The Parma Polyhedra Library User's Manual* (version 0.11.2)

Roberto Bagnara†
Patricia M. Hill‡
Enea Zaffanella§

based on previous work also by

Elisa Ricci

and

Sara Bonini
Andrea Pescetti
Angela Stazzone
Tatiana Zolo

February 27, 2011

*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"; EPSRC project "Numerical Domains for Software Analysis"; EPSRC project "Geometric Abstractions for Scalable Program Analyzers".

†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, Java, OCaml and a number of Prolog systems;
- it is free software: distributed under the terms of the GNU General Public License.

In the following section we describe all the domains available to the PPL user. More detailed descriptions of these domains and the operations provided will be found in subsequent sections.

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

1.1.1 Semantic Geometric Descriptors

A *semantic geometric descriptor* is a subset of $\mathbb{R}^n$. The PPL provides several classes of semantic GDs. These are identified by their C++ class name, together with the class template parameters, if any. These classes include the *simple classes*:

- `C_Polyhedron`,

- `NNC_Polyhedron` ,
- `BD_Shape<T>` ,
- `Octagonal_Shape<T>` ,
- `Box<ITV>` , and
- `Grid` ,

where:

- `T` is a numeric type chosen among `mpz_class`, `mpq_class`, `signed char`, `short`, `int`, `long`, `long` long (or any of the C99 exact width integer equivalents `int8_t`, int16_t, and so forth); and
- `ITV` is an instance of the `Interval` template class.

Other semantic GDs, the *compound classes*, can be constructed (also recursively) from all the GDs classes. These include:

- `Pointset_Powerset<PSET>` ,
- `Partially_Reduced_Product<D1, D2, R>` ,

where `PSET`, `D1` and `D2` can be any semantic GD classes and `R` is the reduction operation to be applied to the component domains of the product class.

A uniform set of operations is provided for creating, testing and maintaining each of the semantic GDs. However, as many of these depend on one or more syntactic GDs, we first describe the syntactic GDs.

1.1.2 Syntactic Geometric Descriptors

A *syntactic geometric descriptor* is for defining, modifying and inspecting a semantic GD. There are three kinds of *syntactic GDs*: *basic GDs*, *constraint GDs* and *generator GDs*. Some of these are *generic* and some *specific*. A generic syntactic GD can be used (in the appropriate context) with any semantic GD; clearly, different semantic GDs will usually provide different levels of support for the different subclasses of generic GDs. In contrast, the use of a specific GD may be restricted to apply to a given subset of the semantic GDs (i.e., some semantic GDs provide no support at all for them).

1.1.2.1 Basic Geometric Descriptors

The following basic GDs currently supported by the PPL are:

- space dimension;
- variable and variable set;
- coefficient;
- linear expression;
- relation symbol;
- vector point.

These classes, which are all generic syntactic GDs, are used to build the constraint and generator GDs as well as support many generic operations on the semantic GDs.

1.1.2.2 Constraint Geometric Descriptors

The PPL currently supports the following classes of *generic* constraint GDs:

- linear constraint;
- linear congruence.

Each linear constraint can be further classified to belong to one or more of the following syntactic subclasses:

- inconsistent constraints (e.g., $0 \geq 2$);
- tautological constraints (e.g., $0 \leq 2$);
- interval constraints (e.g., $x \leq 2$);
- bounded-difference constraints (e.g., $x - y \leq 2$);
- octagonal constraints (e.g., $x + y \leq 2$);
- linear equality constraints (e.g., $x = 2$);
- non-strict linear inequality constraints (e.g., $x - 3y \leq 2$);
- strict linear inequality constraints (e.g., $x - 3y < 2$).

Note that the subclasses are not disjoint.

Similarly, each linear congruence can be classified to belong to one or more of the following syntactic subclasses:

- inconsistent congruences (e.g., $0 \equiv_2 1$);
- tautological congruences (e.g., $0 \equiv_2 2$);
- linear equality, i.e., non-proper congruences (e.g., $x + 3y \equiv_0 0$);
- proper congruences (e.g., $x + 3y \equiv_5 0$).

The library also supports systems, i.e., finite collections, of either linear constraints or linear congruences (but see the note below).

Each semantic GD provides *optimal* support for some of the subclasses of generic syntactic GDs listed above: here, the word "optimal" means that the considered semantic GD computes the *best upward approximation* of the exact meaning of the linear constraint or congruence. When a semantic GD operation is applied to a syntactic GD that is not optimally supported, it will either indicate its unsuitability (e.g., by throwing an exception) or it will apply an upward approximation semantics (possibly not the best one).

For instance, the semantic GD of topologically closed convex polyhedra provides optimal support for non-strict linear inequality and equality constraints, but it does not provide optimal support for strict inequalities. Some of its operations (e.g., `add_constraint` and `add_congruence`) will throw an exception if supplied with a non-trivial strict inequality constraint or a proper congruence; some other operations (e.g., `refine_with_constraint` or `refine_with_congruence`) will compute an over-approximation.

Similarly, the semantic GD of rational boxes (i.e., multi-dimensional intervals) having integral values as interval boundaries provides optimal support for all interval constraints: even though the interval constraint $2x \leq 5$ cannot be represented exactly, it will be optimally approximated by the constraint $x \leq 3$.

Note

When providing an upward approximation for a constraint or congruence, we consider it in isolation: in particular, the approximation of each element of a system of GDs is independent from the other elements; also, the approximation is independent from the current value of the semantic GD.

1.1.2.3 Generator Geometric Descriptors

The PPL currently supports two classes of generator GDs:

- polyhedra generator: these are polyhedra points, rays and lines;
- grid generator: these are grid points, parameters and lines.

Rays, lines and parameters are specific of the mentioned semantic GDs and, therefore, they cannot be used by other semantic GDs. In contrast, as already mentioned above, points are basic geometric descriptors since they are also used in *generic* PPL operations.

1.1.3 Generic Operations on Semantic Geometric Descriptors

1. Constructors of a universe or empty semantic GD with the given space dimension.
2. Operations on a semantic GD that do not depend on the syntactic GDs.
   - `is_empty()`, `is_universe()`, `is_topologically_closed()`, `is_discrete()`, `is_bounded()`, `contains_integer_point()`
     test for the named properties of the semantic GD.
   - `total_memory_in_bytes()`, `external_memory_in_bytes()`
     return the total and external memory size in bytes.
   - `OK()`
     checks that the semantic GD has a valid internal representation. (Some GDs provide this method with an optional Boolean argument that, when true, requires to also check for non-emptiness.)
   - `space_dimension()`, `affine_dimension()`
     return, respectively, the space and affine dimensions of the GD.
   - `add_space_dimensions_and_embed()`, `add_space_dimensions_and_project()`, `expand_space_dimension()`, `remove_space_dimensions()`, `fold_space_dimensions()`, `map_space_dimensions()`
     modify the space dimensions of the semantic GD; where, depending on the operation, the arguments can include the number of space dimensions to be added or removed a variable or set of variables denoting the actual dimensions to be used and a partial function defining a mapping between the dimensions.
   - `contains()`, `strictly_contains()`, `is_disjoint_from()`
     compare the semantic GD with an argument semantic GD of the same class.
   - `topological_closure_assign()`, `intersection_assign()`, `upper_bound_assign()`, `difference_assign()`, `time_elapse_assign()`, `widening_assign()`, `concatenate_assign()`, `swap()`
     modify the semantic GD, possibly with an argument semantic GD of the same class.
   - `constrains()`, `bounds_from_above()`, `bounds_from_below()`, `maximize()`, `minimize()`.
     These find information about the bounds of the semantic GD where the argument variable or linear expression define the direction of the bound.

- `affine_image()`, `affine_preimage()`, `generalized_affine_image()`, `generalized_affine_preimage()`, `bounded_affine_image()`, `bounded_affine_preimage()`.

  These perform several variations of the affine image and preimage operations where, depending on the operation, the arguments can include a variable representing the space dimension to which the transformation will be applied and linear expressions with possibly a relation symbol and denominator value that define the exact form of the transformation.

- `ascii_load()`, `ascii_dump()`

  are the ascii input and output operations.

3. Constructors of a semantic GD of one class from a semantic GD of any other class. These constructors obey an *upward approximation semantics*, meaning that the constructed semantic GD is guaranteed to contain all the points of the source semantic GD, but possibly more. Some of these constructors provide a complexity parameter with which the application can control the complexity/precision trade-off for the construction operation: by using the complexity parameter, it is possible to keep the construction operation in the polynomial or the simplex worst-case complexity class, possibly incurring into a further upward approximation if the precise constructor is based on an algorithm having exponential complexity.

4. Constructors of a semantic GD from a constraint GD; either a linear constraint system or a linear congruence system. These constructors assume that the given semantic GD provides optimal support for the argument syntactic GD: if that is not the case, an invalid argument exception is thrown.

5. Other interaction between the semantic GDs and constraint GDs.

   - `add_constraint()`, `add_constraints()`, `add_recycled_constraints()`, `add_congruence()`, `add_congruences()`, `add_recycled_congruences()`.

     These methods assume that the given semantic GD provides optimal support for the argument syntactic GD: if that is not the case, an invalid argument exception is thrown.

     For `add_recycled_constraints()` and `add_recycled_congruences()`, the only assumption that can be made on the constraint GD after return (successful or exceptional) is that it can be safely destroyed.

   - `refine_with_constraint()`, `refine_with_constraints()`, `refine_with_congruence()`, `refine_with_congruences()`.

     If the argument constraint GD is optimally supported by the semantic GD, the methods behave the same as the corresponding `add_*` methods listed above. Otherwise the constraint GD is used only to a limited extent to refine the semantic GD; possibly not at all. Notice that, while repeating an add operation is pointless, this is not true for the refine operations. For example, in those cases where

     ```
     Semantic_GD.add_constraint(c)
     ```

     raises an exception, a fragment of the form

     ```
     Semantic_GD.refine_with_constraint(c)
     // Other add_constraint(s) or refine_with_constraint(s) operations
     // on Semantic_GD.
     Semantic_GD.refine_with_constraint(c)
     ```

     may give more precise results than a single

     ```
     Semantic_GD.refine_with_constraint(c).
     // Other add_constraint(s) or refine_with_constraint(s) operations
     // on Semantic_GD.
     ```

   - `constraints()`, `minimized_constraints()`, `congruences()`, `minimized_congruences()`.

     Returns the indicated system of constraint GDs satisfied by the semantic GD.

- `can_recycle_constraint_systems()`, `can_recycle_congruence_systems()`.

  Return true if and only if the semantic GD can recycle the indicated constraint GD.

- `relation_with()`.

  This takes a constraint GD as an argument and returns the relations holding between the semantic GD and the constraint GD. The possible relations are: `IS_INCLUDED()`, `SATURATES()`, `STRICTLY_INTERSECTS()`, `IS_DISJOINT()` and `NOTHING()`. This operator also can take a polyhedron generator GD as an argument and returns the relation `SUBSUMES()` or `NOTHING()` that holds between the generator GD and the semantic GD.

1.2 Upward Approximation

The Parma Polyhedra Library, for those cases where an exact result cannot be computed within the specified complexity limits, computes an *upward approximation* of the exact result. For semantic GDs this means that the computed result is a possibly strict superset of the set of points of $\mathbb{R}^n$ that constitutes the exact result. Notice that the PPL does not provide direct support to compute *downward approximations* (i.e., possibly strict subsets of the exact results). While downward approximations can often be computed from upward ones, the required algorithms and the conditions upon which they are correct are outside the current scope of the PPL. Beware, in particular, of the following possible pitfall: the library provides methods to compute upward approximations of set-theoretic difference, which is antitone in its second argument. Applying a difference method to a second argument that is not an exact representation or a downward approximation of reality, would yield a result that, of course, is not an upward approximation of reality. It is the responsibility of the library user to provide the PPL's method with approximations of reality that are consistent with respect to the desired results.

1.3 Approximating Integers

The Parma Polyhedra Library provides support for approximating integer computations using the geometric descriptors it provides. In this section we briefly explain these facilities.

1.3.1 Dropping Non-Integer Points

When a geometric descriptor is used to approximate integer quantities, all the points with non-integral coordinates represent an imprecision of the description. Of course, removing all these points may be impossible (because of convexity) or too expensive. The PPL provides the operator `drop_some_non_integer_points` to possibly tighten a descriptor by dropping some points with non-integer coordinates, using algorithms whose complexity is bounded by a parameter. The set of dimensions that represent integer quantities can be optionally specified. It is worth to stress the role of *some* in the operator name: in general no optimality guarantee is provided.

1.3.2 Approximating Bounded Integers

The Parma Polyhedra Library provides services that allow to compute correct approximations of bounded arithmetic as available in widespread programming languages. Supported bit-widths are 8, 16, 32 and 64 bits, with some limited support for 128 bits. Supported representations are binary unsigned and two's complement signed. Supported overflow behaviors are:

Wrapping: this means that, for a w-bit bounded integer, the computation happens modulo 2^w. In turn, this signifies that the computation happens *as if* the unbounded arithmetic result was computed and then wrapped. For unsigned integers, the wrapping function is simply $x \bmod 2^w$, most conveniently defined as

$$\mathrm{wrap}^{\mathrm{u}}_w(x) \stackrel{\mathrm{def}}{=} x - 2^w \lfloor x/2^w \rfloor.$$

For signed integers the wrapping function is, instead,

$$\mathrm{wrap}^{\mathrm{s}}_w(x) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{wrap}^{\mathrm{u}}_w(x), & \text{if } \mathrm{wrap}^{\mathrm{u}}_w(x) < 2^{w-1}; \\ \mathrm{wrap}^{\mathrm{u}}_w(x) - 2^w, & \text{otherwise.} \end{cases}$$

Undefined: this means that the result of the operation resulting in an overflow can take any value. This is useful to partially model systems where overflow has unspecified effects on the computed result. Even though something more serious can happen in the system being analyzed ---due to, e.g., C's undefined behavior---, here we are only concerned with the results of arithmetic operations. It is the responsibility of the analyzer to ensure that other manifestations of undefined behavior are conservatively approximated.

Impossible: this is for the analysis of languages where overflow is trapped before it affects the state, for which, thus, any indication that an overflow may have affected the state is necessarily due to the imprecision of the analysis.

1.3.2.1 Wrapping Operator

One possibility for precisely approximating the semantics of programs that operate on bounded integer variables is to follow the approach described in [SK07]. The idea is to associate space dimensions to the *unwrapped values* of bounded variables. Suppose `j` is a w-bit, unsigned program variable associated to a space dimension labeled by the variable x. If x is constrained by some numerical abstraction to take values in a set $S \subseteq \mathbb{R}$, then the program variable `j` can only take values in $\{ \mathrm{wrap}^{\mathrm{u}}_w(z) \mid z \in S \}$. There are two reasons why this is interesting: firstly, this allows for the retention of relational information by using a single numerical abstraction tracking multiple program variables. Secondly, the integers modulo 2^w form a ring of equivalence classes on which addition and multiplication are well defined. This means, e.g., that assignments with affine right-hand sides and involving only variables with the same bit-width and representation can be safely modeled by affine images. While upper bounds and widening can be used without any precaution, anything that can be reconducted to intersection requires a preliminary *wrapping* phase, where the dimensions corresponding to bounded integer types are brought back to their natural domain. This necessity arises naturally for the analysis of conditionals and conversion operators, as well as in the realization of domain combinations.

The PPL provides a general wrapping operator that is parametric with respect to the set of space dimensions (variables) to be wrapped, the width, representation and overflow behavior of all these variables. An optional constraint system can, when given, improve the precision. This constraint system, which must only depend on variables with respect to which wrapping is performed, is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation afterwards. The general wrapping operator offered by the PPL also allows control of the complexity/precision ratio by means of two additional parameters: an unsigned integer encoding a complexity threshold, with higher values resulting in possibly improved precision; and a Boolean controlling whether space dimensions should be wrapped individually, something that results in much greater efficiency to the detriment of precision, or collectively.

Note that the PPL assumes that any space dimension subject to wrapping is being used to capture the value of bounded integer values. As a consequence the library is free to drop, from the involved numerical abstraction, any point having a non-integer coordinate that corresponds to a space dimension subject to wrapping. It must be stressed that freedom to drop such points does not constitute an obligation to remove all of them (especially because this would be extraordinarily expensive on some numerical abstractions). The PPL provides operators for the more systematic removal of points with non-integral coordinates.

The wrapping operator will only remove some of these points as a by-product of its main task and only when this comes at a negligible extra cost.

1.4 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].

1.4.1 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 $i \in \{0, \dots, n-1\}$, v_i denotes the i-th component of the (column) vector $\boldsymbol{v} = (v_0, \dots, v_{n-1})^{\mathrm{T}} \in \mathbb{R}^n$. 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: $S_1 + S_2 = \{\, \boldsymbol{v}_1 + \boldsymbol{v}_2 \mid \boldsymbol{v}_1 \in S_1, \boldsymbol{v}_2 \in S_2 \,\}$.

1.4.2 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 $\varnothing$.

1.4.3 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 $\emptyset$ and the vector space $\mathbb{R}^n$ are, respectively, the smallest and the biggest elements of both $\mathbb{P}_n$ and $\mathbb{CP}_n$. The vector space $\mathbb{R}^n$ is also called the *universe* polyhedron.

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

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.

1.4.4 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*.

1.5 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.

1.5.1 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

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

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.

1.5.2 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 $\operatorname{linear.hull}(S)$ (resp., $\operatorname{conic.hull}(S)$, $\operatorname{affine.hull}(S)$, $\operatorname{convex.hull}(S)$) 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 $\operatorname{nnc.hull}(P, C)$ 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 $\operatorname{nnc.hull}(P, C) = \operatorname{nnc.hull}(P, P \cup C)$ so that, if $C \subseteq P$,

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

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

1.5.3 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).

1.5.4 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 $\mathcal{P}$.

1.5.5 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 $C' \subseteq C$.

1.5.6 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.

1.5.7 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.

1.5.8 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.

1.5.9 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 $\dim(\mathcal{P}) = \dim(\mathcal{Q})$.

1.5.10 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.

1.6 Operations on Convex Polyhedra

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

1.6.1 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 $\mathbb{CP}_n$.

1.6.2 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.

1.6.3 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

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

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 $\mathcal{Q}$.

1.6.4 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

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

1.6.5 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

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

1.6.6 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 $\rho \colon \{0,\dots,n-1\} \rightarrowtail \mathbb{N}$ such that $\rho\bigl(\{0,\dots,n-1\}\bigr) = \{0,\dots,m-1\}$ with $m \leq n$. Dimensions corresponding to indices that are not mapped by ρ are removed.

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

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

1.6.7 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,

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

This operation has been proposed in [GDDetal04].

1.6.8 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 $J = \{j_0, \dots, j_{m-1}\}$, 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

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

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

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

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

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

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

This operation has been proposed in [GDDetal04].

1.6.9 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,

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

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

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

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 $i \in \{0, \dots, n-1\}$, 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.

1.6.10 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* $\phi = \bigl(x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b\bigr) : \mathbb{R}^n \to \mathbb{R}^n$ 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.

1.6.11 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

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

1.6.12 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.

1.6.13 Cylindrification Operator

The operator `unconstrain` computes the *cylindrification* [HMT71] of a polyhedron with respect to one of its variables. Formally, the cylindrification $\mathcal{Q} \in \mathbb{P}_n$ of an NNC polyhedron $\mathcal{P} \in \mathbb{P}_n$ with respect to variable index $i \in \{0, \ldots, n-1\}$ is defined as follows:

$$\mathcal{Q} = \bigl\{ \boldsymbol{w} \in \mathbb{R}^n \bigm| \exists \boldsymbol{v} \in \mathcal{P} \,.\, \forall j \in \{0, \ldots, n-1\} : j \neq i \implies w_j = v_j \bigr\}.$$

Cylindrification is an idempotent operation; in particular, note that the computed result has the same space dimension of the original polyhedron. A variant of the operator above allows for the cylindrification of a polyhedron with respect to a finite set of variables.

1.6.14 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.

1.6.15 Meet-Preserving Enlargement and Simplification

Let $\mathcal{P}, \mathcal{Q}, \mathcal{R} \in \mathbb{P}_n$ be NNC polyhedra. Then:

- $\mathcal{R}$ is *meet-preserving* with respect to $\mathcal{P}$ using context $\mathcal{Q}$ if $\mathcal{R} \cap \mathcal{Q} = \mathcal{P} \cap \mathcal{Q}$;

- $\mathcal{R}$ is an *enlargement* of $\mathcal{P}$ if $\mathcal{R} \supseteq \mathcal{P}$.

- $\mathcal{R}$ is a *simplification* with respect to $\mathcal{P}$ if $r \leq p$, where r and p are the cardinalities of minimized constraint representations for $\mathcal{R}$ and $\mathcal{P}$, respectively.

Notice that an enlargement need not be a simplification, and vice versa; moreover, the identity function is (trivially) a meet-preserving enlargement and simplification.

The library provides a binary operator (`simplify_using_context`) for the domain of NNC polyhedra that returns a polyhedron which is a meet-preserving enlargement simplification of its first argument using the second argument as context.

The concept of meet-preserving enlargement and simplification also applies to the other basic domains (boxes, grids, BD and octagonal shapes). See below for a definition of the concept of meet-preserving simplification for powerset domains.

1.6.16 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 $\mathcal{P}$.

1.6.17 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 $\mathcal{P} \ \nabla \ \mathcal{Q}$ 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 $\mathcal{P} \nabla \mathcal{Q}$ 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 ($\mathcal{P} \supseteq \mathcal{P} \nabla \mathcal{Q}$ 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 language interfaces.

1.6.18 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 wild card 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.

1.6.19 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 intersecting the result of the limited extrapolation operation with the box obtained as a result of applying the CC76-widening to the smallest boxes enclosing the two argument polyhedra.

1.7 Intervals and Boxes

The PPL provides support for computations on non-relational domains, called boxes, and also the interval domains used for their representation.

