`, while still ensuring a correct handling of Omega-reduction.

9.34.3 Member Function Documentation

9.34.3.1 template<typename D> void Parma_Polyhedra_Library::Powerset< D >::omega_reduce () const

Drops from the sequence of disjuncts in `*this` all the non-maximal elements so that `*this` is non-redundant.

This method is declared `const` because, even though Omega-reduction may change the syntactic representation of `*this`, its semantics will be unchanged.

9.34.3.2 template<typename D> void Parma_Polyhedra_Library::Powerset< D >::upper_-bound_assign (const Powerset< D > & *y*) `[inline]`

Assigns to `*this` an upper bound of `*this` and `y`.

The result will be the least upper bound of `*this` and `y`.

9.34.3.3 template<typename D> Powerset< D >::iterator Parma_Polyhedra_Library::Powerset< D >::add_non_bottom_disjunct (const D & *d*, iterator *first*, iterator *last*) `[protected]`

Adds to `*this` the disjunct `d`, assuming `d` is not the bottom element and ensuring partial Omega-reduction.

If `d` is not the bottom element and is not Omega-redundant with respect to elements in positions between `first` and `last`, all elements in these positions that would be made Omega-redundant by the addition of `d` are dropped and `d` is added to the reduced sequence.

9.34.3.4 template<typename D> template<typename Binary_Operator_Assign> void Parma_Polyhedra_Library::Powerset< D >::pairwise_apply_assign (const Powerset< D > & *y*, Binary_Operator_Assign *op_assign*) `[protected]`

Assigns to `*this` the result of applying `op_assign` pairwise to the elements in `*this` and `y`.

The elements of the powerset result are obtained by applying `op_assign` to each pair of elements whose components are drawn from `*this` and `y`, respectively.

9.35 Parma_Polyhedra_Library::Throwable Class Reference

User objects the PPL can throw.

Public Member Functions

- virtual void throw_me () const =0

  Throws the user defined exception object.

- virtual ~Throwable ()

  Virtual destructor.

9.35.1 Detailed Description

User objects the PPL can throw.

This abstract base class should be instantiated by those users willing to provide a polynomial upper bound to the time spent by any invocation of a library operator.

9.36 Parma_Polyhedra_Library::Variable Class Reference

A dimension of the vector space.

Public Types

- typedef void output_function_type (std::ostream &s, const 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.

- dimension_type space_dimension () const

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

- memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`*.*

- memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`*.*

- bool OK () const

  Checks if all the invariants are satisfied.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Variable can handle.

- static void set_output_function (output_function_type ∗p)

  Sets the output function to be used for printing Variable objects.

- static output_function_type ∗ get_output_function ()

  Returns the pointer to the current output function.

Friends

- std::ostream & operator<< (std::ostream &s, const Variable &v)

  Output operator.

Related Functions

(Note that these are not member functions.)

- bool less (Variable v, Variable w)

  Defines a total ordering on variables.

Classes

- struct Compare

  Binary predicate defining the total ordering on variables.

9.36.1 Detailed Description

A dimension of the vector space.

An object of the class Variable represents a dimension of the space, that is one of the Cartesian axes. Variables are used as basic 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). The space dimension of a variable is the dimension of the vector space made by all the Cartesian axes having an index less than or equal to that of the considered variable; thus, if a variable has index i, its space dimension is $i + 1$.

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);
Linear_Expression e1 = x + y;
Linear_Expression e2 = y + z;
```

9.36.2 Constructor & Destructor Documentation

9.36.2.1 Parma_Polyhedra_Library::Variable::Variable (dimension_type *i*) `[inline, explicit]`

Builds the variable corresponding to the Cartesian axis of index `i`.

Exceptions:

std::length_error Thrown if the `i+1` exceeds `Variable::max_space_dimension()`.

9.36.3 Member Function Documentation

9.36.3.1 dimension_type Parma_Polyhedra_Library::Variable::space_dimension () const `[inline]`

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

The returned value is `id()+1`.

9.37 Parma_Polyhedra_Library::Variable::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`*.*

9.37.1 Detailed Description

Binary predicate defining the total ordering on variables.

10 PPL Page Documentation

10.1 GNU General Public License

Version 2, June 1991

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.

10.2 GNU Free Documentation License

widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index