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 $\mathbb{R}$. For each vector $\boldsymbol{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, and for each relation symbol $\bowtie \in \{=, \geq, >\}$, the constraint $\langle \boldsymbol{a}, \boldsymbol{x} \rangle \bowtie b$ is said to be a *interval constraint* if there exist an index $i \in \{0, \ldots, n-1\}$ such that, for all $k \in \{0, \ldots, i-1, i+1, \ldots, n-1\}$, $a_k = 0$. Thus each interval constraint that is not a tautology or inconsistent has the form $x = r$, $x \leq r$, $x \geq r$, $x < r$ or $x > r$, with $r \in \mathbb{R}$.

Letting $\mathcal{B}$ be a sequence of n intervals and $\boldsymbol{e}_i = (0, \ldots, 1, \ldots, 0)^{\mathrm{T}}$ 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 interval 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 interval 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.

A convex polyhedron $\mathcal{P} \in \mathbb{CP}_n$ is said to be a *box* if and only if either $\mathcal{P}$ is the set of solutions to a finite set of interval constraints or $n = 0$ and $\mathcal{P} = \varnothing$. Therefore any n-dimensional *box* $\mathcal{P}$ in $\mathbb{R}^n$ where $n > 0$ can be represented by a sequence of n intervals $\mathcal{B}$ in $\mathbb{R}$ and $\mathcal{P}$ is a closed polyhedron if every bound in the intervals in $\mathcal{B}$ is either closed and bounded or open and unbounded.

1.7.1 Widening and Extrapolation Operators on Boxes

The library provides a widening operator for boxes. Given two sequences of intervals defining two n-dimensional boxes, the *CC76-widening* applies, for each corresponding interval and bound, the interval constraint widening defined in [CC76]. For extra precision, this incorporates the widening with thresholds as defined in [BCCetal02] with $\{-2, -1, 0, 1, 2\}$ as the set of default threshold values.

1.8 Weakly-Relational Shapes

The PPL provides support for computations on numerical domains that, in selected contexts, can achieve a better precision/efficiency ratio with respect to the corresponding computations on a "fully relational" domain of convex polyhedra. This is achieved by restricting the syntactic form of the constraints that can be used to describe the domain elements.

1.8.1 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 $i, j \in \{0, \ldots, n-1\}$ 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 $\mathcal{P} = \varnothing$.

1.8.2 Octagonal 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 an *octagonal* if there exist two indices $i, j \in \{0, \ldots, n-1\}$ such that:

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

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

Note that, since any bounded difference is also an octagonal constraint, any BDS is also an OS. The name "octagonal" comes from the fact that, in a vector space of dimension 2, a bounded OS can have eight sides at most.

1.8.3 Weakly-Relational Shapes Interface

By construction, any BDS or OS is always topologically closed. Under the usual set inclusion ordering, the set of all BDSs (resp., OSs) 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 $\mathbb{CP}_n$; moreover, the lattice of BDSs is a meet sublattice of the lattice of OSs. The least upper bound of a finite set of BDSs (resp., OSs) is said to be their *bds-hull* (resp., *oct-hull*).

As far as the representation of the rational inhomogeneous term of each bounded difference or octagonal constraint 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 and OSs 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 domains of BDSs and OSs, even though they are typically characterized by a lower degree of precision. For instance, the *bds-difference* and *oct-difference* operators return (the smallest) over-approximations of the set-theoretical difference operator on the corresponding domains. In the case of (generalized) images and preimages of affine relations, suitable (possibly not-optimal) over-approximations are computed when the considered relations cannot be precisely modeled by only using bounded differences or octagonal constraints.

1.8.4 Widening and Extrapolation Operators on Weakly-Relational Shapes

For the domains of BDSs and OSs, the library provides a variant of the widening operator for convex polyhedra defined in [CH78]. The implementation follows the specification in [BHMZ05a,BHMZ05b], resulting in an operator which is well-defined on the corresponding domain (i.e., it does not depend on the internal representation of BDSs or OSs), 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 and OSs, is not provided with a convergence guarantee and is therefore an extrapolation operator.

1.9 Rational Grids

In this section we introduce rational grids as provided by the library. See also [BDHetal05] for a detailed description of this domain.

The library 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* $\exists \mu \in \mathbb{Z} \,.\, a - b = \mu f$.

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

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

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 $\mathcal{L} = \varnothing$.

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 $\varnothing$, 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 $\operatorname{int.hull}(S)$ (resp., $\operatorname{int.affine.hull}(S)$) 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 *grid 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 *grid 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 [BDHetal05]). The 3-tuple (L, Q, P) is said to be a *grid 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 grid 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* grid generator system $\mathcal{G} = (L, Q, P)$ for $\mathcal{L}$ is such that, if $\mathcal{G}' = (L', Q', P')$ is another grid generator system for $\mathcal{L}$, then $\#\,L \leq \#\,L'$ and $\#\,Q + \#\,P \leq \#\,Q' + \#\,P'$. Note that a minimized grid generator system for a grid has no more than a total of $n + 1$ grid 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 grid 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, grid 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 $0 \leq \dim(\mathcal{G}) \leq n$.

1.10 Operations on Rational Grids

In general, the operations on rational grids are the same as those for the other PPL domains and the definitions of these can be found in Section Operations on Convex Polyhedra. Below we just describe those operations that have features or behavior that is in some way special to the grid domain.

1.10.1 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 $\phi = \bigl(x'_k = \langle \boldsymbol{a}, \boldsymbol{x} \rangle + b\bigr) \colon \mathbb{R}^n \to \mathbb{R}^n$ that transforms any point $(v_0, \dots, v_{n-1})^{\mathrm{T}}$ in a grid $\mathcal{L}$ 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 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 grid 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 grid points $\bigl\{(1,0)^{\mathrm{T}}, (7,3)^{\mathrm{T}}, (10,0)^{\mathrm{T}}\bigr\}$ which is the grid generated by the grid 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.

1.10.2 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 $\phi = (\mathrm{lhs}', \mathrm{rhs}, f) \colon \mathbb{R}^n \to \mathbb{R}^n$, 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

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

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.

1.10.3 Frequency Operator

Let $\mathcal{L} \in \mathbb{G}_n$ be any non-empty grid and $\mathrm{expr} = \bigl(\langle \boldsymbol{a}, \boldsymbol{x} \rangle + b\bigr)$ be a linear expression. Then if, for some $c, f \in \mathbb{R}$, all the points in $\mathcal{L}$ satisfy the congruence $\mathrm{cg} = (\mathrm{expr} \equiv_f c)$, then the maximum f such that this holds is called the *frequency* of $\mathcal{L}$ with respect to expr.

The frequency operator provided by the library returns both the frequency f and a value $\mathrm{val} = \langle \boldsymbol{a}, \boldsymbol{w} \rangle + b$ where $\boldsymbol{w} \in \mathcal{L}$ and

$$|\mathrm{val}| = \min \Bigl\{ \, \bigl| \langle \boldsymbol{a}, \boldsymbol{v} \rangle + b \bigr| \Bigm| \boldsymbol{v} \in \mathcal{L} \, \Bigr\}.$$

Observe that the above definition is also applied to other simple objects in the library like polyhedra, octagonal shapes, bd-shapes and boxes and in such cases the definition of frequency can be simplified. For instance, the frequency for an object $\mathcal{P} \in \mathbb{P}_n$ is defined if and only if there is a unique value c such that $\mathcal{P}$ saturates the equality $(\mathrm{expr} = c)$; in this case the frequency is 0 and the value returned is c.

1.10.4 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

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

1.10.5 Relation-with Operators

The library provides operators for checking the relation holding between a grid and a congruence, a grid generator, a constraint or a (polyhedron) 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 $\mathcal{L} \cap \mathcal{L}_{\mathrm{cg}} = \varnothing$; that is, adding cg to $\mathcal{C}$ gives us the empty grid.
- $\mathcal{L}$ *strictly intersects* cg if $\mathcal{L} \cap \mathcal{L}_{\mathrm{cg}} \neq \varnothing$ 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.
- $\mathcal{L}$ *saturates* cg if $\mathcal{L}$ *is included* in cg and $f = 0$, i.e., cg is an equality congruence.

For the relation between $\mathcal{L}$ and a constraint, suppose 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 $\mathcal{L}$ and c are as follows.

- $\mathcal{L}$ *is disjoint* from c if $\mathcal{L} \cap \mathcal{Q} = \varnothing$.
- $\mathcal{L}$ *strictly intersects* c if $\mathcal{L} \cap \mathcal{Q} \neq \varnothing$ and $\mathcal{L} \cap \mathcal{Q} \subset \mathcal{L}$.
- $\mathcal{L}$ *is included* in c if $\mathcal{L} \subseteq \mathcal{Q}$.
- $\mathcal{L}$ *saturates* c if $\mathcal{L}$ *is included* in c and $\bowtie$ is $=$.

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

A grid $\mathcal{L}$ *subsumes* a (polyhedron) point or closure point g if adding the corresponding grid point to any grid generator system representing $\mathcal{L}$ does not change $\mathcal{L}$. A grid $\mathcal{L}$ *subsumes* a (polyhedron) ray or line g if adding the corresponding grid line to any grid generator system representing $\mathcal{L}$ does not change $\mathcal{L}$.

1.10.6 Wrapping Operator

The operator wrap_assign provided by the library, allows for the wrapping of a subset of the set of space dimensions so as to fit the given bounded integer type and have the specified overflow behavior. In order to maximize the precision of this operator for grids, the exact behavior differs in some respects from the other simple classes of geometric descriptors.

Suppose $\mathcal{L} \in \mathbb{G}_n$ is a grid and J a subset of the set of space dimensions $\{0, \dots, n-1\}$. Suppose also that the width of the bounded integer type is w so that the range of values $R = \{r \in \mathbb{R} \mid 0 \leq r < 2^w\}$ if the type is unsigned and $R = \{r \in \mathbb{R} \mid -2^{w-1} \leq r < 2^{w-1}\}$ otherwise. Consider a space dimension $j \in J$ and a variable v_j for dimension j.

If the value in $\mathcal{L}$ for the variable v_j is a constant in the range R, then it is unchanged. Otherwise the result $\mathcal{L}'$ of the operation on $\mathcal{L}$ will depend on the specified overflow behavior.

- Overflow impossible. In this case, it is known that no wrapping can occur. If the grid $\mathcal{L}$ has no value for the variable v_j in the range R, then $\mathcal{L}$ is set empty. If v_j has exactly one value $a \in R$ in $\mathcal{L}$, then v_j is set equal to a. Otherwise, $\mathcal{L}' = \mathcal{L}$.

- Overflow undefined. In this case, for each value a for v_j in the grid $\mathcal{L}$, the wrapped value can be any value $a + z \in R$ where $z \in \mathbb{Z}$. Therefore $\mathcal{L}'$ is obtained by adding the parameter $(0, \dots, 0, v_j, 0, \dots, 0)$, where $v_j = 1$, to the generator system for $\mathcal{L}$.

- Overflow wraps. In this case, if $\mathcal{L}$ already satisfies the congruence $v_j = a \mod 2^w$, for some $a \in \mathbb{R}$, then v_j is set equal to a' where $a' = a \mod 2^w$ and $a' \in R$. Otherwise, $\mathcal{L}'$ is obtained by adding the parameter $(0, \dots, 0, v_j, 0, \dots, 0)$, where $v_j = 2^w$, to the generator system for $\mathcal{L}$.

1.10.7 Widening Operators

The library provides *grid widening* operators for the domain of grids. The congruence widening and generator widening follow the specifications provided in [BDHetal05]. The third widening uses either the congruence or the generator widening, the exact rule governing this choice at the time of the call is left to the implementation. Note that, as for the widenings provided for convex polyhedra, all the operations provided by the library for computing a widening $\mathcal{L}_1 \nabla \mathcal{L}_2$ of grids $\mathcal{L}_1, \mathcal{L}_2 \in \mathbb{G}_n$ require 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, in any widening operation that computes the widening $\mathcal{L}_1 \nabla \mathcal{L}_2$, the resulting grid will be assigned to overwrite the store containing the bigger grid $\mathcal{L}_2$. The smaller grid $\mathcal{L}_1$ 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 language interfaces.

1.10.8 Widening with Tokens

This is as for widening with tokens for convex polyhedra.

1.10.9 Extrapolation Operators

Besides the 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.

1.11 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 pointset powerset domain instantiation of this construction described in Section The Pointset Powerset Domain).

1.11.1 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 '$\mathbf{1}$' and bottom element '$\mathbf{0}$'.

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

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

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

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

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

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.

1.12 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.

1.12.1 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.

1.12.2 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 $\Omega_D^{\vdash}\bigl(\mathcal{S} \cup \{d\}\bigr)$.

1.12.3 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.

1.13 The Pointset Powerset Domain

The pointset powerset domain provided by the PPL is the finite powerset domain (defined in Section The Powerset Construction) whose base-level domain D is one of the classes of semantic geometric descriptors listed in Section Semantic Geometric Descriptors.

In addition to the operations described for the generic powerset domain in Section Operations on the Powerset Construction, the PPL provides all the generic operations listed in Generic Operations on Semantic Geometric Descriptors. Here we just describe those operations that are particular to the pointset powerset domain.

1.13.1 Meet-Preserving Simplification

Let $\mathcal{S}_1 = \{d_1, \ldots, d_m\}$, $\mathcal{S}_2 = \{c_1, \ldots, c_n\}$ and $\mathcal{S} = \{s_1, \ldots, s_q\}$ be Omega-reduced elements of a pointset powerset domain over the same base-level domain. Then:

- $\mathcal{S}$ is *powerset meet-preserving* with respect to $\mathcal{S}_1$ using context $\mathcal{S}_2$ if the meet of $\mathcal{S}$ and $\mathcal{S}_2$ is equal to the meet of $\mathcal{S}_1$ and $\mathcal{S}_2$;
- $\mathcal{S}$ is a *powerset simplification* with respect to $\mathcal{S}_1$ if $q \leq m$.
- $\mathcal{S}$ is a *disjunct meet-preserving simplification* with respect to $\mathcal{S}_1$ if, for each $s_k \in \mathcal{S}$, there exists $d_i \in \mathcal{S}_1$ such that, for each $c_j \in \mathcal{S}_2$, s_k is a meet-preserving enlargement and simplification of d_i using context c_j.

The library provides a binary operator (`simplify_using_context`) for the pointset powerset domain that returns a powerset which is a powerset meet-preserving, powerset simplification and disjunct meet-preserving simplification of its first argument using the second argument as context.

Notice that, due to the powerset simplification property, in general a meet-preserving powerset simplification is *not* an enlargement with respect to the ordering defined on the powerset lattice. Because of this, the operator provided by the library is only well-defined when the base-level domain is not itself a powerset domain.

1.13.2 Geometric Comparisons

Given the pointset powersets $\mathcal{S}_1, \mathcal{S}_2$ over the same base-level domain and with the same space dimension, then we say that $\mathcal{S}_1$ *geometrically covers* $\mathcal{S}_2$ if every point (in some disjunct) of $\mathcal{S}_2$ is also a point in a disjunct of $\mathcal{S}_1$. 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*.

1.13.3 Pairwise Merge

Given the pointset powerset $\mathcal{S}$ over a base-level semantic GD domain D, then the *pairwise merge* operator takes pairs of distinct elements in $\mathcal{S}$ whose upper bound (denoted here by $\uplus$) in D (using the PPL operator `upper_bound_assign()` for D) is the same as their set-theoretical union and replaces them by their union. This replacement is done recursively so that, for each pair c, d of distinct disjuncts in the result set, we have $c \uplus d \neq c \cup d$.

1.13.4 Powerset 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 any PPL extrapolation operator for the base-level domain. Note that, even when the extrapolation operator for the base-level domain D is known to be a widening on D, the `BGP99_extrapolation_assign` operator cannot guarantee the convergence of the iteration sequence in a finite number of steps (for a counter-example, see [BHZ04]).

1.13.5 Certificate-Based Widenings

The PPL library provides support for the specification of proper widening operators on the pointset powerset domain. 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 D, together with the corresponding convergence certificate, the BHZ03 framework is able to lift this widening on D to a widening on the pointset powerset domain; 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 upper bound operator for the base-level domain 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 upper bound of the two arguments, possibly improving the result obtained by means of the difference operator for the base-level domain. For more details and a justification of the overall approach, see [BHZ03b] and [BHZ04].

The library provides several convergence certificates. Note that, for the domain of Polyhedra, while Parma_Polyhedra_Library::BHRZ03_Certificate the "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.

1.14 Using the Library

1.14.1 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 might still make 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.

Warning

For the reasons mentioned above, starting from version 0.10 of the library, the usage of the eager versions (i.e., the ones having a name ending with `_and_minimize`) of these operators is *deprecated*; this is in preparation of their complete removal, which will occur starting from version 0.11.

1.14.2 On Pointset_Powerset and Partially_Reduced_Product Domains: A Warning

For future versions of the PPL library all practical instantiations for the disjuncts for a pointset_powerset and component domains for the partially_reduced_product domains will be fully supported. However, for version 0.10, these compound domains should not themselves occur as one of their argument domains. Therefore their use comes with the following warning.

Warning

The `Pointset_Powerset<PSET>` and `Partially_Reduced_Product<D1, D2, R>` should only be used with the following instantiations for the disjunct domain template `PSET` and component domain templates `D1` and `D2`: `C_Polyhedron`, `NNC_Polyhedron`, `Grid`, `Octagonal_Shape<T>`, `BD_Shape<T>`, `Box<T>`.

1.14.3 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.

1.14.4 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.

1.15 Bibliography

[Anc91] C. Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales*. PhD thesis, Université de Paris VI, Paris, France, March 1991.

[BA05] J. M. Bjorndalen and O. Anshus. Lessons learned in benchmarking - Floating point benchmarks: Can you trust them? In *Proceedings of the Norsk informatikkonferanse 2005 (NIK 2005)*, pages 89-100, Bergen, Norway, 2005. Tapir Akademisk Forlag.

[Bag97] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, March 1997. Printed as Report TD-1/97.

[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.

[BCC+ 02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. Æ. Mogensen, D. A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 85-108. Springer-Verlag, Berlin, 2002.

[BDH+ 05] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. A linear domain for analyzing the distribution of numerical values. Report 2005.06, School of Computing, University of Leeds, UK, 2005. Available at http://www.comp.leeds.ac.uk/research/pubs/reports.shtml.

[BDH+ 06] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. A practical tool for analyzing the distribution of numerical values, 2006. Available at http://www.comp.leeds.ac.uk/hill/Papers/papers.html.

[BDH+ 07] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. Grids: A domain for analyzing the distribution of numerical values. In G. Puebla, editor, *Logic-based Program Synthesis and Transformation, 16th International Symposium*, volume 4407 of *Lecture Notes in Computer Science*, pages 219-235, Venice, Italy, 2007. Springer-Verlag, Berlin.

[BFT00] A. Bemporad, K. Fukuda, and F. D. Torrisi. Convexity recognition of the union of polyhedra. Report AUT00-13, Automatic Control Laboratory, ETHZ, Zurich, Switzerland, 2000.

[BFT01] A. Bemporad, K. Fukuda, and F. D. Torrisi. Convexity recognition of the union of polyhedra. *Computational Geometry: Theory and Applications*, 18(3):141-154, 2001.

[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.

[BHMZ04] R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. Report arXiv:cs.PL/0412043, 2004. Extended abstract. Contribution to the *International workshop on "Numerical & Symbolic Abstract Domains"* (NSAD'05, Paris, January 21, 2005). Available at http://arxiv.org/ and http://www.cs.unipr.it/ppl/.

[BHMZ05a] R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. Quaderno 399, Dipartimento di Matematica, Università di Parma, Italy, 2005. Available at http://www.cs.unipr.it/Publications/.

[BHMZ05b] R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. In C. Hankin and I. Siveroni, 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/.

[BHRZ05] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2):28-56, 2005.

[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, *Verification, Model Checking and Abstract Interpretation: Proceedings of the 5th International Conference (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/`.

[BHZ05] R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17(2):222-257, 2005.

[BHZ06a] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[BHZ06b] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *Software Tools for Technology Transfer*, 8(4/5):449-466, 2006. In the printed version of this article, all the figures have been improperly printed (rendering them useless). See [BHZ07c].

[BHZ07a] R. Bagnara, P. M. Hill, and E. Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. Quaderno 458, Dipartimento di Matematica, Università di Parma, Italy, 2007. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.CG/0701122`, available from `http://arxiv.org/`.

[BHZ07b] R. Bagnara, P. M. Hill, and E. Zaffanella. An improved tight closure algorithm for integer octagonal constraints. Quaderno 467, Dipartimento di Matematica, Università di Parma, Italy, 2007. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:0705.4618v2 [cs.DS]`, available from `http://arxiv.org/`.

[BHZ07c] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *Software Tools for Technology Transfer*, 9(3/4):413-414, 2007. Erratum to [BHZ06b] containing all the figures properly printed.

[BHZ08a] R. Bagnara, P. M. Hill, and E. Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In F. Logozzo, D. Peled, and L. Zuck, editors, *Verification, Model Checking and Abstract Interpretation: Proceedings of the 9th International Conference (VMCAI 2008)*, volume 4905 of *Lecture Notes in Computer Science*, pages 8-21, San Francisco, USA, 2008. Springer-Verlag, Berlin.

[BHZ08b] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3-21, 2008.

[BHZ09a] R. Bagnara, P. M. Hill, and E. Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. *Theoretical Computer Science*, 410(46):4672-4691, 2009.

[BHZ09b] R. Bagnara, P. M. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. Quaderno 492, Dipartimento di Matematica, Università di Parma, Italy, 2009. Available at `http://www.cs.unipr.it/Publications/`. A corrected and improved version (corrected an error in the statement of condition (3) of Theorem 3.6, typos corrected in statement and proof of Theorem 6.8) has been published in [BHZ09c].

[BHZ09c] R. Bagnara, P. M. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. Report `arXiv:cs.CG/0904.1783`, 2009. Available at `http://arxiv.org/` and `http://www.cs.unipr.it/ppl/`.

[BHZ09d] R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279-323, 2009.

[BHZ10] R. Bagnara, P. M. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *Computational Geometry: Theory and Applications*, 43(5):453-473, 2010. To appear in print. Available online at `http://dx.doi.org/10.1016/j.comgeo.2009.09.002`.

[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.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In B. Knobe, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, volume 24(7) of *ACM SIGPLAN Notices*, pages 41-53, Portland, Oregon, USA, 1989. ACM Press.

[BMPZ10] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. The automatic synthesis of linear ranking functions: The complete unabridged version. Quaderno 498, Dipartimento di Matematica, Università di Parma, Italy, 2010. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.PL/1004.0944`, available from `http://arxiv.org/`.

[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 Second International Symposium on Programming*, pages 106-130, Paris, France, 1976. Dunod, Paris, France.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269-282, New York, 1979. ACM Press.

[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.

[FCB07] P. Feautrier, J.-F. Collard, and C. Bastoul. *PIP/PipLib: A Solver for Parametric Integer Programming Problems*, 5.0 edition, July 2007. Distributed with PIP/PipLib 1.4.0.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243-268, 1988.

[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/polyfaq/polyfaq.html`, 1998.

[GDD+ 04] D. Gopan, F. DiMaio, N. Dor, T. W. 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, Barcelona, Spain, 2004. Springer-Verlag, Berlin.

[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.

[GR77] D. Goldfarb and J. K. Reid. A practical steepest-edge simplex algorithm. *Mathematical Proramming*, 12(1):361-371, 1977.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169-192, Brighton, UK, 1991. Springer-Verlag, Berlin.

[Gra97] P. Granger. Static analyses of congruence properties on rational numbers (extended abstract). In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the 4th International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 278-292, Paris, France, 1997. Springer-Verlag, Berlin.

[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 (CAV'93)*, 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.

[HHL90] L. Huelsbergen, D. Hahn, and J. Larus. Exact dependence analysis using data access descriptors. Technical Report 945, Department of Computer Science, University of Wisconsin, Madison, 1990.

[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.

[HMT71] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras: Part I*. North-Holland, Amsterdam, 1971.

[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.

[JMSY94] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond finite domains. In A. Borning, editor, *Principles and Practice of Constraint Programming: Proceedings of the Second International Workshop*, volume 874 of *Lecture Notes in Computer Science*, pages 86-94, Rosario, Orcas Island, Washington, USA, 1994. Springer-Verlag, Berlin.

[KBB+ 06] L. Khachiyan, E. Boros, K. Borys, K. Elbassioni, and V. Gurvich. Generating all vertices of a polyhedron is hard. *Discrete and Computational Geometry*, 2006. Invited contribution. To appear.

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

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

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

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

[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.

[Min01a] A. Miné. A new numerical abstract domain based on difference-bound matrices. In O. Danvy and A. Filinski, editors, *Proceedings of the 2nd Symposium on Programs as Data Objects (PADO 2001)*, volume 2053 of *Lecture Notes in Computer Science*, pages 155-172, Aarhus, Denmark, 2001. Springer-Verlag, Berlin.

[Min01b] A. Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 310-319, Stuttgart, Germany, 2001. IEEE Computer Society Press.

[Min02] A. Miné. A few graph-based relational numerical abstract domains. 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 117-132, Madrid, Spain, 2002. Springer-Verlag, Berlin.

[Min04] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Programming Languages and Systems: Proceedings of the 13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 3-17, Barcelona, Spain, 2004. Springer-Verlag, Berlin.

[Min05] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Paris, France, March 2005.

[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.

[NF01] T. Nakanishi and A. Fukuda. Modulo interval arithmetic and its application to program analysis. *Transactions of Information Processing Society of Japan*, 42(4):829-837, 2001.

[NJPF99] T. Nakanishi, K. Joe, C. D. Polychronopoulos, and A. Fukuda. The modulo interval: A simple and practical representation for program analysis. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 91-96, Newport Beach, California, USA, 1999. IEEE Computer Society.

[NO77] G. Nelson and D. C. Oppen. Fast decision algorithms based on Union and Find. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 114-119, Providence, RI, USA, 1977. IEEE Computer Society Press. The journal version of this paper is [NO80].

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356-364, 1980. An earlier version of this paper is [NO77].

[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.

[Pra77] V. R. Pratt. Two easy theories whose combination is hard. Memo sent to Nelson and Oppen concerning a preprint of their paper [NO77], September 1977.

[PS98] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, second edition, 1998.

[QRR96] P. Quinton, S. Rajopadhye, and T. Risset. On manipulating Z-polyhedra. Technical Report 1016, IRISA, Campus Universitaire de Bealieu, Rennes, France, July 1996.

[QRR97] P. Quinton, S. Rajopadhye, and T. Risset. On manipulating Z-polyhedra using a canonic representation. *Parallel Processing Letters*, 7(2):181-194, 1997.

[QRW00] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469-498, 2000.

[RBL06] T. W. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In J. Hatcliff and F. Tip, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 100-111, Charleston, South Carolina, USA, 2006. ACM Press.

[Ric02] E. Ricci. Rappresentazione e manipolazione di poliedri convessi per l'analisi e la verifica di programmi. Laurea dissertation, University of Parma, Parma, Italy, July 2002. In Italian.

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

[Sho81] R. E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769-779, 1981.

[SK07] A. Simon and A. King. Taming the wrapping of integer arithmetic. In H. Riis Nielson and G. Filé, editors, *Static Analysis: Proceedings of the 14th International Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 121-136, Kongens Lyngby, Denmark, 2007. Springer-Verlag, Berlin.

[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.

[SS07a] R. Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007)*, pages 39-48, Nice, France, 2007. IEEE Computer Society Press.

[SS07b] R. Sen and Y. N. Srikant. Executable analysis with circular linear progressions. Technical Report IISc-CSA-TR-2007-3, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India, 2007.

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

[War03] H. S. Warren, Jr. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[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 GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. 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 them 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 prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. 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.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

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

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey 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; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may

be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If 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 convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

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

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

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

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

16. Limitation of Liability.

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

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

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

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

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

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
```

```
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

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

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

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

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

```
 program  Copyright (C)  year   name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

3 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some 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 the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled
"GNU Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

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.

4 Module Index

4.1 Modules

Here is a list of all modules:

5 Namespace Index

5.1 Namespace List

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

6 Class Index

6.1 Class Hierarchy

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

7 Class Index

7.1 Class List

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

8 Module Documentation

8.1 C++ Language Interface

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

Classes

- class Parma_Polyhedra_Library::Throwable

  User objects the PPL can throw.

- struct Parma_Polyhedra_Library::Recycle_Input

  A tag class.

- struct Parma_Polyhedra_Library::Is_Checked< T >
- struct Parma_Polyhedra_Library::Is_Checked< Checked_Number< T, P > >
- struct Parma_Polyhedra_Library::Is_Native_Or_Checked< T >
- class Parma_Polyhedra_Library::Checked_Number< T, Policy >

  A wrapper for numeric types implementing a given policy.

- 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::Linear_Expression

  A linear expression.

- 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::Constraint

  A linear equality or inequality.

- 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::Generator

  A line, ray, point or closure point.

- 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::Congruence

  A linear congruence.

- 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::Grid_Generator

  A grid line, parameter or grid point.

- class Parma_Polyhedra_Library::PIP_Problem

  A Parametric Integer (linear) Programming problem.

- 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::Poly_Gen_Relation

  The relation between a polyhedron and a generator.

- class Parma_Polyhedra_Library::Polyhedron

  The base class for convex polyhedra.

- class Parma_Polyhedra_Library::MIP_Problem

  A Mixed Integer (linear) Programming problem.

- class Parma_Polyhedra_Library::Interval< Boundary, Info >

  A generic, not necessarily closed, possibly restricted interval.

- class Parma_Polyhedra_Library::Grid_Certificate

The convergence certificate for the Grid widening operator.

- 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::Grid

  A grid.

- class Parma_Polyhedra_Library::Box< ITV >

  A not necessarily closed, iso-oriented hyperrectangle.

- class Parma_Polyhedra_Library::BD_Shape< T >

  A bounded difference shape.

- class Parma_Polyhedra_Library::Octagonal_Shape< T >

  An octagonal shape.

- class Parma_Polyhedra_Library::Smash_Reduction< D1, D2 >

  This class provides the reduction method for the Smash_Product domain.

- class Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >

  This class provides the reduction method for the Constraints_Product domain.

- class Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 >

  This class provides the reduction method for the Congruences_Product domain.

- class Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 >

  This class provides the reduction method for the Shape_Preserving_Product domain.

- class Parma_Polyhedra_Library::No_Reduction< D1, D2 >

  This class provides the reduction method for the Direct_Product domain.

- class Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >

  The partially reduced product of two abstractions.

- class Parma_Polyhedra_Library::Determinate< PSET >

  A wrapper for PPL pointsets, providing them with a determinate constraint system interface, as defined in [Bag98].

- class Parma_Polyhedra_Library::Powerset< D >

  The powerset construction on a base-level domain.

- class Parma_Polyhedra_Library::Pointset_Powerset< PSET >

  The powerset construction instantiated on PPL pointset domains.

- class Parma_Polyhedra_Library::GMP_Integer

  Unbounded integers as provided by the GMP library.

Namespaces

- namespace Parma_Polyhedra_Library::IO_Operators

  All input/output operators are confined to this namespace.

- namespace std

  The standard C++ namespace.

Defines

- #define PPL_VERSION_MAJOR 0

  The major number of the PPL version.

- #define PPL_VERSION_MINOR 11

  The minor number of the PPL version.

- #define PPL_VERSION_REVISION 2

  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 "0.11.2"

  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 PPL_COEFFICIENT_TYPE Parma_Polyhedra_Library::Coefficient

  An alias for easily naming the type of PPL coefficients.

Enumerations

- enum Parma_Polyhedra_Library::Result {

  Parma_Polyhedra_Library::V_EMPTY, Parma_Polyhedra_Library::V_EQ, Parma_Polyhedra_Library::V_LT, Parma_Polyhedra_Library::V_GT,

  Parma_Polyhedra_Library::V_NE, Parma_Polyhedra_Library::V_LE, Parma_Polyhedra_Library::V_GE, Parma_Polyhedra_Library::V_LGE,

  Parma_Polyhedra_Library::V_OVERFLOW, Parma_Polyhedra_Library::V_LT_INF, Parma_Polyhedra_Library::V_GT_SUP, Parma_Polyhedra_Library::V_LT_PLUS_INFINITY, Parma_Polyhedra_Library::V_GT_MINUS_INFINITY, Parma_Polyhedra_Library::V_EQ_-MINUS_INFINITY, Parma_Polyhedra_Library::V_EQ_PLUS_INFINITY, Parma_Polyhedra_-Library::V_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_UNREPRESENTABLE }

Possible outcomes of a checked arithmetic computation.

- 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::EQUAL, Parma_Polyhedra_Library::LESS_THAN, Parma_-Polyhedra_Library::LESS_OR_EQUAL, Parma_Polyhedra_Library::GREATER_THAN,

  Parma_Polyhedra_Library::GREATER_OR_EQUAL, Parma_Polyhedra_Library::NOT_EQUAL }

  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::Bounded_Integer_Type_Width {

  Parma_Polyhedra_Library::BITS_8, Parma_Polyhedra_Library::BITS_16, Parma_Polyhedra_-Library::BITS_32, Parma_Polyhedra_Library::BITS_64,

  Parma_Polyhedra_Library::BITS_128 }

  Widths of bounded integer types.

- enum Parma_Polyhedra_Library::Bounded_Integer_Type_Representation { Parma_Polyhedra_-Library::UNSIGNED, Parma_Polyhedra_Library::SIGNED_2_COMPLEMENT }

  Representation of bounded integer types.

- enum Parma_Polyhedra_Library::Bounded_Integer_Type_Overflow { Parma_Polyhedra_-Library::OVERFLOW_WRAPS, Parma_Polyhedra_Library::OVERFLOW_UNDEFINED, Parma_Polyhedra_Library::OVERFLOW_IMPOSSIBLE }

  Overflow behavior of bounded integer types.

- 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 ,

  Parma_Polyhedra_Library::ROUND_STRICT_RELATION }

  Rounding directions for arithmetic computations.

- enum Parma_Polyhedra_Library::PIP_Problem_Status { Parma_Polyhedra_-Library::UNFEASIBLE_PIP_PROBLEM, Parma_Polyhedra_Library::OPTIMIZED_PIP_-PROBLEM }

  Possible outcomes of the PIP_Problem solver.

- enum Parma_Polyhedra_Library::MIP_Problem_Status { Parma_Polyhedra_-Library::UNFEASIBLE_MIP_PROBLEM, Parma_Polyhedra_Library::UNBOUNDED_MIP_-PROBLEM, Parma_Polyhedra_Library::OPTIMIZED_MIP_PROBLEM }

  Possible outcomes of the MIP_Problem solver.

Variables

- const Throwable ∗volatile Parma_Polyhedra_Library::abandon_expensive_computations

  A pointer to an exception object.

8.1.1 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.

8.1.2 Define Documentation

8.1.2.1 #define PPL_VERSION_MAJOR 0

The major number of the PPL version.

8.1.2.2 #define PPL_VERSION_MINOR 11

The minor number of the PPL version.

8.1.2.3 #define PPL_VERSION_REVISION 2

The revision number of the PPL version.

8.1.2.4 #define PPL_VERSION "0.11.2"

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.

8.1.3 Typedef Documentation

8.1.3.1 typedef size_t Parma_Polyhedra_Library::dimension_type

An unsigned integral type for representing space dimensions.

8.1.3.2 typedef size_t Parma_Polyhedra_Library::memory_size_type

An unsigned integral type for representing memory size in bytes.

8.1.3.3 typedef PPL_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 the policy Bounded_Integer_Coefficient_-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).

8.1.4 Enumeration Type Documentation

8.1.4.1 enum Parma_Polyhedra_Library::Result

Possible outcomes of a checked arithmetic computation.

Enumerator:

V_EMPTY The exact result is not comparable.

V_EQ The computed result is exact.

V_LT The computed result is inexact and rounded up.

V_GT The computed result is inexact and rounded down.

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.

V_OVERFLOW The exact result is a number out of finite bounds.

V_LT_INF A negative integer overflow occurred (rounding up).

V_GT_SUP A positive integer overflow occurred (rounding down).

V_LT_PLUS_INFINITY A positive integer overflow occurred (rounding up).

V_GT_MINUS_INFINITY A negative integer overflow occurred (rounding down).

V_EQ_MINUS_INFINITY Negative infinity result.

V_EQ_PLUS_INFINITY Positive infinity result.

V_NAN Not a number result.

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_UNREPRESENTABLE The computed result is not representable.

8.1.4.2 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.

8.1.4.3 enum Parma_Polyhedra_Library::Relation_Symbol

Relation symbols.

Enumerator:

EQUAL Equal to.

LESS_THAN Less than.

LESS_OR_EQUAL Less than or equal to.

GREATER_THAN Greater than.

GREATER_OR_EQUAL Greater than or equal to.

NOT_EQUAL Not equal to.

8.1.4.4 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.

8.1.4.5 enum Parma_Polyhedra_Library::Optimization_Mode

Possible optimization modes.

Enumerator:

MINIMIZATION Minimization is requested.

MAXIMIZATION Maximization is requested.

8.1.4.6 enum Parma_Polyhedra_Library::Bounded_Integer_Type_Width

Widths of bounded integer types.

See the section on approximating bounded integers.

Enumerator:

BITS_8 8 bits.

BITS_16 16 bits.

BITS_32 32 bits.

BITS_64 64 bits.

BITS_128 128 bits.

8.1.4.7 enum Parma_Polyhedra_Library::Bounded_Integer_Type_Representation

Representation of bounded integer types.

See the section on approximating bounded integers.

Enumerator:

UNSIGNED Unsigned binary.

SIGNED_2_COMPLEMENT Signed binary where negative values are represented by the two's complement of the absolute value.

8.1.4.8 enum Parma_Polyhedra_Library::Bounded_Integer_Type_Overflow

Overflow behavior of bounded integer types.

See the section on approximating bounded integers.

Enumerator:

OVERFLOW_WRAPS On overflow, wrapping takes place. This means that, for a w-bit bounded integer, the computation happens modulo 2^w.

OVERFLOW_UNDEFINED On overflow, the result is undefined. This simply means that the result of the operation resulting in an overflow can take any value.

Note

Even though something more serious can happen in the system being analyzed ---due to, e.g., C's undefined behavior---, here we are only concerned with the results of arithmetic operations. It is the responsibility of the analyzer to ensure that other manifestations of undefined behavior are conservatively approximated.

OVERFLOW_IMPOSSIBLE Overflow is impossible. This is for the analysis of languages where overflow is trapped before it affects the state, for which, thus, any indication that an overflow may have affected the state is necessarily due to the imprecision of the analysis.

8.1.4.9 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 that the operation result is exact and representable in the destination type. Result info is evaluated lazily.

ROUND_STRICT_RELATION The client code is willing to pay an extra price to know the exact relation beetwen the exact result and the computed one.

8.1.4.10 enum Parma_Polyhedra_Library::PIP_Problem_Status

Possible outcomes of the PIP_Problem solver.

Enumerator:

UNFEASIBLE_PIP_PROBLEM The problem is unfeasible.

OPTIMIZED_PIP_PROBLEM The problem has an optimal solution.

8.1.4.11 enum Parma_Polyhedra_Library::MIP_Problem_Status

Possible outcomes of the MIP_Problem solver.

Enumerator:

UNFEASIBLE_MIP_PROBLEM The problem is unfeasible.

UNBOUNDED_MIP_PROBLEM The problem is unbounded.

OPTIMIZED_MIP_PROBLEM The problem has an optimal solution.

8.1.5 Variable Documentation

8.1.5.1 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.

9 Namespace Documentation

9.1 Parma_Polyhedra_Library Namespace Reference

The entire library is confined to this namespace.

Namespaces

- namespace IO_Operators

  All input/output operators are confined to this namespace.

Classes

- class Throwable

  User objects the PPL can throw.

- struct Recycle_Input

  A tag class.

- struct Is_Checked
- struct Is_Checked< Checked_Number< T, P > >
- struct Is_Native_Or_Checked
- class Checked_Number

  A wrapper for numeric types implementing a given policy.

- class Variable

  A dimension of the vector space.

- class Linear_Expression

  A linear expression.

- class Constraint_System

  A system of constraints.

- class Constraint

  A linear equality or inequality.

- class Poly_Con_Relation

  The relation between a polyhedron and a constraint.

- class Generator_System

  A system of generators.

- class Generator

  A line, ray, point or closure point.

- class Congruence_System

  A system of congruences.

- class Congruence

  A linear congruence.

- class Grid_Generator_System

  A system of grid generators.

- class Grid_Generator

  A grid line, parameter or grid point.

- class Variables_Set

  An std::set of variables' indexes.

- class PIP_Problem

  A Parametric Integer (linear) Programming problem.

- class PIP_Tree_Node

  A node of the PIP solution tree.

- class PIP_Solution_Node

  A tree node representing part of the space of solutions.

- class PIP_Decision_Node

  A tree node representing a decision in the space of solutions.

- class BHRZ03_Certificate

  The convergence certificate for the BHRZ03 widening operator.

- class H79_Certificate

  A convergence certificate for the H79 widening operator.

- class Poly_Gen_Relation

  The relation between a polyhedron and a generator.

- class Polyhedron

  The base class for convex polyhedra.

- class MIP_Problem

  A Mixed Integer (linear) Programming problem.

- class Interval

  A generic, not necessarily closed, possibly restricted interval.

- class Grid_Certificate

  The convergence certificate for the Grid widening operator.

- class C_Polyhedron

  A closed convex polyhedron.

- class NNC_Polyhedron

  A not necessarily closed convex polyhedron.

- class Grid

  A grid.

- class Box

  A not necessarily closed, iso-oriented hyperrectangle.

- class BD_Shape

  A bounded difference shape.

- class Octagonal_Shape

  An octagonal shape.

- class Smash_Reduction

  This class provides the reduction method for the Smash_Product domain.

- class Constraints_Reduction

  This class provides the reduction method for the Constraints_Product domain.

- class Congruences_Reduction

  This class provides the reduction method for the Congruences_Product domain.

- class Shape_Preserving_Reduction

  This class provides the reduction method for the Shape_Preserving_Product domain.

- class No_Reduction

  This class provides the reduction method for the Direct_Product domain.

- class Partially_Reduced_Product

  The partially reduced product of two abstractions.

- class Domain_Product

  This class is temporary and will be removed when template typedefs will be supported in C++.

- class Determinate

  A wrapper for PPL pointsets, providing them with a determinate constraint system interface, as defined in [Bag98].

- class Powerset

  The powerset construction on a base-level domain.

- class Pointset_Powerset

  The powerset construction instantiated on PPL pointset domains.

- class GMP_Integer

  Unbounded integers as provided by the GMP library.

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 PPL_COEFFICIENT_TYPE Coefficient

  An alias for easily naming the type of PPL coefficients.

Enumerations

- enum Result_Class { VC_NORMAL, VC_MINUS_INFINITY, VC_PLUS_INFINITY, VC_NAN }
- enum Result_Relation {
  VR_EMPTY, VR_EQ, VR_LT, VR_GT,
  VR_NE, VR_LE, VR_GE, VR_LGE }
- enum Result {
  V_EMPTY, V_EQ, V_LT, V_GT,
  V_NE, V_LE, V_GE, V_LGE,
  V_OVERFLOW, V_LT_INF, V_GT_SUP, V_LT_PLUS_INFINITY,
  V_GT_MINUS_INFINITY, V_EQ_MINUS_INFINITY, V_EQ_PLUS_INFINITY, V_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_UNREPRESENTABLE }

  Possible outcomes of a checked arithmetic computation.

- enum Degenerate_Element { UNIVERSE, EMPTY }

  Kinds of degenerate abstract elements.

- enum Relation_Symbol {
  EQUAL, LESS_THAN, LESS_OR_EQUAL, GREATER_THAN,
  GREATER_OR_EQUAL, NOT_EQUAL }

  Relation symbols.

- enum Complexity_Class { POLYNOMIAL_COMPLEXITY, SIMPLEX_COMPLEXITY, ANY_COMPLEXITY }

  Complexity pseudo-classes.

- enum Optimization_Mode { MINIMIZATION, MAXIMIZATION }

  Possible optimization modes.

- enum Bounded_Integer_Type_Width {
  BITS_8, BITS_16, BITS_32, BITS_64,
  BITS_128 }

  Widths of bounded integer types.

- enum Bounded_Integer_Type_Representation { UNSIGNED, SIGNED_2_COMPLEMENT }

  Representation of bounded integer types.

- enum Bounded_Integer_Type_Overflow { OVERFLOW_WRAPS, OVERFLOW_UNDEFINED, OVERFLOW_IMPOSSIBLE }

Overflow behavior of bounded integer types.

- enum Rounding_Dir {
  ROUND_DOWN, ROUND_UP, ROUND_IGNORE , ROUND_NOT_NEEDED ,
  ROUND_STRICT_RELATION }

  Rounding directions for arithmetic computations.

- enum PIP_Problem_Status { UNFEASIBLE_PIP_PROBLEM, OPTIMIZED_PIP_PROBLEM }

  Possible outcomes of the PIP_Problem solver.

- enum MIP_Problem_Status { UNFEASIBLE_MIP_PROBLEM, UNBOUNDED_MIP_PROBLEM, OPTIMIZED_MIP_PROBLEM }

  Possible outcomes of the MIP_Problem solver.

- enum I_Result {
  I_EMPTY = 1, I_SINGLETON = 2, I_SOME = 4, I_UNIVERSE = 8,
  I_NOT_EMPTY = I_SINGLETON | I_SOME | I_UNIVERSE, I_ANY = I_EMPTY | I_NOT_-EMPTY, I_NOT_UNIVERSE = I_EMPTY | I_SINGLETON | I_SOME, I_NOT_DEGENERATE = I_SINGLETON | I_SOME,
  I_EXACT = 16, I_INEXACT = 32, I_CHANGED = 64, I_UNCHANGED = 128,
  I_SINGULARITIES = 256 }

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.

- void set_rounding_for_PPL ()

  Sets the FPU rounding mode so that the PPL abstractions based on floating point numbers work correctly.

- void restore_pre_PPL_rounding ()

  Sets the FPU rounding mode as it was before initialization of the PPL.

- void fpu_initialize_control_functions ()

Initializes the FPU control functions.

- 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.

- Result_Class result_class (Result r)

  Extracts the value class part of `r` *(representable number, unrepresentable minus/plus infinity or nan).*

- Result_Relation result_relation (Result r)

  Extracts the relation part of `r`.

- dimension_type not_a_dimension ()

  Returns a value that does not designate a valid dimension.

- Rounding_Dir inverse (Rounding_Dir dir)

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

- void initialize ()

  Initializes the library.

- void finalize ()

  Finalizes the library.

- unsigned irrational_precision ()

  Returns the precision parameter used for irrational calculations.

- void set_irrational_precision (const unsigned p)

  Sets the precision parameter used for irrational calculations.

- Coefficient_traits::const_reference Coefficient_zero ()

Returns a const reference to a Coefficient with value 0.

- Coefficient_traits::const_reference Coefficient_one ()

  Returns a const reference to a Coefficient with value 1.

- unsigned long isqrt (unsigned long x)

  Returns the integer square root of x*.*

- dimension_type max_space_dimension ()

  Returns the maximum space dimension this library can handle.

- template<typename PSET >
  bool termination_test_MS (const PSET &pset)

  Termination test using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  bool termination_test_MS_2 (const PSET &pset_before, const PSET &pset_after)

  Termination test using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  bool one_affine_ranking_function_MS (const PSET &pset, Generator &mu)

  Termination test with witness ranking function using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  bool one_affine_ranking_function_MS_2 (const PSET &pset_before, const PSET &pset_after, Generator &mu)

  Termination test with witness ranking function using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  void all_affine_ranking_functions_MS (const PSET &pset, C_Polyhedron &mu_space)

  Termination test with ranking function space using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  void all_affine_ranking_functions_MS_2 (const PSET &pset_before, const PSET &pset_after, C_Polyhedron &mu_space)

  Termination test with ranking function space using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  void all_affine_quasi_ranking_functions_MS (const PSET &pset, C_Polyhedron &decreasing_mu_space, C_Polyhedron &bounded_mu_space)

  Computes the spaces of affine quasi ranking functions using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template<typename PSET >
  void all_affine_quasi_ranking_functions_MS_2 (const PSET &pset_before, const PSET &pset_after, C_Polyhedron &decreasing_mu_space, C_Polyhedron &bounded_mu_space)

  Computes the spaces of affine quasi ranking functions using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

- template< typename PSET >
  bool termination_test_PR (const PSET &pset)

  Like termination_test_MS() but using the method by Podelski and Rybalchenko [BMPZ10].

- template< typename PSET >
  bool termination_test_PR_2 (const PSET &pset_before, const PSET &pset_after)

  Like termination_test_MS_2() but using an alternative formalization of the method by Podelski and Rybalchenko [BMPZ10].

- template< typename PSET >
  bool one_affine_ranking_function_PR (const PSET &pset, Generator &mu)

  Like one_affine_ranking_function_MS() but using the method by Podelski and Rybalchenko [BMPZ10].

- template< typename PSET >
  bool one_affine_ranking_function_PR_2 (const PSET &pset_before, const PSET &pset_after, Generator &mu)

  Like one_affine_ranking_function_MS_2() but using an alternative formalization of the method by Podelski and Rybalchenko [BMPZ10].

- template< typename PSET >
  void all_affine_ranking_functions_PR (const PSET &pset, NNC_Polyhedron &mu_space)

  Like all_affine_ranking_functions_MS() but using the method by Podelski and Rybalchenko [BMPZ10].

- template< typename PSET >
  void all_affine_ranking_functions_PR_2 (const PSET &pset_before, const PSET &pset_after, NNC_Polyhedron &mu_space)

  Like all_affine_ranking_functions_MS_2() but using an alternative formalization of the method by Podelski and Rybalchenko [BMPZ10].

Memory Size Inspection Functions

- template< typename T >
  Enable_If< Is_Native< T >::value, memory_size_type >::type **total_memory_in_bytes** (const T &)
- template< typename T >
  Enable_If< Is_Native< T >::value, memory_size_type >::type **external_memory_in_bytes** (const T &)
- memory_size_type **total_memory_in_bytes** (const mpz_class &x)
- memory_size_type **external_memory_in_bytes** (const mpz_class &x)
- memory_size_type **total_memory_in_bytes** (const mpq_class &x)
- memory_size_type **external_memory_in_bytes** (const mpq_class &x)

Relational Operators and Comparison Functions

- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **equal** (const T1 &x, const T2 &y)
- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **not_equal** (const T1 &x, const T2 &y)

- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **greater_or_equal** (const T1 &x, const T2 &y)
- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **greater_than** (const T1 &x, const T2 &y)
- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **less_or_equal** (const T1 &x, const T2 &y)
- template< typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value, bool >::type **less_than** (const T1 &x, const T2 &y)

Input-Output Operators

- template< typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, void >::type ascii_dump (std::ostream &s, const T &t)

  Ascii dump for native or checked.
- template< typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, bool >::type ascii_load (std::ostream &s, T &t)

  Ascii load for native or checked.

Variables

- const Throwable *volatile abandon_expensive_computations

  A pointer to an exception object.

9.1.1 Detailed Description

The entire library is confined to this namespace.

9.1.2 Enumeration Type Documentation

9.1.2.1 enum Parma_Polyhedra_Library::Result_Class

Enumerator:

VC_NORMAL Representable number result class.

VC_MINUS_INFINITY Negative infinity result class.

VC_PLUS_INFINITY Positive infinity result class.

VC_NAN Not a number result class.

9.1.2.2 enum Parma_Polyhedra_Library::Result_Relation

Enumerator:

VR_EMPTY No values satisfies the relation.

VR_EQ Equal. This need to be accompanied by a value.

VR_LT Less than. This need to be accompanied by a value.

VR_GT Greater than. This need to be accompanied by a value.

VR_NE Not equal. This need to be accompanied by a value.

VR_LE Less or equal. This need to be accompanied by a value.

VR_GE Greater or equal. This need to be accompanied by a value.

VR_LGE All values satisfy the relation.

9.1.2.3 enum Parma_Polyhedra_Library::I_Result

Enumerator:

I_EMPTY The resulting set may be empty.

I_SINGLETON The resulting set may have only one value.

I_SOME The resulting set may have more than one value and to be not the domain universe.

I_UNIVERSE The resulting set may be the domain universe.

I_NOT_EMPTY The resulting set is not empty.

I_ANY The resulting set may be empty or not empty.

I_NOT_UNIVERSE The resulting set may be empty or not empty.

I_NOT_DEGENERATE The resulting set can'be empty or the domain universe.

I_EXACT The resulting set is definitely exact.

I_INEXACT The resulting set is definitely inexact.

I_CHANGED The operation has definitely changed the set.

I_UNCHANGED The operation has left the set definitely unchanged.

I_SINGULARITIES The operation is undefined for some combination of values.

9.1.3 Function Documentation

9.1.3.1 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.

9.1.3.2 void Parma_Polyhedra_Library::set_rounding_for_PPL () `[inline]`

Sets the FPU rounding mode so that the PPL abstractions based on floating point numbers work correctly.

This is performed automatically at initialization-time. Calling this function is needed only if restore_pre_-PPL_rounding() has been previously called.

9.1.3.3 void Parma_Polyhedra_Library::restore_pre_PPL_rounding () `[inline]`

Sets the FPU rounding mode as it was before initialization of the PPL.

After calling this function it is absolutely necessary to call set_rounding_for_PPL() before using any PPL abstractions based on floating point numbers. This is performed automatically at finalization-time.

9.1.3.4 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.

9.1.3.5 void Parma_Polyhedra_Library::set_irrational_precision (const unsigned *p*) `[inline]`

Sets the precision parameter used for irrational calculations.

The lesser between numerator and denominator is limited to `2**p`.

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

Exceptions

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

9.1.3.6 template<typename PSET > bool Parma_Polyhedra_Library::termination_test_MS (const PSET & *pset*) `[inline]`

Termination test using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:

- $x'_1, \dots, x'_n$ go onto space dimensions $0, \dots, n-1$,
- $x_1, \dots, x_n$ go onto space dimensions $n, \dots, 2n-1$,

where unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update.

Returns

`true` if any loop approximated by `pset` definitely terminates; `false` if the test is inconclusive. However, if `pset` *precisely* characterizes the effect of the loop body onto the loop-relevant program variables, then `true` is returned *if and only if* the loop terminates.

9.1.3.7 template<typename PSET > bool Parma_Polyhedra_Library::termination_test_MS_2 (const PSET & *pset_before*, const PSET & *pset_after*) [inline]

Termination test using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset_before A pointset approximating the values of loop-relevant variables *before* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x_1, \dots, x_n$ go onto space dimensions $0, \dots, n-1$.

pset_after A pointset approximating the values of loop-relevant variables *after* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x'_1, \dots, x'_n$ go onto space dimensions $0, \dots, n-1$,
- $x_1, \dots, x_n$ go onto space dimensions $n, \dots, 2n-1$,

Note that unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update. Note also that unprimed variables are assigned to different space dimensions in `pset_before` and `pset_after`.

Returns

`true` if any loop approximated by `pset` definitely terminates; `false` if the test is inconclusive. However, if `pset_before` and `pset_after` *precisely* characterize the effect of the loop body onto the loop-relevant program variables, then `true` is returned *if and only if* the loop terminates.

9.1.3.8 template<typename PSET > bool Parma_Polyhedra_Library::one_affine_ranking_function_MS (const PSET & *pset*, Generator & *mu*) [inline]

Termination test with witness ranking function using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n-1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n-1$,

where unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update.

mu When `true` is returned, this is assigned a point of space dimension $n+1$ encoding one (not further specified) affine ranking function for the loop being analyzed. The ranking function is of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ are the coefficients of `mu` corresponding to the space dimensions $n, 0, \ldots, n-1$, respectively.

Returns

`true` if any loop approximated by `pset` definitely terminates; `false` if the test is inconclusive. However, if `pset` *precisely* characterizes the effect of the loop body onto the loop-relevant program variables, then `true` is returned *if and only if* the loop terminates.

9.1.3.9 template<typename PSET > bool Parma_Polyhedra_Library::one_affine_ranking_function_MS_2 (const PSET & *pset_before*, const PSET & *pset_after*, Generator & *mu*) `[inline]`

Termination test with witness ranking function using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset_before A pointset approximating the values of loop-relevant variables *before* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x_1, \ldots, x_n$ go onto space dimensions $0, \ldots, n-1$.

pset_after A pointset approximating the values of loop-relevant variables *after* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n-1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n-1$,

Note that unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update. Note also that unprimed variables are assigned to different space dimensions in `pset_before` and `pset_after`.

Parameters

mu When `true` is returned, this is assigned a point of space dimension $n + 1$ encoding one (not further specified) affine ranking function for the loop being analyzed. The ranking function is of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ are the coefficients of `mu` corresponding to the space dimensions $n, 0, \ldots, n - 1$, respectively.

Returns

`true` if any loop approximated by `pset` definitely terminates; `false` if the test is inconclusive. However, if `pset_before` and `pset_after` *precisely* characterize the effect of the loop body onto the loop-relevant program variables, then `true` is returned *if and only if* the loop terminates.

9.1.3.10 template<typename PSET > void Parma_Polyhedra_Library::all_affine_ranking_functions_MS (const PSET & *pset*, C_Polyhedron & *mu_space*) [inline]

Termination test with ranking function space using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n - 1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n - 1$,

where unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update.

mu_space This is assigned a closed polyhedron of space dimension $n + 1$ representing the space of all the affine ranking functions for the loops that are precisely characterized by `pset`. These ranking functions are of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ identify any point of the `mu_space` polyhedron. The variables $\mu_0, \mu_1, \ldots, \mu_n$ correspond to the space dimensions of `mu_space` $n, 0, \ldots, n - 1$, respectively. When `mu_space` is empty, it means that the test is inconclusive. However, if `pset` *precisely* characterizes the effect of the loop body onto the loop-relevant program variables, then `mu_space` is empty *if and only if* the loop does *not* terminate.

9.1.3.11 template<typename PSET > void Parma_Polyhedra_Library::all_affine_ranking_functions_MS_2 (const PSET & *pset_before*, const PSET & *pset_after*, C_Polyhedron & *mu_space*) [inline]

Termination test with ranking function space using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset_before A pointset approximating the values of loop-relevant variables *before* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x_1, \ldots, x_n$ go onto space dimensions $0, \ldots, n-1$.

pset_after A pointset approximating the values of loop-relevant variables *after* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n-1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n-1$,

Note that unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update. Note also that unprimed variables are assigned to different space dimensions in `pset_before` and `pset_after`.

Parameters

mu_space This is assigned a closed polyhedron of space dimension $n+1$ representing the space of all the affine ranking functions for the loops that are precisely characterized by `pset`. These ranking functions are of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ identify any point of the `mu_space` polyhedron. The variables $\mu_0, \mu_1, \ldots, \mu_n$ correspond to the space dimensions of `mu_space` $n, 0, \ldots, n-1$, respectively. When `mu_space` is empty, it means that the test is inconclusive. However, if `pset_before` and `pset_after` *precisely* characterize the effect of the loop body onto the loop-relevant program variables, then `mu_space` is empty *if and only if* the loop does *not* terminate.

9.1.3.12 template<typename PSET > void Parma_Polyhedra_Library::all_affine_quasi_ranking_functions_MS (const PSET & *pset*, C_Polyhedron & *decreasing_mu_space*, C_Polyhedron & *bounded_mu_space*) `[inline]`

Computes the spaces of affine *quasi* ranking functions using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n-1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n-1$,

where unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update.

decreasing_mu_space This is assigned a closed polyhedron of space dimension $n + 1$ representing the space of all the decreasing affine functions for the loops that are precisely characterized by `pset`.

bounded_mu_space This is assigned a closed polyhedron of space dimension $n + 1$ representing the space of all the lower bounded affine functions for the loops that are precisely characterized by `pset`.

These quasi-ranking functions are of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ identify any point of the `decreasing_mu_space` and `bounded_mu_space` polyhedrons. The variables $\mu_0, \mu_1, \ldots, \mu_n$ correspond to the space dimensions $n, 0, \ldots, n-1$, respectively. When `decreasing_mu_space` (resp., `bounded_mu_space`) is empty, it means that the test is inconclusive. However, if `pset` *precisely* characterizes the effect of the loop body onto the loop-relevant program variables, then `decreasing_mu_space` (resp., `bounded_mu_space`) will be empty *if and only if* there is no decreasing (resp., lower bounded) affine function, so that the loop does not terminate.

9.1.3.13 template<typename PSET > void Parma_Polyhedra_Library::all_affine_quasi_ranking_functions_MS_2 (const PSET & *pset_before*, const PSET & *pset_after*, C_Polyhedron & *decreasing_mu_space*, C_Polyhedron & *bounded_mu_space*) `[inline]`

Computes the spaces of affine *quasi* ranking functions using an improvement of the method by Mesnard and Serebrenik [BMPZ10].

Template Parameters

PSET Any pointset supported by the PPL that provides the `minimized_constraints()` method.

Parameters

pset_before A pointset approximating the values of loop-relevant variables *before* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x_1, \ldots, x_n$ go onto space dimensions $0, \ldots, n-1$.

pset_after A pointset approximating the values of loop-relevant variables *after* the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:

- $x'_1, \ldots, x'_n$ go onto space dimensions $0, \ldots, n-1$,
- $x_1, \ldots, x_n$ go onto space dimensions $n, \ldots, 2n-1$,

Note that unprimed variables represent the values of the loop-relevant program variables before the update performed in the loop body, and primed variables represent the values of those program variables after the update. Note also that unprimed variables are assigned to different space dimensions in `pset_before` and `pset_after`.

Parameters

decreasing_mu_space This is assigned a closed polyhedron of space dimension $n + 1$ representing the space of all the decreasing affine functions for the loops that are precisely characterized by `pset`.

bounded_mu_space This is assigned a closed polyhedron of space dimension $n + 1$ representing the space of all the lower bounded affine functions for the loops that are precisely characterized by `pset`.

These ranking functions are of the form $\mu_0 + \sum_{i=1}^{n} \mu_i x_i$ where $\mu_0, \mu_1, \ldots, \mu_n$ identify any point of the `decreasing_mu_space` and `bounded_mu_space` polyhedrons. The variables $\mu_0, \mu_1, \ldots, \mu_n$ correspond to the space dimensions $n, 0, \ldots, n-1$, respectively. When `decreasing_mu_space` (resp., `bounded_mu_space`) is empty, it means that the test is inconclusive. However, if `pset_before` and `pset_after` *precisely* characterize the effect of the loop body onto the loop-relevant program variables, then `decreasing_mu_space` (resp., `bounded_mu_space`) will be empty *if and only if* there is no decreasing (resp., lower bounded) affine function, so that the loop does not terminate.

9.2 Parma_Polyhedra_Library::IO_Operators Namespace Reference

All input/output operators are confined to this namespace.

Functions

- std::string wrap_string (const std::string &src_string, unsigned indent_depth, unsigned preferred_-first_line_length, unsigned preferred_line_length)

  Utility function for the wrapping of lines of text.

9.2.1 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 predefined 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<<;
}
```

9.2.2 Function Documentation

9.2.2.1 std::string Parma_Polyhedra_Library::IO_Operators::wrap_string (const std::string & *src_string*, unsigned *indent_depth*, unsigned *preferred_first_line_length*, unsigned *preferred_line_length*)

Utility function for the wrapping of lines of text.

Parameters

src_string The source string holding the lines to wrap.

indent_depth The indentation depth.

preferred_first_line_length The preferred length for the first line of text.

preferred_line_length The preferred length for all the lines but the first one.

Returns

The wrapped string.

9.3 std Namespace Reference

The standard C++ namespace.

9.3.1 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.

10 Class Documentation

10.1 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter Class Reference

Artificial parameters in PIP solution trees.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_Expression.

Public Member Functions

- Artificial_Parameter ()

  Default constructor: builds a zero artificial parameter.

- Artificial_Parameter (const Linear_Expression &expr, Coefficient_traits::const_reference den)

  Constructor.

- Artificial_Parameter (const Artificial_Parameter &y)

  Copy constructor.

- Coefficient_traits::const_reference denominator () const

  Returns the normalized (i.e., positive) denominator.

- void swap (Artificial_Parameter &y)

  Swaps `*this` with `y`.

- bool operator== (const Artificial_Parameter &y) const

  Returns `true` if and only if `*this` and `y` are equal.

- bool operator!= (const Artificial_Parameter &y) const

  Returns `true` if and only if `*this` and `y` are different.

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- 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

  Returns `true` if and only if the parameter is well-formed.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &os, const PIP_Tree_Node::Artificial_Parameter &x)

  Output operator.

10.1.1 Detailed Description

Artificial parameters in PIP solution trees. These parameters are built from a linear expression combining other parameters (constant term included) divided by a positive integer denominator. Coefficients at variables indices corresponding to PIP problem variables are always zero.

10.1.2 Constructor & Destructor Documentation

10.1.2.1 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter::Artificial_Parameter (const Linear_Expression & *expr*, Coefficient_traits::const_reference *den*)

Constructor.

Builds artificial parameter $\frac{expr}{den}$.

Parameters

expr The expression that, after normalization, will form the numerator of the artificial parameter.

den The integer constant thatm after normalization, will form the denominator of the artificial parameter.

Exceptions

std::invalid_argument Thrown if `den` is zero.

Normalization will ensure that the denominator is positive.

10.1.3 Member Function Documentation

10.1.3.1 bool Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter::operator== (const Artificial_Parameter & *y*) const

Returns `true` if and only if `*this` and `y` are equal.

Note that two artificial parameters having different space dimensions are considered to be different.

10.1.4 Friends And Related Function Documentation

10.1.4.1 std::ostream & operator<< (std::ostream & *os*, const PIP_Tree_Node::Artificial_Parameter & *x*) `[related]`

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.2 Parma_Polyhedra_Library::BD_Shape< T > Class Template Reference

A bounded difference shape.

```
#include <ppl.hh>
```

Public Types

- typedef T coefficient_type_base

    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

- void ascii_dump () const

    Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

    Writes to `s` *an ASCII representation of* `*this`.

- void print () const

    Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

    Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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`.

- int32_t hash_code () const

    Returns a 32-bit hash code for `*this`.

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, Complexity_Class complexity=ANY_COMPLEXITY)

    Ordinary copy constructor.

- template< typename U >
  BD_Shape (const BD_Shape< U > &y, Complexity_Class complexity=ANY_COMPLEXITY)

    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 Congruence_System &cgs)

    Builds a BDS from a system of congruences.

- 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`.

- template<typename Interval >
  BD_Shape (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a BDS out of a box.

- BD_Shape (const Grid &grid, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a BDS out of a grid.

- template<typename U >
  BD_Shape (const Octagonal_Shape< U > &os, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a BDS from an octagonal shape.

- 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`.

- Congruence_System congruences () const

  Returns a system of (equality) congruences satisfied by `*this`.

- Congruence_System minimized_congruences () const

  Returns a minimal system of (equality) congruences satisfied by `*this` *with the same affine dimension as* `*this`.

- 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 &g) 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 &g) 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 frequency (const Linear_Expression &expr, Coefficient &freq_n, Coefficient &freq_d, Coefficient &val_n, Coefficient &val_d) const

  Returns `true` *if and only if there exist a unique value* `val` *such that* `*this` *saturates the equality* `expr = val`*.*

- 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`*.*

- bool is_disjoint_from (const BD_Shape &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are disjoint.*

- Poly_Con_Relation relation_with (const Constraint &c) const

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

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between `*this` *and the congruence* `cg`*.*

- 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 is_discrete () const

  Returns `true` *if and only if* `*this` *is discrete.*

- bool is_topologically_closed () const

Returns `true` *if and only if* `*this` *is a topologically closed subset of the vector space.*

- bool is_bounded () const

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

- bool contains_integer_point () const

  Returns `true` *if and only if* `*this` *contains at least one integer point.*

- bool constrains (Variable var) const

  Returns `true` *if and only if* `var` *is constrained in* `*this`*.*

- 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`*.*

- void add_congruence (const Congruence &cg)

  Adds a copy of congruence `cg` *to the system of congruences of* `*this`*.*

- void add_constraints (const Constraint_System &cs)

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

- void add_recycled_constraints (Constraint_System &cs)

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

- void add_congruences (const Congruence_System &cgs)

  Adds to `*this` *constraints equivalent to the congruences in* `cgs`*.*

- void add_recycled_congruences (Congruence_System &cgs)

  Adds to `*this` *constraints equivalent to the congruences in* `cgs`*.*

- void refine_with_constraint (const Constraint &c)

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

- void refine_with_congruence (const Congruence &cg)

  Uses a copy of congruence `cg` *to refine the system of bounded differences of* `*this`*.*

- void refine_with_constraints (const Constraint_System &cs)

  Uses a copy of the constraints in `cs` *to refine the system of bounded differences defining* `*this`*.*

- void refine_with_congruences (const Congruence_System &cgs)

  Uses a copy of the congruences in `cgs` *to refine the system of bounded differences defining* `*this`*.*

- void unconstrain (Variable var)

  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`*.*

- void unconstrain (const Variables_Set &vars)

  Computes the cylindrification of `*this` *with respect to the set of space dimensions* `vars`*, assigning the result to* `*this`*.*

- void intersection_assign (const BD_Shape &y)

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

- void upper_bound_assign (const BD_Shape &y)

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

- bool upper_bound_assign_if_exact (const BD_Shape &y)

  If the upper bound of `*this` *and* `y` *is exact, it is assigned to* `*this` *and* `true` *is returned, otherwise* `false` *is returned.*

- bool integer_upper_bound_assign_if_exact (const BD_Shape &y)

  If the integer upper bound of `*this` *and* `y` *is exact, it is assigned to* `*this` *and* `true` *is returned; otherwise* `false` *is returned.*

- void difference_assign (const BD_Shape &y)

  Assigns to `*this` *the smallest BD shape containing the set difference of* `*this` *and* `y`*.*

- bool simplify_using_context_assign (const BD_Shape &y)

  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`*. If* `false` *is returned, then the intersection is empty.*

- 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 generalized_affine_preimage (const Linear_Expression &lhs, Relation_Symbol relsym, const Linear_Expression &rhs)

  Assigns to `*this` *the preimage of* `*this` *with respect to the 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 BD_Shape &y)

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

- void wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_-Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

    Wraps the specified dimensions of the vector space.

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

    Possibly tightens `*this` *by dropping some points with non-integer coordinates.*

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

    Possibly tightens `*this` *by dropping some points with non-integer coordinates for the space dimensions corresponding to* `vars`.

- void topological_closure_assign ()

    Assigns to `*this` *its topological closure.*

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

    Same as H79_widening_assign(y, tp).

- 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)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`*, taken in this order.*

- void remove_space_dimensions (const Variables_Set &vars)

  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 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 &vars, Variable dest)

  Folds the space dimensions in `vars` *into* `dest`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension that a BDS can handle.

- static bool can_recycle_constraint_systems ()

  Returns `false` *indicating that this domain cannot recycle constraints.*

- static bool can_recycle_congruence_systems ()

  Returns `false` *indicating that this domain cannot recycle congruences.*

Friends

- 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.*

Related Functions

(Note that these are not member functions.)

- template<typename T >
  std::ostream & operator<< (std::ostream &s, const BD_Shape< T > &bds)

  Output operator.

- template<typename T >
  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.*

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

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

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

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

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

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

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

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

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

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

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

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

- template<typename T >
  void swap (Parma_Polyhedra_Library::BD_Shape< T > &x, Parma_Polyhedra_Library::BD_Shape< T > &y)

  Specializes `std::swap`.

10.2.1 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.

The domain of BD shapes *optimally supports*:

- tautological and inconsistent constraints and congruences;
- bounded difference constraints;
- non-proper congruences (i.e., equalities) that are expressible as bounded-difference constraints.

Depending on the method, using a constraint or congruence that is not optimally supported by the domain will either raise an exception or result in a (possibly non-optimal) upward approximation.

A constraint is a bounded difference if it has 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 bounded difference Constraint object will be mapped into a *correct* and *optimal* approximation that, depending on the expressive power of the chosen template argument `T`, may loose some precision. Also note that strict constraints are not bounded differences.

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, depending from the context, a Constraint object encoding $3x - y \leq 1$ will be either upward approximated (e.g., by safely ignoring it) or it will cause an exception.

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 optimally supported, the following code will throw an exception (caused by constraints 7, 8 and 9):

```
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);
```

10.2.2 Constructor & Destructor Documentation

10.2.2.1 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.

10.2.2.2 template<typename T > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const BD_Shape< T > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Ordinary copy constructor.

The complexity argument is ignored.

10.2.2.3 template< typename T > template< typename U > Parma_Polyhedra_Library::BD_-Shape< T >::BD_Shape (const BD_Shape< U > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a conservative, upward approximation of `y`.

The complexity argument is ignored.

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

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

The BDS inherits the space dimension of `cs`.

Parameters

cs A system of BD constraints.

Exceptions

std::invalid_argument Thrown if `cs` contains a constraint which is not optimally supported by the BD shape domain.

10.2.2.5 template< typename T > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Congruence_System & *cgs*) [inline, explicit]

Builds a BDS from a system of congruences.

The BDS inherits the space dimension of `cgs`

Parameters

cgs A system of congruences.

Exceptions

std::invalid_argument Thrown if `cgs` contains congruences which are not optimally supported by the BD shape domain.

10.2.2.6 template< typename T > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Generator_System & *gs*) [inline, explicit]

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.

10.2.2.7 template<typename T > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

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`.

10.2.2.8 template<typename T > template<typename Interval > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a BDS out of a box.

The BDS inherits the space dimension of the box. The built BDS is the most precise BDS that includes the box.

Parameters

box The box representing the BDS to be built.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.2.2.9 template<typename T > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Grid & *grid*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a BDS out of a grid.

The BDS inherits the space dimension of the grid. The built BDS is the most precise BDS that includes the grid.

Parameters

grid The grid used to build the BDS.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `grid` exceeds the maximum allowed space dimension.

10.2.2.10 template< typename T > template< typename U > Parma_Polyhedra_Library::BD_Shape< T >::BD_Shape (const Octagonal_Shape< U > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a BDS from an octagonal shape.

The BDS inherits the space dimension of the octagonal shape. The built BDS is the most precise BDS that includes the octagonal shape.

Parameters

os The octagonal shape used to build the BDS.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `os` exceeds the maximum allowed space dimension.

10.2.3 Member Function Documentation

10.2.3.1 template< typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::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.

10.2.3.2 template< typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::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.

10.2.3.3 template< typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::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.

10.2.3.4 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.2.3.5 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::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.

10.2.3.6 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *g*) 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;

g 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 g are left untouched.

10.2.3.7 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::frequency (const Linear_Expression & *expr*, Coefficient & *freq_n*, Coefficient & *freq_d*, Coefficient & *val_n*, Coefficient & *val_d*) const [inline]

Returns true if and only if there exist a unique value val such that *this saturates the equality expr = val.

Parameters

expr The linear expression for which the frequency is needed;

freq_n If true is returned, the value is set to 0; Present for interface compatibility with class Grid, where the frequency can have a non-zero value;

freq_d If true is returned, the value is set to 1;

val_n The numerator of val;

val_d The denominator of val;

Exceptions

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `false` is returned, then `freq_n`, `freq_d`, `val_n` and `val_d` are left untouched.

10.2.3.8 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::contains (const BD_Shape< T > & *y*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.2.3.9 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.

10.2.3.10 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::is_disjoint_from (const BD_Shape< T > & *y*) const [inline]

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.

10.2.3.11 template<typename T > Poly_Con_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation_with (const Constraint & *c*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.2.3.12 template<typename T > Poly_Con_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation_with (const Congruence & *cg*) const [inline]

Returns the relations holding between *this and the congruence cg.

Exceptions

std::invalid_argument Thrown if *this and congruence cg are dimension-incompatible.

10.2.3.13 template<typename T > Poly_Gen_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation_with (const Generator & *g*) const [inline]

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

Exceptions

std::invalid_argument Thrown if *this and generator g are dimension-incompatible.

10.2.3.14 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::constrains (Variable *var*) const [inline]

Returns true if and only if var is constrained in *this.

Exceptions

std::invalid_argument Thrown if var is not a space dimension of *this.

10.2.3.15 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_constraint (const Constraint & *c*) [inline]

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

Parameters

c The constraint to be added.

Exceptions

std::invalid_argument Thrown if *this and constraint c are dimension-incompatible, or c is not optimally supported by the BD shape domain.

10.2.3.16 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_congruence (const Congruence & *cg*) [inline]

Adds a copy of congruence cg to the system of congruences of *this.

Parameters

cg The congruence to be added.

Exceptions

std::invalid_argument Thrown if *this and congruence cg are dimension-incompatible, or cg is not optimally supported by the BD shape domain.

10.2.3.17 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.

Exceptions

std::invalid_argument Thrown if *this and cs are dimension-incompatible, or cs contains a constraint which is not optimally supported by the BD shape domain.

10.2.3.18 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_recycled_constraints (Constraint_System & *cs*) [inline]

Adds the constraints in cs to the system of constraints of *this.

Parameters

cs The constraint system to be added to *this. The constraints in cs may be recycled.

Exceptions

std::invalid_argument Thrown if *this and cs are dimension-incompatible, or cs contains a constraint which is not optimally supported by the BD shape domain.

Warning

The only assumption that can be made on cs upon successful or exceptional return is that it can be safely destroyed.

10.2.3.19 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_congruences (const Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

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 dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the BD shape domain.

10.2.3.20 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_recycled_congruences (Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

Parameters

cgs Contains the congruences that will be added to the system of constraints of `*this`. Its elements may be recycled.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the BD shape domain.

Warning

The only assumption that can be made on `cgs` upon successful or exceptional return is that it can be safely destroyed.

10.2.3.21 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::refine_with_constraint (const Constraint & *c*) [inline]

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

Parameters

c The constraint. If it is not a bounded difference, it will be ignored.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.2.3.22 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::refine_with_congruence (const Congruence & *cg*) [inline]

Uses a copy of congruence `cg` to refine the system of bounded differences of `*this`.

Parameters

cg The congruence. If it is not a bounded difference equality, it will be ignored.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

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

Uses a copy of the constraints in `cs` to refine the system of bounded differences defining `*this`.

Parameters

cs The constraint system to be used. Constraints that are not bounded differences are ignored.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.2.3.24 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Uses a copy of the congruences in `cgs` to refine the system of bounded differences defining `*this`.

Parameters

cgs The congruence system to be used. Congruences that are not bounded difference equalities are ignored.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.2.3.25 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::unconstrain (Variable *var*) [inline]

Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.2.3.26 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::unconstrain (const Variables_Set & *vars*) [inline]

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.2.3.27 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::intersection_assign (const BD_Shape< T > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.2.3.28 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::upper_bound_assign (const BD_Shape< T > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.2.3.29 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::upper_bound_assign_if_exact (const BD_Shape< T > & *y*) [inline]

If the upper bound 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.

10.2.3.30 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::integer_upper_bound_assign_if_exact (const BD_Shape< T > & *y*) `[inline]`

If the *integer* upper bound 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.

Note

The integer upper bound of two rational BDS is the smallest rational BDS containing all the integral points of the two arguments. This method requires that the coefficient type parameter `T` is an integral type.

10.2.3.31 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::difference_assign (const BD_Shape< T > & *y*) `[inline]`

Assigns to `*this` the smallest BD shape containing the set difference of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.2.3.32 template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::simplify_using_context_assign (const BD_Shape< T > & *y*) `[inline]`

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.2.3.33 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()`) `[inline]`

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`.

10.2.3.34 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()`) `[inline]`

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`.

10.2.3.35 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()`) `[inline]`

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.

10.2.3.36 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) [inline]

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.

10.2.3.37 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()) [inline]

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.

10.2.3.38 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) [inline]

Assigns to `*this` the preimage 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.

10.2.3.39 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()) [inline]`

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`.

10.2.3.40 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()) [inline]`

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`.

10.2.3.41 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.

10.2.3.42 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System * *pcs* = 0, unsigned *complexity_threshold* = 16, bool *wrap_individually* = true) [inline]

Wraps the specified dimensions of the vector space.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be wrapped.

w The width of the bounded integer type corresponding to all the dimensions to be wrapped.

r The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

o The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.

pcs Possibly null pointer to a constraint system whose variables are contained in `vars`. If `*pcs` depends on variables not in `vars`, the behavior is undefined. When non-null, the pointed-to constraint system is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation with the constraints in `*pcs`.

complexity_threshold A precision parameter of the wrapping operator: higher values result in possibly improved precision.

wrap_individually `true` if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

std::invalid_argument Thrown if `*pcs` is dimension-incompatible with `vars`, or if `*this` is dimension-incompatible `vars` or with `*pcs`.

10.2.3.43 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.2.3.44 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.2.3.45 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.

10.2.3.46 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) [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`.

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.

10.2.3.47 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::BHMZ05_widening_assign (const BD_Shape< T > & *y*, unsigned * *tp* = 0) [inline]

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.

10.2.3.48 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) [inline]

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.

10.2.3.49 template< typename T > void Parma_Polyhedra_Library::BD_Shape< T >::CC76_narrowing_assign (const BD_Shape< T > & *y*) [inline]

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 *decreasing* iteration sequence). Hence, the call `x.CC76_narrowing_assign(y)` will assign to `x` the result of the computation $\mathtt{y} \Delta \mathtt{x}$.

10.2.3.50 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) [inline]

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.

10.2.3.51 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.

10.2.3.52 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.

10.2.3.53 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_embed (dimension_type *m*) [inline]

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

$$\left\{ (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \;\middle|\; (x, y)^{\mathrm{T}} \in \mathcal{B} \right\}.$$

10.2.3.54 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_project (dimension_type *m*) [inline]

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

$$\{ (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{B} \}.$$

10.2.3.55 template< typename T > void Parma_Polyhedra_Library::BD_Shape< T >::concatenate_assign (const BD_Shape< T > & *y*) `[inline]`

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()`.

10.2.3.56 template< typename T > void Parma_Polyhedra_Library::BD_Shape< T >::remove_space_dimensions (const Variables_Set & *vars*) `[inline]`

Removes all the specified dimensions.

Parameters

vars 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 `vars`.

10.2.3.57 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`.

10.2.3.58 template<typename T > template<typename Partial_Function > void Parma_Polyhedra_Library::BD_Shape< T >::map_space_dimensions (const Partial_Function & *pfunc*) [inline]

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 type parameter 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 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.

10.2.3.59 template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::expand_space_dimension (Variable *var*, dimension_type *m*) [inline]

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 $n, n+1, \dots, n+m-1$.

10.2.3.60 template< typename T > void Parma_Polyhedra_Library::BD_Shape< T >::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*) [inline]

Folds the space dimensions in `vars` into `dest`.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 `dest` or with one of the Variable objects contained in `vars`. Also thrown if `dest` is contained in `vars`.

If `*this` has space dimension n, with $n > 0$, `dest` has space dimension $k \leq n$, `vars` is a set of variables whose maximum space dimension is also less than or equal to n, and `dest` is not a member of `vars`, then the space dimensions corresponding to variables in `vars` are folded into the k-th space dimension.

10.2.3.61 template< typename T > int32_t Parma_Polyhedra_Library::BD_Shape< T >::hash_code () const [inline]

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.2.4 Friends And Related Function Documentation

10.2.4.1 template< typename T > bool operator== (const BD_Shape< T > & *x*, const BD_Shape< T > & *y*) [friend]

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.

10.2.4.2 template< typename T > std::ostream & operator<< (std::ostream & *s*, const BD_Shape< T > & *c*) [related]

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 ", ".

10.2.4.3 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.

10.2.4.4 template<typename To , typename T > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.2.4.5 template<typename Temp , typename To , typename T > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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`.

10.2.4.6 template<typename To , typename T > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_Policy>.

10.2.4.7 template< typename Temp , typename To , typename T > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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`.

10.2.4.8 template< typename To , typename T > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_Policy>.

10.2.4.9 template< typename Temp , typename To , typename T > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const BD_Shape< T > & *x*, const BD_Shape< T > & *y*, 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`.

10.2.4.10 template<typename T > void swap (Parma_Polyhedra_Library::BD_Shape< T > & *x*, Parma_Polyhedra_Library::BD_Shape< T > & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.3 Parma_Polyhedra_Library::BHRZ03_Certificate Class Reference

The convergence certificate for the BHRZ03 widening operator.

`#include <ppl.hh>`

Classes

- struct Compare

  A total ordering on BHRZ03 certificates.

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`.

10.3.1 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.

10.3.2 Member Function Documentation

10.3.2.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.4 Parma_Polyhedra_Library::Box< ITV > Class Template Reference

A not necessarily closed, iso-oriented hyperrectangle.

`#include <ppl.hh>`

Public Types

- typedef ITV interval_type

  The type of intervals used to implement the box.

Public Member Functions

- const ITV & get_interval (Variable var) const

  Returns a reference the interval that bounds `var`.

- void set_interval (Variable var, const ITV &i)

  Sets to `i` *the interval that bounds* `var`.

- bool get_lower_bound (dimension_type k, bool &closed, Coefficient &n, Coefficient &d) const

If the `k`-th space dimension is unbounded below, returns `false`. Otherwise returns `true` and set `closed`, `n` and `d` accordingly.

- bool get_upper_bound (dimension_type k, bool &closed, Coefficient &n, Coefficient &d) const

  If the `k`-th space dimension is unbounded above, returns `false`. Otherwise returns `true` and set `closed`, `n` and `d` accordingly.

- Constraint_System constraints () const

  Returns a system of constraints defining `*this`.

- Constraint_System minimized_constraints () const

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

- Congruence_System congruences () const

  Returns a system of congruences approximating `*this`.

- Congruence_System minimized_congruences () const

  Returns a minimized system of congruences approximating `*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`.

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- void set_empty ()

  Causes the box to become empty, i.e., to represent the empty set.

Constructors, Assignment, Swap and Destructor

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

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

- Box (const Box &y, Complexity_Class complexity=ANY_COMPLEXITY)

  Ordinary copy constructor.

- template<typename Other_ITV >
  Box (const Box< Other_ITV > &y, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a conservative, upward approximation of `y`.

- Box (const Constraint_System &cs)

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

- Box (const Constraint_System &cs, Recycle_Input dummy)

    Builds a box recycling a system of constraints `cs`.

- Box (const Generator_System &gs)

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

- Box (const Generator_System &gs, Recycle_Input dummy)

    Builds a box recycling the system of generators `gs`.

- Box (const Congruence_System &cgs)
- Box (const Congruence_System &cgs, Recycle_Input dummy)
- template<typename T >
  Box (const BD_Shape< T > &bds, Complexity_Class complexity=POLYNOMIAL_-COMPLEXITY)

    Builds a box containing the BDS `bds`.

- template<typename T >
  Box (const Octagonal_Shape< T > &oct, Complexity_Class complexity=POLYNOMIAL_-COMPLEXITY)

    Builds a box containing the octagonal shape `oct`.

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

    Builds a box containing the polyhedron `ph`.

- Box (const Grid &ph, Complexity_Class complexity=POLYNOMIAL_COMPLEXITY)

    Builds a box containing the grid `gr`.

- template<typename D1 , typename D2 , typename R >
  Box (const Partially_Reduced_Product< D1, D2, R > &dp, Complexity_Class complexity=ANY_COMPLEXITY)

    Builds a box containing the partially reduced product `dp`.

- Box & operator= (const Box &y)

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

- void swap (Box &y)

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

Member Functions that Do Not Modify the Box

- 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`.

- bool is_empty () const

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

- bool is_universe () const

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

- bool is_topologically_closed () const

Returns `true` *if and only if* `*this` *is a topologically closed subset of the vector space.*

- 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 a bounded box.*

- bool contains_integer_point () const

  Returns `true` *if and only if* `*this` *contains at least one integer point.*

- bool constrains (Variable var) const

  Returns `true` *if and only if* `var` *is constrained in* `*this`.

- Poly_Con_Relation relation_with (const Constraint &c) const

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

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between `*this` *and the congruence* `cg`.

- Poly_Gen_Relation relation_with (const Generator &g) const

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

- 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 &g) 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 &g) 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 frequency (const Linear_Expression &expr, Coefficient &freq_n, Coefficient &freq_d, Coefficient &val_n, Coefficient &val_d) const

  Returns `true` *if and only if there exist a unique value* `val` *such that* `*this` *saturates the equality* `expr = val`.

- bool contains (const Box &y) const

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

- bool strictly_contains (const Box &y) const
  Returns `true` *if and only if* `*this` *strictly contains* `y`*.*
- bool is_disjoint_from (const Box &y) const
  Returns `true` *if and only if* `*this` *and* `y` *are disjoint.*
- bool OK () const
  Returns `true` *if and only if* `*this` *satisfies all its invariants.*

Space-Dimension Preserving Member Functions that May Modify the Box

- void add_constraint (const Constraint &c)
  Adds a copy of constraint `c` *to the system of constraints defining* `*this`*.*
- void add_constraints (const Constraint_System &cs)
  Adds the constraints in `cs` *to the system of constraints defining* `*this`*.*
- void add_recycled_constraints (Constraint_System &cs)
  Adds the constraints in `cs` *to the system of constraints defining* `*this`*.*
- void add_congruence (const Congruence &cg)
  Adds to `*this` *a constraint equivalent to the congruence* `cg`*.*
- void add_congruences (const Congruence_System &cgs)
  Adds to `*this` *constraints equivalent to the congruences in* `cgs`*.*
- void add_recycled_congruences (Congruence_System &cgs)
  Adds to `*this` *constraints equivalent to the congruences in* `cgs`*.*
- void refine_with_constraint (const Constraint &c)
  Use the constraint `c` *to refine* `*this`*.*
- void refine_with_constraints (const Constraint_System &cs)
  Use the constraints in `cs` *to refine* `*this`*.*
- void refine_with_congruence (const Congruence &cg)
  Use the congruence `cg` *to refine* `*this`*.*
- void refine_with_congruences (const Congruence_System &cgs)
  Use the congruences in `cgs` *to refine* `*this`*.*
- void propagate_constraint (const Constraint &c)
  Use the constraint `c` *for constraint propagation on* `*this`*.*
- void propagate_constraints (const Constraint_System &cs, dimension_type max_iterations=0)
  Use the constraints in `cs` *for constraint propagagion on* `*this`*.*
- void unconstrain (Variable var)
  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`*.*
- void unconstrain (const Variables_Set &vars)

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

- void intersection_assign (const Box &y)

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

- void upper_bound_assign (const Box &y)

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

- bool upper_bound_assign_if_exact (const Box &y)

  If the upper bound of `*this` and `y` is exact, it is assigned to `*this` and `true` is returned, otherwise `false` is returned.

- void difference_assign (const Box &y)

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

- bool simplify_using_context_assign (const Box &y)

  Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

- 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, 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, 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, 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, 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 Box &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 wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

  Wraps the specified dimensions of the vector space.

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates.*

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates for the space dimensions corresponding to* `vars`.

- template<typename T >
  Enable_If< Is_Same< T, Box >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type CC76_widening_assign (const T &y, unsigned *tp=0)

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

- template<typename T , typename Iterator >
  Enable_If< Is_Same< T, Box >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type CC76_widening_assign (const T &y, Iterator first, Iterator last)

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

- void widening_assign (const Box &y, unsigned *tp=0)

  Same as CC76_widening_assign(y, tp).

- void limited_CC76_extrapolation_assign (const Box &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`.

- template<typename T >
  Enable_If< Is_Same< T, Box >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type CC76_narrowing_assign (const T &y)

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

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 box into the new space.*

- void add_space_dimensions_and_project (dimension_type m)

Adds m new dimensions to the box and does not embed it in the new vector space.

- void concatenate_assign (const Box &y)

  *Seeing a box 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 &vars)

  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 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 &vars, Variable dest)

  Folds the space dimensions in vars into dest.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension that a Box can handle.

- static bool can_recycle_constraint_systems ()

  Returns false indicating that this domain does not recycle constraints.

- static bool can_recycle_congruence_systems ()

  Returns false indicating that this domain does not recycle congruences.

Friends

- bool operator== (const Box< ITV > &x, const Box< ITV > &y)

  Returns true if and only if x and y are the same box.

Related Functions

(Note that these are not member functions.)

- template<typename ITV >
  bool operator!= (const Box< ITV > &x, const Box< ITV > &y)

  Returns true if and only if x and y aren't the same box.

- template<typename ITV >
  std::ostream & operator<< (std::ostream &s, const Box< ITV > &box)

Output operator.

- template<typename To , typename ITV >
  bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir)

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

- template<typename Temp , typename To , typename ITV >
  bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

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

- template<typename To , typename ITV >
  bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir)

  Computes the euclidean distance between x and y.

- template<typename Temp , typename To , typename ITV >
  bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

  Computes the euclidean distance between x and y.

- template<typename To , typename ITV >
  bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir)

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

- template<typename Temp , typename To , typename ITV >
  bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)

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

10.4.1 Detailed Description

template<typename ITV> class Parma_Polyhedra_Library::Box< ITV >

A not necessarily closed, iso-oriented hyperrectangle. A Box object represents the smash product of n not necessarily closed and possibly unbounded intervals represented by objects of class `ITV`, where n is the space dimension of the box.

An *interval constraint* (resp., *interval congruence*) is a syntactic constraint (resp., congruence) that only mentions a single space dimension.

The Box domain *optimally supports*:

- tautological and inconsistent constraints and congruences;
- the interval constraints that are optimally supported by the template argument class `ITV`;
- the interval congruences that are optimally supported by the template argument class `ITV`.

Depending on the method, using a constraint or congruence that is not optimally supported by the domain will either raise an exception or result in a (possibly non-optimal) upward approximation.

The user interface for the Box domain is meant to be as similar as possible to the one developed for the polyhedron class C_Polyhedron.

10.4.2 Constructor & Destructor Documentation

10.4.2.1 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (dimension_type *num_dimensions* = 0, Degenerate_Element *kind* = UNIVERSE) [inline, explicit]

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

Parameters

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

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

10.4.2.2 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Box< ITV > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Ordinary copy constructor.

The complexity argument is ignored.

10.4.2.3 template<typename ITV > template<typename Other_ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Box< Other_ITV > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a conservative, upward approximation of `y`.

The complexity argument is ignored.

10.4.2.4 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Constraint_System & *cs*) [inline, explicit]

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

The box inherits the space dimension of `cs`.

Parameters

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

10.4.2.5 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Constraint_System & *cs*, Recycle_Input *dummy*) [inline]

Builds a box recycling a system of constraints `cs`.

The box inherits the space dimension of `cs`.

Parameters

- ***cs*** A system of constraints: constraints that are not interval constraints are ignored (even though they may have contributed to the space dimension).
- ***dummy*** A dummy tag to syntactically differentiate this one from the other constructors.

10.4.2.6 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Generator_System & *gs*) [inline, explicit]

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

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

Exceptions

- ***std::invalid_argument*** Thrown if the system of generators is not empty but has no points.

10.4.2.7 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Generator_System & *gs*, Recycle_Input *dummy*) [inline]

Builds a box recycling the system of generators `gs`.

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

Parameters

- ***gs*** The generator system describing the polyhedron to be approximated.
- ***dummy*** A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

- ***std::invalid_argument*** Thrown if the system of generators is not empty but has no points.

10.4.2.8 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Congruence_System & *cgs*) [inline, explicit]

Builds the smallest box containing the grid defined by a system of congruences `cgs`. The box inherits the space dimension of `cgs`.

Parameters

cgs A system of congruences: congruences that are not non-relational equality constraints are ignored (though they may have contributed to the space dimension).

10.4.2.9 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Congruence_System & *cgs*, Recycle_Input *dummy*) `[inline]`

Builds the smallest box containing the grid defined by a system of congruences `cgs`, recycling `cgs`. The box inherits the space dimension of `cgs`.

Parameters

cgs A system of congruences: congruences that are not non-relational equality constraints are ignored (though they will contribute to the space dimension).

dummy A dummy tag to syntactically differentiate this one from the other constructors.

10.4.2.10 template<typename ITV > template<typename T > Parma_Polyhedra_Library::Box< ITV >::Box (const BD_Shape< T > & *bds*, Complexity_Class *complexity* = `POLYNOMIAL_COMPLEXITY`) `[inline, explicit]`

Builds a box containing the BDS `bds`.

Builds the smallest box containing `bds` using a polynomial algorithm. The `complexity` argument is ignored.

10.4.2.11 template<typename ITV > template<typename T > Parma_Polyhedra_Library::Box< ITV >::Box (const Octagonal_Shape< T > & *oct*, Complexity_Class *complexity* = `POLYNOMIAL_COMPLEXITY`) `[inline, explicit]`

Builds a box containing the octagonal shape `oct`.

Builds the smallest box containing `oct` using a polynomial algorithm. The `complexity` argument is ignored.

10.4.2.12 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Polyhedron & *ph*, Complexity_Class *complexity* = `ANY_COMPLEXITY`) `[inline, explicit]`

Builds a box containing the polyhedron `ph`.

Builds a box containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the built box is the smallest one containing `ph`.

10.4.2.13 template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Grid & *ph*, Complexity_Class *complexity* = POLYNOMIAL_COMPLEXITY) [inline, explicit]

Builds a box containing the grid `gr`.

Builds the smallest box containing `gr` using a polynomial algorithm. The `complexity` argument is ignored.

10.4.2.14 template<typename ITV > template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Box< ITV >::Box (const Partially_Reduced_Product< D1, D2, R > & *dp*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a box containing the partially reduced product `dp`.

Builds a box containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`.

10.4.3 Member Function Documentation

10.4.3.1 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::constrains (Variable *var*) const [inline]

Returns `true` if and only if `var` is constrained in `*this`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.4.3.2 template<typename ITV > Poly_Con_Relation Parma_Polyhedra_Library::Box< ITV >::relation_with (const Constraint & *c*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.4.3.3 template<typename ITV > Poly_Con_Relation Parma_Polyhedra_Library::Box< ITV >::relation_with (const Congruence & *cg*) const [inline]

Returns the relations holding between `*this` and the congruence `cg`.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `cg` are dimension-incompatible.

10.4.3.4 template<typename ITV > Poly_Gen_Relation Parma_Polyhedra_Library::Box< ITV >::relation_with (const Generator & *g*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and generator `g` are dimension-incompatible.

10.4.3.5 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::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.

10.4.3.6 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::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.

10.4.3.7 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::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.

10.4.3.8 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.4.3.9 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::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.

10.4.3.10 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.4.3.11 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::frequency (const Linear_Expression & *expr*, Coefficient & *freq_n*, Coefficient & *freq_d*, Coefficient & *val_n*, Coefficient & *val_d*) const [inline]

Returns `true` if and only if there exist a unique value `val` such that `*this` saturates the equality `expr = val`.

Parameters

expr The linear expression for which the frequency is needed;

freq_n If `true` is returned, the value is set to 0; Present for interface compatibility with class Grid, where the frequency can have a non-zero value;

freq_d If `true` is returned, the value is set to 1;

val_n The numerator of `val`;

val_d The denominator of `val`;

Exceptions

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `false` is returned, then `freq_n`, `freq_d`, `val_n` and `val_d` are left untouched.

10.4.3.12 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::contains (const Box< ITV > & *y*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `x` and `y` are dimension-incompatible.

10.4.3.13 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::strictly_contains (const Box< ITV > & *y*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `x` and `y` are dimension-incompatible.

10.4.3.14 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::is_disjoint_from (const Box< ITV > & *y*) const [inline]

Returns `true` if and only if `*this` and `y` are disjoint.

Exceptions

std::invalid_argument Thrown if `x` and `y` are dimension-incompatible.

10.4.3.15 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_constraint (const Constraint & *c*) [inline]

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

Parameters

c The constraint to be added.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible, or `c` is not optimally supported by the Box domain.

10.4.3.16 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_constraints (const Constraint_System & *cs*) [inline]

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

Parameters

cs The constraints to be added.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible, or `cs` contains a constraint which is not optimally supported by the box domain.

10.4.3.17 template<typename T > void Parma_Polyhedra_Library::Box< T >::add_recycled_constraints (Constraint_System & *cs*) [inline]

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

Parameters

cs The constraints to be added. They may be recycled.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible, or `cs` contains a constraint which is not optimally supported by the box domain.

Warning

The only assumption that can be made on `cs` upon successful or exceptional return is that it can be safely destroyed.

10.4.3.18 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_congruence (const Congruence & *cg*) [inline]

Adds to `*this` a constraint equivalent to the congruence `cg`.

Parameters

cg The congruence to be added.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible, or `cg` is not optimally supported by the box domain.

10.4.3.19 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_congruences (const Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

Parameters

cgs The congruences to be added.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the box domain.

10.4.3.20 template<typename T > void Parma_Polyhedra_Library::Box< T >::add_recycled_congruences (Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

Parameters

cgs The congruence system to be added to `*this`. The congruences in `cgs` may be recycled.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the box domain.

Warning

The only assumption that can be made on `cgs` upon successful or exceptional return is that it can be safely destroyed.

10.4.3.21 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_constraint (const Constraint & *c*) [inline]

Use the constraint `c` to refine `*this`.

Parameters

c The constraint to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `c` are dimension-incompatible.

10.4.3.22 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_constraints (const Constraint_System & *cs*) [inline]

Use the constraints in `cs` to refine `*this`.

Parameters

cs The constraints to be used for refinement. To avoid termination problems, each constraint in `cs` will be used for a single refinement step.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

Note

The user is warned that the accuracy of this refinement operator depends on the order of evaluation of the constraints in `cs`, which is in general unpredictable. If a fine control on such an order is needed, the user should consider calling the method `refine_with_constraint(const Constraint& c)` inside an appropriate looping construct.

10.4.3.23 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_congruence (const Congruence & *cg*) [inline]

Use the congruence `cg` to refine `*this`.

Parameters

cg The congruence to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cg` are dimension-incompatible.

10.4.3.24 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Use the congruences in `cgs` to refine `*this`.

Parameters

cgs The congruences to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.4.3.25 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::propagate_constraint (const Constraint & *c*) [inline]

Use the constraint c for constraint propagation on *this.

Parameters

c The constraint to be used for constraint propagation.

Exceptions

std::invalid_argument Thrown if *this and c are dimension-incompatible.

10.4.3.26 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::propagate_constraints (const Constraint_System & *cs*, dimension_type *max_iterations* = 0) [inline]

Use the constraints in cs for constraint propagagion on *this.

Parameters

cs The constraints to be used for constraint propagation.

max_iterations The maximum number of propagation steps for each constraint in cs. If zero (the default), the number of propagations will be unbounded, possibly resulting in an infinite loop.

Exceptions

std::invalid_argument Thrown if *this and cs are dimension-incompatible.

Warning

This method may lead to non-termination if max_iterations is 0.

10.4.3.27 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::unconstrain (Variable *var*) [inline]

Computes the cylindrification of *this with respect to space dimension var, assigning the result to *this.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if var is not a space dimension of *this.

10.4.3.28 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::unconstrain (const Variables_Set & *vars*) [inline]

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.4.3.29 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::intersection_assign (const Box< ITV > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.4.3.30 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::upper_bound_assign (const Box< ITV > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.4.3.31 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::upper_bound_assign_if_exact (const Box< ITV > & *y*) [inline]

If the upper bound 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.

10.4.3.32 template< typename ITV > void Parma_Polyhedra_Library::Box< ITV >::difference_assign (const Box< ITV > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.4.3.33 template< typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::simplify_using_context_assign (const Box< ITV > & *y*) [inline]

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.4.3.34 template< typename ITV > void Parma_Polyhedra_Library::Box< ITV >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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`.

10.4.3.35 template< typename ITV > void Parma_Polyhedra_Library::Box< ITV >::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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`.

10.4.3.36 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.4.3.37 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.4.3.38 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) [inline]

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`.

10.4.3.39 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) [inline]

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`.

10.4.3.40 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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`.

10.4.3.41 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.4.3.42 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::time_elapse_assign (const Box< ITV > & *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.

10.4.3.43 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System ∗ *pcs* = 0, unsigned *complexity_threshold* = 16, bool *wrap_individually* = true) [inline]

Wraps the specified dimensions of the vector space.

Parameters

- ***vars*** The set of Variable objects corresponding to the space dimensions to be wrapped.
- ***w*** The width of the bounded integer type corresponding to all the dimensions to be wrapped.
- ***r*** The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
- ***o*** The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
- ***pcs*** Possibly null pointer to a constraint system. When non-null, the pointed-to constraint system is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation with the constraints in `*pcs`.
- ***complexity_threshold*** A precision parameter which is ignored for the Box domain.
- ***wrap_individually*** A precision parameter which is ignored for the Box domain.

Exceptions

- ***std::invalid_argument*** Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars` or with `*pcs`.

10.4.3.44 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

- ***complexity*** The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.4.3.45 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens *this by dropping some points with non-integer coordinates for the space dimensions corresponding to vars.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if complexity is ANY_COMPLEXITY.

10.4.3.46 template<typename ITV > template<typename T > Enable_If< Is_Same< T, Box< ITV > >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type Parma_Polyhedra_Library::Box< ITV >::CC76_widening_assign (const T & *y*, unsigned * *tp* = 0) [inline]

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

Parameters

y A box 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.

10.4.3.47 template<typename ITV > template<typename T , typename Iterator > Enable_If< Is_Same< T, Box< ITV > >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type Parma_Polyhedra_Library::Box< ITV >::CC76_widening_assign (const T & *y*, Iterator *first*, Iterator *last*) [inline]

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

Parameters

y A box that *must* be contained in *this.

first An iterator that points to the first stop-point.

last An iterator that points one past the last stop-point.

Exceptions

std::invalid_argument Thrown if *this and y are dimension-incompatible.

10.4.3.48 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::limited_CC76_extrapolation_assign (const Box< ITV > & *y*, const Constraint_System & *cs*, unsigned * *tp* = 0) [inline]

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 box that *must* be contained in `*this`.

cs The system of constraints used to improve the widened box.

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.

10.4.3.49 template<typename ITV > template<typename T > Enable_If< Is_Same< T, Box< ITV > >::value &&Is_Same_Or_Derived< Interval_Base, ITV >::value, void >::type Parma_Polyhedra_Library::Box< ITV >::CC76_narrowing_assign (const T & *y*) [inline]

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

Parameters

y A Box 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 *decreasing* iteration sequence). Hence, the call `x.CC76_narrowing_assign(y)` will assign to `x` the result of the computation $\mathtt{y}\Delta\mathtt{x}$.

10.4.3.50 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_space_dimensions_and_embed (dimension_type *m*) [inline]

Adds `m` new dimensions and embeds the old box 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 box, which is defined by a system of interval constraints in which the variables running through the new dimensions are unconstrained. For instance, when starting from the box $\mathcal{B} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the box

$$\{ (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{B} \}.$$

10.4.3.51 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::add_space_dimensions_and_project (dimension_type *m*) `[inline]`

Adds `m` new dimensions to the box 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 box, 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 box $\mathcal{B} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the box

$$\{ (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{B} \}.$$

10.4.3.52 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::concatenate_assign (const Box< ITV > & *y*) `[inline]`

Seeing a box 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 boxes corresponding, on entry, to `*this` and `y`, respectively. Upon successful completion, `*this` will represent the box $R \subseteq \mathbb{R}^{n+m}$ 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`.

10.4.3.53 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::remove_space_dimensions (const Variables_Set & *vars*) `[inline]`

Removes all the specified dimensions.

Parameters

vars 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 `vars`.

10.4.3.54 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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`.

10.4.3.55 template<typename ITV > template<typename Partial_Function > void Parma_Polyhedra_Library::Box< ITV >::map_space_dimensions (const Partial_Function & *pfunc*) `[inline]`

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 type parameter 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 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.

10.4.3.56 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::expand_space_dimension (Variable *var*, dimension_type *m*) [inline]

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 $n, n+1, \ldots, n+m-1$.

10.4.3.57 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*) [inline]

Folds the space dimensions in `vars` into `dest`.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 `dest` or with one of the Variable objects contained in `vars`. Also thrown if `dest` is contained in `vars`.

If `*this` has space dimension n, with $n > 0$, `dest` has space dimension $k \leq n$, `vars` is a set of variables whose maximum space dimension is also less than or equal to n, and `dest` is not a member of `vars`, then the space dimensions corresponding to variables in `vars` are folded into the k-th space dimension.

10.4.3.58 template<typename ITV > const ITV & Parma_Polyhedra_Library::Box< ITV >::get_interval (Variable *var*) const [inline]

Returns a reference the interval that bounds `var`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.4.3.59 template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::set_interval (Variable *var*, const ITV & *i*) [inline]

Sets to i the interval that bounds var.

Exceptions

std::invalid_argument Thrown if var is not a space dimension of *this.

10.4.3.60 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::get_lower_bound (dimension_type *k*, bool & *closed*, Coefficient & *n*, Coefficient & *d*) const [inline]

If the k-th space dimension is unbounded below, returns false. Otherwise returns true and set closed, n and d accordingly.

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.

An undefined behavior is obtained if k is greater than or equal to the space dimension of *this.

10.4.3.61 template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::get_upper_bound (dimension_type *k*, bool & *closed*, Coefficient & *n*, Coefficient & *d*) const [inline]

If the k-th space dimension is unbounded above, returns false. Otherwise returns true and set closed, n and d accordingly.

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.

An undefined behavior is obtained if k is greater than or equal to the space dimension of *this.

10.4.4 Friends And Related Function Documentation

10.4.4.1 template<typename ITV > bool operator== (const Box< ITV > & *x*, const Box< ITV > & *y*) [friend]

Returns true if and only if x and y are the same box.

Note that x and y may be dimension-incompatible boxes: in this case, the value false is returned.

10.4.4.2 template<typename ITV > bool operator!= (const Box< ITV > & *x*, const Box< ITV > & *y*) [related]

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

Note that `x` and `y` may be dimension-incompatible boxes: in this case, the value `true` is returned.

10.4.4.3 template<typename ITV > std::ostream & operator<< (std::ostream & *s*, const Box< ITV > & *box*) [related]

Output operator.

10.4.4.4 template<typename To , typename ITV > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.4.4.5 template<typename Temp , typename To , typename ITV > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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`.

10.4.4.6 template< typename To , typename ITV > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.4.4.7 template< typename Temp , typename To , typename ITV > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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.

10.4.4.8 template< typename To , typename ITV > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.4.4.9 template<typename Temp , typename To , typename ITV > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Box< ITV > & *x*, const Box< ITV > & *y*, 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`.

The documentation for this class was generated from the following file:

- ppl.hh

10.5 Parma_Polyhedra_Library::C_Polyhedron Class Reference

A closed convex polyhedron.

`#include <ppl.hh>`

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, Recycle_Input dummy)
  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, Recycle_Input dummy)
  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, Recycle_Input dummy)
  Builds a C polyhedron recycling a system of congruences.

- C_Polyhedron (const NNC_Polyhedron &y, Complexity_Class complexity=ANY_-COMPLEXITY)

    Builds a C polyhedron representing the topological closure of the NNC polyhedron `y`*.*

- template<typename Interval >
  C_Polyhedron (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

    Builds a C polyhedron out of a box.

- template<typename U >
  C_Polyhedron (const BD_Shape< U > &bd, Complexity_Class complexity=ANY_-COMPLEXITY)

    Builds a C polyhedron out of a BD shape.

- template<typename U >
  C_Polyhedron (const Octagonal_Shape< U > &os, Complexity_Class complexity=ANY_-COMPLEXITY)

    Builds a C polyhedron out of an octagonal shape.

- C_Polyhedron (const Grid &grid, Complexity_Class complexity=ANY_COMPLEXITY)

    Builds a C polyhedron out of a grid.

- C_Polyhedron (const C_Polyhedron &y, Complexity_Class complexity=ANY_COMPLEXITY)

    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).

10.5.1 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.

10.5.2 Constructor & Destructor Documentation

10.5.2.1 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.

10.5.2.2 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.

10.5.2.3 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Constraint_System & *cs*, Recycle_Input *dummy*) [inline]

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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

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

10.5.2.4 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.

10.5.2.5 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Generator_System & *gs*, Recycle_Input *dummy*) `[inline]`

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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

std::invalid_argument Thrown if the system of generators is not empty but has no points, or if it contains closure points.

10.5.2.6 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.

10.5.2.7 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (Congruence_System & *cgs*, Recycle_Input *dummy*)

Builds a 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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

10.5.2.8 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const NNC_Polyhedron & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [explicit]

Builds a C polyhedron representing the topological closure of the NNC polyhedron `y`.

Parameters

y The NNC polyhedron to be used;

complexity This argument is ignored.

10.5.2.9 template<typename Interval > Parma_Polyhedra_Library::C_Polyhedron::C_-Polyhedron (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a C polyhedron out of a box.

The polyhedron inherits the space dimension of the box and is the most precise that includes the box. The algorithm used has polynomial complexity.

Parameters

box The box representing the polyhedron to be approximated;

complexity This argument is ignored.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.5.2.10 template<typename U > Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const BD_Shape< U > & *bd*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a C polyhedron out of a BD shape.

The polyhedron inherits the space dimension of the BDS and is the most precise that includes the BDS.

Parameters

bd The BDS used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.5.2.11 template<typename U > Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Octagonal_Shape< U > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a C polyhedron out of an octagonal shape.

The polyhedron inherits the space dimension of the octagonal shape and is the most precise that includes the octagonal shape.

Parameters

os The octagonal shape used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.5.2.12 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const Grid & *grid*, Complexity_Class *complexity* = ANY_COMPLEXITY) [explicit]

Builds a C polyhedron out of a grid.

The polyhedron inherits the space dimension of the grid and is the most precise that includes the grid.

Parameters

grid The grid used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.5.2.13 Parma_Polyhedra_Library::C_Polyhedron::C_Polyhedron (const C_Polyhedron & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Ordinary copy constructor.

The complexity argument is ignored.

10.5.3 Member Function Documentation

10.5.3.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.6 Parma_Polyhedra_Library::Checked_Number< T, Policy > Class Template Reference

A wrapper for numeric types implementing a given policy.

`#include <ppl.hh>`

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 (char y, Rounding_Dir dir)

  Direct initialization from a plain char and rounding mode.

- Checked_Number (signed char y, Rounding_Dir dir)

  Direct initialization from a signed char and rounding mode.

- Checked_Number (signed short y, Rounding_Dir dir)
  Direct initialization from a signed short and rounding mode.
- Checked_Number (signed int y, Rounding_Dir dir)
  Direct initialization from a signed int and rounding mode.
- Checked_Number (signed long y, Rounding_Dir dir)
  Direct initialization from a signed long and rounding mode.
- Checked_Number (signed long long y, Rounding_Dir dir)
  Direct initialization from a signed long long and rounding mode.
- Checked_Number (unsigned char y, Rounding_Dir dir)
  Direct initialization from an unsigned char and rounding mode.
- Checked_Number (unsigned short y, Rounding_Dir dir)
  Direct initialization from an unsigned short and rounding mode.
- Checked_Number (unsigned int y, Rounding_Dir dir)
  Direct initialization from an unsigned int and rounding mode.
- Checked_Number (unsigned long y, Rounding_Dir dir)
  Direct initialization from an unsigned long and rounding mode.
- Checked_Number (unsigned long long y, Rounding_Dir dir)
  Direct initialization from an unsigned long long and rounding mode.
- Checked_Number (float y, Rounding_Dir dir)
  Direct initialization from a float and rounding mode.
- Checked_Number (double y, Rounding_Dir dir)
  Direct initialization from a double and rounding mode.
- Checked_Number (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.
- template< typename From >
  Checked_Number (const From &, Rounding_Dir dir, typename Enable_If< Is_Special< From >::value, bool >::type ignored=false)
  Direct initialization from special 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 (char y)

  Direct initialization from a plain char, default rounding mode.

- Checked_Number (signed char y)

  Direct initialization from a signed char, default rounding mode.

- Checked_Number (signed short y)

  Direct initialization from a signed short, default rounding mode.

- Checked_Number (signed int y)

  Direct initialization from a signed int, default rounding mode.

- Checked_Number (signed long y)

  Direct initialization from a signed long, default rounding mode.

- Checked_Number (signed long long y)

  Direct initialization from a signed long long, default rounding mode.

- Checked_Number (unsigned char y)

  Direct initialization from an unsigned char, default rounding mode.

- Checked_Number (unsigned short y)

  Direct initialization from an unsigned short, default rounding mode.

- Checked_Number (unsigned int y)

  Direct initialization from an unsigned int, default rounding mode.

- Checked_Number (unsigned long y)

  Direct initialization from an unsigned long, default rounding mode.

- Checked_Number (unsigned long long y)

  Direct initialization from an unsigned long long, default rounding mode.

- Checked_Number (float y)

  Direct initialization from a float, default rounding mode.

- Checked_Number (double y)

  Direct initialization from a double, default rounding mode.

- Checked_Number (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.

- template<typename From >
  Checked_Number (const From &, typename Enable_If< Is_Special< From >::value, bool >::type ignored=false)

  Direct initialization from special, 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 >
  Checked_Number & operator= (const From &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 >
  Enable_If< Is_Native_Or_Checked< From >::value, Checked_Number< T, Policy > & >::type operator+= (const From &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 >
  Enable_If< Is_Native_Or_Checked< From >::value, Checked_Number< T, Policy > & >::type 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 >
  Enable_If< Is_Native_Or_Checked< From >::value, Checked_Number< T, Policy > & >::type 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 >
  Enable_If< Is_Native_Or_Checked< From >::value, Checked_Number< T, Policy > & >::type 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 >
  Enable_If< Is_Native_Or_Checked< From >::value, Checked_Number< T, Policy > & >::type 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.)

- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_not_a_number (const T &x)
- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_minus_infinity (const T &x)
- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_plus_infinity (const T &x)
- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, int >::type is_infinity (const T &x)
- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_integer (const T &x)
- template<typename To , typename From >
  Enable_If< Is_Native_Or_Checked< To >::value &&Is_Special< From >::value, Result >::type construct (To &to, const From &x, Rounding_Dir dir)
- template<typename To , typename From >
  Enable_If< Is_Native_Or_Checked< To >::value &&Is_Special< From >::value, Result >::type assign_r (To &to, const From &x, Rounding_Dir dir)
- template<typename To >
  Enable_If< Is_Native_Or_Checked< To >::value, Result >::type assign_r (To &to, const char *x, Rounding_Dir dir)
- template<typename To , typename To_Policy >
  Enable_If< Is_Native_Or_Checked< To >::value, Result >::type assign_r (To &to, char *x, Rounding_Dir dir)
- template<typename T , typename Policy >
  void swap (Checked_Number< T, Policy > &x, Checked_Number< T, Policy > &y)

  Swaps x with y.

- template<typename T , typename Policy >
  const T & raw_value (const Checked_Number< T, Policy > &x)
- template<typename T , typename Policy >
  T & raw_value (Checked_Number< T, Policy > &x)

Memory Size Inspection Functions

- template<typename T , typename Policy >
  size_t total_memory_in_bytes (const Checked_Number< T, Policy > &x)

  Returns the total size in bytes of the memory occupied by x.

- template<typename T , typename Policy >
  memory_size_type external_memory_in_bytes (const Checked_Number< T, Policy > &x)

  Returns the size in bytes of the memory managed by x.

Arithmetic Operators

- template<typename T , typename Policy >
  Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > &x)

  Unary plus operator.

- template<typename T , typename Policy >
  Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > &x)

  Unary minus operator.

- template<typename T , typename Policy >
  void floor_assign (Checked_Number< T, Policy > &x)
  Assigns to x largest integral value not greater than x.

- template<typename T , typename Policy >
  void floor_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)
  Assigns to x largest integral value not greater than y.

- template<typename T , typename Policy >
  void ceil_assign (Checked_Number< T, Policy > &x)
  Assigns to x smallest integral value not less than x.

- template<typename T , typename Policy >
  void ceil_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)
  Assigns to x smallest integral value not less than y.

- template<typename T , typename Policy >
  void trunc_assign (Checked_Number< T, Policy > &x)
  Round x to the nearest integer not larger in absolute value.

- template<typename T , typename Policy >
  void trunc_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)
  Assigns to x the value of y rounded to the nearest integer not larger in absolute value.

- template<typename T , typename Policy >
  void neg_assign (Checked_Number< T, Policy > &x)
  Assigns to x its negation.

- template<typename T , typename Policy >
  void neg_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)
  Assigns to x the negation of y.

- template<typename T , typename Policy >
  void abs_assign (Checked_Number< T, Policy > &x)
  Assigns to x its absolute value.

- template<typename T , typename Policy >
  void abs_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y)
  Assigns to x the absolute value of y.

- template<typename T , typename Policy >
  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.*

- template<typename T , typename Policy >
  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.*

- template<typename T , typename Policy >
  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*.*

- template<typename T , typename Policy >
  void gcdext_assign (Checked_Number< T, Policy > &x, Checked_Number< T, Policy > &s, Checked_Number< T, Policy > &t, const Checked_Number< T, Policy > &y, const Checked_Number< T, Policy > &z)

  Assigns to x *the greatest common divisor of* y *and* z*, setting* s *and* t *such that* $s*y + t*z = x = gcd(y, z)$*.*

- template<typename T , typename Policy >
  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*.*

- template<typename T , typename Policy >
  void mul_2exp_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, unsigned int exp)

  Assigns to x *the value* $y \cdot 2^{\mathrm{exp}}$*.*

- template<typename T , typename Policy >
  void div_2exp_assign (Checked_Number< T, Policy > &x, const Checked_Number< T, Policy > &y, unsigned int exp)

  Assigns to x *the value* $y/2^{\mathrm{exp}}$*.*

- 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)

  If z *divides* y*, assigns to* x *the quotient of the integer division of* y *and* z*.*

- template<typename T , typename Policy >
  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

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator== (const T1 &x, const T2 &y)

  Equality operator.

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator!= (const T1 &x, const T2 &y)

  Disequality operator.

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator>= (const T1 &x, const T2 &y)

Greater than or equal to operator.

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator> (const T1 &x, const T2 &y)

  Greater than operator.

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator<= (const T1 &x, const T2 &y)

  Less than or equal to operator.

- template<typename T1 , typename T2 >
  Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator< (const T1 &x, const T2 &y)

  Less than operator.

- template<typename From >
  Enable_If< Is_Native_Or_Checked< From >::value, int >::type sgn (const From &x)

  Returns -1, 0 *or* 1 *depending on whether the value of* x *is negative, zero or positive, respectively.*

- template<typename From1 , typename From2 >
  Enable_If< Is_Native_Or_Checked< From1 >::value &&Is_Native_Or_Checked< From2 >::value, int >::type cmp (const From1 &x, const From2 &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

- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, Result >::type output (std::ostream &os, const T &x, const Numeric_Format &fmt, Rounding_Dir dir)
- template<typename T , typename Policy >
  std::ostream & operator<< (std::ostream &os, const Checked_Number< T, Policy > &x)

  Output operator.

- template<typename T >
  Enable_If< Is_Native_Or_Checked< T >::value, Result >::type input (T &x, std::istream &is, Rounding_Dir dir)

  Input function.

- template<typename T , typename Policy >
  std::istream & operator>> (std::istream &is, Checked_Number< T, Policy > &x)

  Input operator.

10.6.1 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.

10.6.2 Member Function Documentation

10.6.2.1 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`.

10.6.3 Friends And Related Function Documentation

10.6.3.1 template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_not_a_number (const T & *x*) `[related]`

10.6.3.2 template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_minus_infinity (const T & *x*) `[related]`

10.6.3.3 template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_plus_infinity (const T & *x*) `[related]`

10.6.3.4 template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, int >::type is_infinity (const T & *x*) `[related]`

10.6.3.5 template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_integer (const T & *x*) `[related]`

10.6.3.6 template<typename To , typename From > Enable_If< Is_Native_Or_Checked< To >::value &&Is_Special< From >::value, Result >::type construct (To & *to*, const From & *x*, Rounding_Dir *dir*) `[related]`

10.6.3.7 template<typename To , typename From > Enable_If< Is_Native_Or_Checked< To >::value &&Is_Special< From >::value, Result >::type assign_r (To & *to*, const From & *x*, Rounding_Dir *dir*) `[related]`

10.6.3.8 template<typename To > Enable_If< Is_Native_Or_Checked< To >::value, Result >::type assign_r (To & *to*, const char ∗ *x*, Rounding_Dir *dir*) `[related]`

10.6.3.9 template<typename To , typename To_Policy > Enable_If< Is_Native_Or_Checked< To >::value, Result >::type assign_r (To & *to*, char ∗ *x*, Rounding_Dir *dir*) `[related]`

10.6.3.10 template<typename T , typename Policy > memory_size_type total_memory_in_bytes (const Checked_Number< T, Policy > & *x*) `[related]`

Returns the total size in bytes of the memory occupied by `x`.

10.6.3.11 template<typename T , typename Policy > memory_size_type external_memory_in_bytes (const Checked_Number< T, Policy > & *x*) `[related]`

Returns the size in bytes of the memory managed by `x`.

10.6.3.12 template<typename T , typename Policy > Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > & *x*) `[related]`

Unary plus operator.

10.6.3.13 template<typename T , typename Policy > Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > & *x*) [related]

Unary minus operator.

10.6.3.14 template<typename T , typename Policy > void floor_assign (Checked_Number< T, Policy > & *x*) [related]

Assigns to x largest integral value not greater than x.

10.6.3.15 template<typename T , typename Policy > void floor_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) [related]

Assigns to x largest integral value not greater than y.

10.6.3.16 template<typename T , typename Policy > void ceil_assign (Checked_Number< T, Policy > & *x*) [related]

Assigns to x smallest integral value not less than x.

10.6.3.17 template<typename T , typename Policy > void ceil_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) [related]

Assigns to x smallest integral value not less than y.

10.6.3.18 template<typename T , typename Policy > void trunc_assign (Checked_Number< T, Policy > & *x*) [related]

Round x to the nearest integer not larger in absolute value.

10.6.3.19 template<typename T , typename Policy > void trunc_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) [related]

Assigns to x the value of y rounded to the nearest integer not larger in absolute value.

10.6.3.20 template<typename T , typename Policy > void neg_assign (Checked_Number< T, Policy > & *x*) [related]

Assigns to x its negation.

10.6.3.21 template<typename T , typename Policy > void neg_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) [related]

Assigns to x the negation of y.

10.6.3.22 template<typename T , typename Policy > void abs_assign (Checked_Number< T, Policy > & *x*) [related]

Assigns to x its absolute value.

10.6.3.23 template<typename T , typename Policy > void abs_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) [related]

Assigns to x the absolute value of y.

10.6.3.24 template<typename T , typename Policy > void add_mul_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) [related]

Assigns to x the value `x + y * z`.

10.6.3.25 template<typename T , typename Policy > void sub_mul_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) [related]

Assigns to x the value `x - y * z`.

10.6.3.26 template<typename T , typename Policy > void gcd_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) [related]

Assigns to x the greatest common divisor of y and z.

10.6.3.27 template<typename T , typename Policy > void gcdext_assign (Checked_Number< T, Policy > & *x*, Checked_Number< T, Policy > & *s*, Checked_Number< T, Policy > & *t*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) `[related]`

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).

10.6.3.28 template<typename T , typename Policy > void lcm_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, const Checked_Number< T, Policy > & *z*) `[related]`

Assigns to `x` the least common multiple of `y` and `z`.

10.6.3.29 template<typename T , typename Policy > void mul_2exp_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, unsigned int *exp*) `[related]`

Assigns to `x` the value $y \cdot 2^{\mathtt{exp}}$.

10.6.3.30 template<typename T , typename Policy > void div_2exp_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*, unsigned int *exp*) `[related]`

Assigns to `x` the value $y/2^{\mathtt{exp}}$.

10.6.3.31 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`.

10.6.3.32 template<typename T , typename Policy > void sqrt_assign (Checked_Number< T, Policy > & *x*, const Checked_Number< T, Policy > & *y*) `[related]`

Assigns to `x` the integer square root of `y`.

10.6.3.33 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator== (const T1 & *x*, const T2 & *y*) [related]

Equality operator.

10.6.3.34 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator!= (const T1 & *x*, const T2 & *y*) [related]

Disequality operator.

10.6.3.35 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator>= (const T1 & *x*, const T2 & *y*) [related]

Greater than or equal to operator.

10.6.3.36 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator> (const T1 & *x*, const T2 & *y*) [related]

Greater than operator.

10.6.3.37 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator<= (const T1 & *x*, const T2 & *y*) [related]

Less than or equal to operator.

10.6.3.38 template<typename T1 , typename T2 > Enable_If< Is_Native_Or_Checked< T1 >::value &&Is_Native_Or_Checked< T2 >::value &&(Is_Checked< T1 >::value||Is_Checked< T2 >::value), bool >::type operator< (const T1 & *x*, const T2 & *y*) [related]

Less than operator.

10.6.3.39 template< typename From > Enable_If< Is_Native_Or_Checked< From >::value, int >::type sgn (const From & *x*) [related]

Returns -1, 0 or 1 depending on whether the value of x is negative, zero or positive, respectively.

10.6.3.40 template< typename From1 , typename From2 > Enable_If< Is_Native_Or_Checked< From1 >::value &&Is_Native_Or_Checked< From2 >::value, int >::type cmp (const From1 & *x*, const From2 & *y*) [related]

Returns a negative, zero or positive value depending on whether x is lower than, equal to or greater than y, respectively.

10.6.3.41 template< typename T > Enable_If< Is_Native_Or_Checked< T >::value, Result >::type output (std::ostream & *os*, const T & *x*, const Numeric_Format & *fmt*, Rounding_Dir *dir*) [related]

10.6.3.42 template< typename T , typename Policy > std::ostream & operator<< (std::ostream & *os*, const Checked_Number< T, Policy > & *x*) [related]

Output operator.

10.6.3.43 template< typename T > Enable_If< Is_Native_Or_Checked< T >::value, Result >::type input (T & *x*, std::istream & *is*, 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^^`;
- the C-style binary exponent indicator `p` can only be used when base 16 has been specified; if used, the exponent will be applied to base 2 (instead of base 16, as is the case when the indicator `e` is used);
- 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
        ;
```

```
INF     : 'inf'
        ;

NAN     : 'nan'
        ;

SIGN    : '-'
        | '+'
        ;

EXP     : 'e'
        | 'p'
        | '*^'
        ;
```

```
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
         ;
```

```
POINT   : '.'
        ;

DIV     : '/'
        ;

MINUS   : '-'
        ;

PLUS    : '+'
;

HEX     : '0x'
;

BASE    : '^^'
        ;

DIGIT   : '0' .. '9'
        ;

BDIGIT  : '0' .. '9'
        | 'a' .. 'z'
        ;
```

10.6.3.44 template<typename T , typename Policy > std::istream & operator>> (std::istream & *is*, Checked_Number< T, Policy > & *x*) `[related]`

Input operator.

10.6.3.45 template<typename T , typename Policy > void swap (Checked_Number< T, Policy > & *x*, Checked_Number< T, Policy > & *y*) `[related]`

Swaps `x` with `y`.

10.6.3.46 template<typename T , typename Policy > const T & raw_value (const Checked_Number< T, Policy > & *x*) `[related]`

10.6.3.47 template<typename T , typename Policy > T & raw_value (Checked_Number< T, Policy > & *x*) `[related]`

The documentation for this class was generated from the following file:

- ppl.hh

10.7 Parma_Polyhedra_Library::Variable::Compare Struct Reference

Binary predicate defining the total ordering on variables.

`#include <ppl.hh>`

Public Member Functions

- bool operator() (Variable x, Variable y) const

  Returns `true` *if and only if* `x` *comes before* `y`*.*

10.7.1 Detailed Description

Binary predicate defining the total ordering on variables.

The documentation for this struct was generated from the following file:

- ppl.hh

10.8 Parma_Polyhedra_Library::BHRZ03_Certificate::Compare Struct Reference

A total ordering on BHRZ03 certificates.

`#include <ppl.hh>`

Public Member Functions

- bool operator() (const BHRZ03_Certificate &x, const BHRZ03_Certificate &y) const

  Returns `true` *if and only if* `x` *comes before* `y`*.*

10.8.1 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.

The documentation for this struct was generated from the following file:

- ppl.hh

10.9 Parma_Polyhedra_Library::H79_Certificate::Compare Struct Reference

A total ordering on H79 certificates.

`#include <ppl.hh>`

Public Member Functions

- bool operator() (const H79_Certificate &x, const H79_Certificate &y) const

  Returns `true` *if and only if* `x` *comes before* `y`*.*

10.9.1 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.

The documentation for this struct was generated from the following file:

- ppl.hh

10.10 Parma_Polyhedra_Library::Grid_Certificate::Compare Struct Reference

A total ordering on Grid certificates.

`#include <ppl.hh>`

Public Member Functions

- bool operator() (const Grid_Certificate &x, const Grid_Certificate &y) const

    Returns `true` *if and only if* `x` *comes before* `y`*.*

10.10.1 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.

The documentation for this struct was generated from the following file:

- ppl.hh

10.11 Parma_Polyhedra_Library::Congruence Class Reference

A linear congruence.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Row.

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/= (Coefficient_traits::const_reference k)
  Multiplies `k` *into the modulus of* `*this`.
- bool is_tautological () const
  Returns `true` *if and only if* `*this` *is a tautology (i.e., an always true congruence).*
- bool is_inconsistent () const
  Returns `true` *if and only if* `*this` *is inconsistent (i.e., an always 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`.
- void ascii_dump () const
  Writes to `std::cerr` *an ASCII representation of* `*this`.
- void ascii_dump (std::ostream &s) const
  Writes to `s` *an ASCII representation of* `*this`.
- void print () const
  Prints `*this` *to* `std::cerr` *using* `operator<<`.
- bool ascii_load (std::istream &s)
  Loads from `s` *an ASCII representation of the internal representation of* `*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 void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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 $0 = 1 \pmod{0}$.

- static Congruence create (const Linear_Expression &e1, const Linear_Expression &e2)

  Returns the congruence $e1 = e2 \pmod{1}$.

- static Congruence create (const Linear_Expression &e, Coefficient_traits::const_reference n)

  Returns the congruence $e = n \pmod{1}$.

- static Congruence create (Coefficient_traits::const_reference n, const Linear_Expression &e)

  Returns the congruence $n = e \pmod{1}$.

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

- Congruence operator/ (const Congruence &cg, Coefficient_traits::const_reference k)

  Returns a copy of `cg`*, multiplying* `k` *into the copy's modulus.*

- Congruence operator/ (const Constraint &c, Coefficient_traits::const_reference m)

  Creates a congruence from `c`*, with* `m` *as the modulus.*

- 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.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Congruence &c)

  Output operators.

- 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 $e = n \pmod{1}$.

- void swap (Parma_Polyhedra_Library::Congruence &x, Parma_Polyhedra_Library::Congruence &y)

  Specializes `std::swap`.

10.11.1 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;
const Coefficient& m = cg1.modulus();
if (m == 0)
  cout << "Congruence cg1 is an equality." << endl;
else {
  Linear_Expression e;
  for (dimension_type i = cg1.space_dimension(); i-- > 0; )
    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.

10.11.2 Constructor & Destructor Documentation

10.11.2.1 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 an inequality.

10.11.3 Member Function Documentation

10.11.3.1 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`.

10.11.3.2 Congruence & Parma_Polyhedra_Library::Congruence::operator/= (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 $e_1 = e_2 \pmod{mk}$.

10.11.3.3 bool Parma_Polyhedra_Library::Congruence::is_tautological () const

Returns `true` if and only if `*this` is a tautology (i.e., an always true congruence).

A tautological congruence has 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 $b = 0 \pmod{m}$.

10.11.3.4 bool Parma_Polyhedra_Library::Congruence::is_inconsistent () const

Returns `true` if and only if `*this` is inconsistent (i.e., an always false congruence).

An inconsistent congruence has one of the following two forms:

- an equality: $\sum_{i=0}^{n-1} 0x_i + b == 0$ where $b \neq 0$; or
- a proper congruence: $\sum_{i=0}^{n-1} 0x_i + b\% = 0/m$, where $b \neq 0 \pmod{m}$.

10.11.3.5 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.

10.11.3.6 bool Parma_Polyhedra_Library::Congruence::is_equality () const [inline]

Returns `true` if `*this` is an equality.

A modulus of zero denotes a linear equality.

10.11.3.7 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.

10.11.3.8 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.

10.11.3.9 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.

10.11.4 Friends And Related Function Documentation

10.11.4.1 Congruence operator/ (const Congruence & *cg*, Coefficient_traits::const_reference *k*) [friend]

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 $e_1 = e_2 \pmod{mk}$.

10.11.4.2 Congruence operator/ (const Constraint & *c*, Coefficient_traits::const_reference *m*) [friend]

Creates a congruence from `c`, with `m` as the modulus.

10.11.4.3 bool operator== (const Congruence & *x*, const Congruence & *y*) `[friend]`

Returns `true` if and only if `x` and `y` are equivalent.

10.11.4.4 bool operator!= (const Congruence & *x*, const Congruence & *y*) `[friend]`

Returns `false` if and only if `x` and `y` are equivalent.

10.11.4.5 std::ostream & operator<< (std::ostream & *s*, const Congruence & *c*) `[related]`

Output operators.

10.11.4.6 Congruence operator%= (const Linear_Expression & *e1*, const Linear_Expression & *e2*) `[related]`

Returns the congruence $e1 = e2 \pmod{1}$.

10.11.4.7 Congruence operator%= (const Linear_Expression & *e*, Coefficient_traits::const_-reference *n*) `[related]`

Returns the congruence $e = n \pmod{1}$.

10.11.4.8 void swap (Parma_Polyhedra_Library::Congruence & *x*, Parma_Polyhedra_Library::Congruence & *y*) `[related]`

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.12 Parma_Polyhedra_Library::Congruence_System Class Reference

A system of congruences.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Matrix.

Classes

- class const_iterator

  An iterator over 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.

- bool empty () const

  Returns `true` if and only if `*this` has no congruences.

- 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.

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- 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.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Congruence_System can handle.

- static void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Congruence_System &cgs)

  Output operator.

- void swap (Parma_Polyhedra_Library::Congruence_System &x, Parma_Polyhedra_-Library::Congruence_System &y)

  Specializes `std::swap`.

10.12.1 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.

10.12.2 Constructor & Destructor Documentation

10.12.2.1 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.

10.12.3 Member Function Documentation

10.12.3.1 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.

10.12.3.2 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 relational constraint.

10.12.3.3 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.

10.12.3.4 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.

10.12.4 Friends And Related Function Documentation

10.12.4.1 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 ", ".

10.12.4.2 void swap (Parma_Polyhedra_Library::Congruence_System & *x*, Parma_Polyhedra_Library::Congruence_System & *y*) `[related]`

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.13 Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 > Class Template Reference

This class provides the reduction method for the Congruences_Product domain.

`#include <ppl.hh>`

Public Member Functions

- Congruences_Reduction ()

  Default constructor.

- void product_reduce (D1 &d1, D2 &d2)

  The congruences reduction operator for detect emptiness or any equalities implied by each of the congruences defining one of the components and the bounds of the other component. It is assumed that the components are already constraints reduced.

- ~Congruences_Reduction ()

  Destructor.

10.13.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::Congruences_-Reduction< D1, D2 >

This class provides the reduction method for the Congruences_Product domain. The reduction classes are used to instantiate the Partially_Reduced_Product domain.

This class uses the minimized congruences defining each of the components. For each of the congruences, it checks if the other component intersects none, one or more than one hyperplane defined by the congruence and adds equalities or emptiness as appropriate; in more detail: Letting the components be d1 and d2, then, for each congruence cg representing d1:

- if more than one hyperplane defined by cg intersects d2, then d1 and d2 are unchanged;
- if exactly one hyperplane intersects d2, then d1 and d2 are refined with the corresponding equality ;
- otherwise, d1 and d2 are set to empty. Unless d1 and d2 are already empty, the process is repeated where the roles of d1 and d2 are reversed. If d1 or d2 is empty, then the emptiness is propagated.

10.13.2 Member Function Documentation

10.13.2.1 template<typename D1 , typename D2 > void Parma_Polyhedra_-Library::Congruences_Reduction< D1, D2 >::product_reduce (D1 & *d1*, D2 & *d2*) `[inline]`

The congruences reduction operator for detect emptiness or any equalities implied by each of the congruences defining one of the components and the bounds of the other component. It is assumed that the components are already constraints reduced.

The minimized congruence system defining the domain element `d1` is used to check if `d2` intersects none, one or more than one of the hyperplanes defined by the congruences: if it intersects none, then product is set empty; if it intersects one, then the equality defining this hyperplane is added to both components; otherwise, the product is unchanged. In each case, the donor domain must provide a congruence system in minimal form.

Parameters

d1 A pointset domain element;

d2 A pointset domain element;

The documentation for this class was generated from the following file:

- ppl.hh

10.14 Parma_Polyhedra_Library::Constraint_System::const_iterator Class Reference

An iterator over a system of constraints.

`#include <ppl.hh>`

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.*

10.14.1 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;
```

The documentation for this class was generated from the following file:

- ppl.hh

10.15 Parma_Polyhedra_Library::Generator_System::const_iterator Class Reference

An iterator over a system of generators.

`#include <ppl.hh>`

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-> () 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.*

10.15.1 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"));
```

The documentation for this class was generated from the following file:

- ppl.hh

10.16 Parma_Polyhedra_Library::Congruence_System::const_iterator Class Reference

An iterator over a system of congruences.

`#include <ppl.hh>`

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.

10.16.1 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;
```

The documentation for this class was generated from the following file:

- ppl.hh

10.17 Parma_Polyhedra_Library::Grid_Generator_System::const_iterator Class Reference

An iterator over a system of grid generators.

`#include <ppl.hh>`

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.

10.17.1 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& ggs = gr.generators();
for (Grid_Generator_System::const_iterator i = ggs.begin(),
      ggs_end = ggs.end(); i != ggs_end; ++i)
  cout << *i << endl;
```

The same effect can be obtained more concisely by using more features of the STL:

```
const Grid_Generator_System& ggs = gr.generators();
copy(ggs.begin(), ggs.end(), ostream_iterator<Grid_Generator>(cout, "\n"));
```

The documentation for this class was generated from the following file:

- ppl.hh

10.18 Parma_Polyhedra_Library::Constraint Class Reference

A linear equality or inequality.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_Row.

Public Types

- enum Type { EQUALITY, NONSTRICT_INEQUALITY, STRICT_INEQUALITY }

  The constraint type.

Public Member Functions

- Constraint (const Constraint &c)

  Ordinary copy constructor.

- Constraint (const Congruence &cg)

  Copy-constructs from equality congruence `cg`*.*

- ~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.*

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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== (Variable v1, Variable v2)

  Returns the constraint `v1 = v2`.

- 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>= (Variable v1, Variable v2)

  Returns the constraint `v1 >= v2`.

- 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> (Variable v1, Variable v2)

  Returns the constraint `v1 > v2`.

- 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`.

- Constraint operator<= (Variable v1, Variable v2)

  Returns the constraint `v1` *<=* `v2`.

- Constraint operator< (Variable v1, Variable v2)

  Returns the constraint `v1` *<* `v2`.

- 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.

10.18.1 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 (dimension_type i = c1.space_dimension(); i-- > 0; )
    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.

10.18.2 Member Enumeration Documentation

10.18.2.1 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.

10.18.3 Constructor & Destructor Documentation

10.18.3.1 Parma_Polyhedra_Library::Constraint::Constraint (const Congruence & *cg*) `[explicit]`

Copy-constructs from equality congruence `cg`.

Exceptions

std::invalid_argument Thrown if `cg` is a proper congruence.

10.18.4 Member Function Documentation

10.18.4.1 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`.

10.18.4.2 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 $b > 0$.

10.18.4.3 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 $b \leq 0$.

10.18.4.4 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.

10.18.5 Friends And Related Function Documentation

10.18.5.1 Constraint operator== (const Linear_Expression & *e1*, const Linear_Expression & *e2*) `[friend]`

Returns the constraint `e1` = `e2`.

10.18.5.2 Constraint operator== (Variable *v1*, Variable *v2*) `[friend]`

Returns the constraint `v1` = `v2`.

10.18.5.3 Constraint operator== (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) `[friend]`

Returns the constraint `e` = `n`.

10.18.5.4 Constraint operator== (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) `[friend]`

Returns the constraint `n` = `e`.

10.18.5.5 Constraint operator>= (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the constraint e1 >= e2.

10.18.5.6 Constraint operator>= (Variable *v1*, Variable *v2*) [friend]

Returns the constraint v1 >= v2.

10.18.5.7 Constraint operator>= (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the constraint e >= n.

10.18.5.8 Constraint operator>= (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the constraint n >= e.

10.18.5.9 Constraint operator<= (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the constraint e1 <= e2.

10.18.5.10 Constraint operator<= (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the constraint e <= n.

10.18.5.11 Constraint operator<= (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the constraint n <= e.

10.18.5.12 Constraint operator> (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the constraint `e1` $>$ `e2`.

10.18.5.13 Constraint operator> (Variable *v1*, Variable *v2*) [friend]

Returns the constraint `v1` $>$ `v2`.

10.18.5.14 Constraint operator> (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the constraint `e` $>$ `n`.

10.18.5.15 Constraint operator> (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the constraint `n` $>$ `e`.

10.18.5.16 Constraint operator< (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the constraint `e1` $<$ `e2`.

10.18.5.17 Constraint operator< (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the constraint `e` $<$ `n`.

10.18.5.18 Constraint operator< (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the constraint `n` $<$ `e`.

10.18.5.19 bool operator== (const Constraint & *x*, const Constraint & *y*) [related]

Returns `true` if and only if `x` is equivalent to `y`.

10.18.5.20 bool operator!= (const Constraint & *x*, const Constraint & *y*) [related]

Returns `true` if and only if `x` is not equivalent to `y`.

10.18.5.21 Constraint operator<= (Variable *v1*, Variable *v2*) [related]

Returns the constraint `v1` <= `v2`.

10.18.5.22 Constraint operator< (Variable *v1*, Variable *v2*) [related]

Returns the constraint `v1` < `v2`.

10.18.5.23 void swap (Parma_Polyhedra_Library::Constraint & *x*, Parma_Polyhedra_Library::Constraint & *y*) [related]

Specializes `std::swap`.

10.18.5.24 std::ostream & operator<< (std::ostream & *s*, const Constraint & *c*) [related]

Output operator.

10.18.5.25 std::ostream & operator<< (std::ostream & *s*, const Constraint::Type & *t*) [related]

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.19 Parma_Polyhedra_Library::Constraint_System Class Reference

A system of constraints.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_System.

Classes

- class const_iterator

  An iterator over 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 Congruence_System &cgs)

  Builds a system containing copies of any equalities in `cgs`*.*

- 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_equalities () const

  Returns `true` *if and only if* `*this` *contains one or more equality constraints.*

- 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.*

- bool empty () const

  Returns `true` *if and only if* `*this` *has no constraints.*

- 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.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`*.*

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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`.

10.19.1 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.

10.19.2 Friends And Related Function Documentation

10.19.2.1 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.

10.19.2.2 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 ", ".

10.19.2.3 void swap (Parma_Polyhedra_Library::Constraint_System & *x*, Parma_Polyhedra_Library::Constraint_System & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.20 Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 > Class Template Reference

This class provides the reduction method for the Constraints_Product domain.

`#include <ppl.hh>`

Public Member Functions

- Constraints_Reduction ()

  Default constructor.

- void product_reduce (D1 &d1, D2 &d2)

  The constraints reduction operator for sharing constraints between the domains.

- ∼Constraints_Reduction ()

  Destructor.

10.20.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >

This class provides the reduction method for the Constraints_Product domain. The reduction classes are used to instantiate the Partially_Reduced_Product domain. This class adds the constraints defining each of the component domains to the other component.

10.20.2 Member Function Documentation

10.20.2.1 template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >::product_reduce (D1 & *d1*, D2 & *d2*) `[inline]`

The constraints reduction operator for sharing constraints between the domains.

The minimized constraint system defining the domain element `d1` is added to `d2` and the minimized constraint system defining `d2` is added to `d1`. In each case, the donor domain must provide a constraint system in minimal form; this must define a polyhedron in which the donor element is contained. The recipient domain selects a subset of these constraints that it can add to the recipient element. For example: if the domain `D1` is the Grid domain and `D2` the NNC Polyhedron domain, then only the equality constraints are copied from `d1` to `d2` and from `d2` to `d1`.

Parameters

d1 A pointset domain element;

d2 A pointset domain element;

The documentation for this class was generated from the following file:

- ppl.hh

10.21 Parma_Polyhedra_Library::Determinate< PSET > Class Template Reference

A wrapper for PPL pointsets, providing them with a *determinate constraint system* interface, as defined in [Bag98].

`#include <ppl.hh>`

Public Member Functions

Constructors and Destructor

- Determinate (const PSET &p)

  Constructs a COW-wrapped object corresponding to the pointset `p`.

- Determinate (const Constraint_System &cs)

  Constructs a COW-wrapped object corresponding to the pointset defined by `cs`.

- Determinate (const Congruence_System &cgs)

  Constructs a COW-wrapped object corresponding to the pointset defined by `cgs`*.*

- Determinate (const Determinate &y)

  Copy constructor.

- ∼Determinate ()

  Destructor.

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 weakening_assign (const Determinate &y)

  Assigns to `*this` *the result of weakening* `*this` *with* `y`*.*

- void concatenate_assign (const Determinate &y)

  Assigns to `*this` *the concatenation of* `*this` *and* `y`*, taken in this order.*

- PSET & pointset ()

  Returns a reference to the embedded element.

- void **mutate** ()
- Determinate & operator= (const Determinate &y)

  Assignment operator.

- void swap (Determinate &y)

  Swaps `*this` *with* `y`*.*

Friends

- bool operator== (const Determinate< PSET > &x, const Determinate< PSET > &y)

  Returns `true` *if and only if* `x` *and* `y` *are the same COW-wrapped pointset.*

- bool operator!= (const Determinate< PSET > &x, const Determinate< PSET > &y)

  Returns `true` *if and only if* `x` *and* `y` *are different COW-wrapped pointsets.*

Related Functions

(Note that these are not member functions.)

- template<typename PSET >
  std::ostream & operator<< (std::ostream &, const Determinate< PSET > &)

  Output operator.

- template<typename PSET >
void swap (Parma_Polyhedra_Library::Determinate< PSET > &x, Parma_Polyhedra_Library::Determinate< PSET > &y)

  Specializes `std::swap`.

Member Functions that Do Not Modify the Domain Element

- const PSET & pointset () const

  Returns a const reference to the embedded pointset.

- bool is_top () const

  Returns `true` if and only if `*this` embeds the universe element `PSET`.

- bool is_bottom () const

  Returns `true` if and only if `*this` embeds the empty element of `PSET`.

- 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 definitely 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.

- static bool has_nontrivial_weakening ()

10.21.1 Detailed Description

template<typename PSET> class Parma_Polyhedra_Library::Determinate< PSET >

A wrapper for PPL pointsets, providing them with a *determinate constraint system* interface, as defined in [Bag98]. The implementation uses a copy-on-write optimization, making the class suitable for constructions, like the *finite powerset* and *ask-and-tell* of [Bag98], that are likely to perform many copies.

10.21.2 Member Function Documentation

10.21.2.1 template<typename PSET > bool Parma_Polyhedra_Library::Determinate< PSET >::has_nontrivial_weakening () [inline, static]

Returns `true` if and only if this domain has a nontrivial weakening operator.

10.21.3 Friends And Related Function Documentation

10.21.3.1 template<typename PSET > bool operator== (const Determinate< PSET > & *x*, const Determinate< PSET > & *y*) [friend]

Returns `true` if and only if `x` and `y` are the same COW-wrapped pointset.

10.21.3.2 template<typename PSET > bool operator!= (const Determinate< PSET > & *x*, const Determinate< PSET > & *y*) [friend]

Returns `true` if and only if `x` and `y` are different COW-wrapped pointsets.

10.21.3.3 template<typename PSET > std::ostream & operator<< (std::ostream & *s*, const Determinate< PSET > & *x*) [related]

Output operator.

10.21.3.4 template<typename PSET > void swap (Parma_Polyhedra_Library::Determinate< PSET > & *x*, Parma_Polyhedra_Library::Determinate< PSET > & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.22 Parma_Polyhedra_Library::Domain_Product< D1, D2 > Class Template Reference

This class is temporary and will be removed when template typedefs will be supported in C++.

`#include <ppl.hh>`

10.22.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::Domain_Product< D1, D2 >

This class is temporary and will be removed when template typedefs will be supported in C++. When template typedefs will be supported in C++, what now is verbosely denoted by Domain_Product<Domain1, Domain2>::Direct_Product will simply be denoted by Direct_Product<Domain1, Domain2>.

The documentation for this class was generated from the following file:

- ppl.hh

10.23 Parma_Polyhedra_Library::Generator Class Reference

A line, ray, point or closure point.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_Row.

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.*
- void ascii_dump () const
  Writes to `std::cerr` *an ASCII representation of* `*this`.
- void ascii_dump (std::ostream &s) const
  Writes to `s` *an ASCII representation of* `*this`.
- void print () const
  Prints `*this` *to* `std::cerr` *using* `operator<<`.
- bool ascii_load (std::istream &s)
  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*
- 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)
  Returns the line of direction `e`.
- static Generator ray (const Linear_Expression &e)
  Returns the ray of direction `e`.
- static Generator point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_-traits::const_reference d=Coefficient_one())
  Returns the point at `e` / `d`.
- static Generator closure_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one())
  Returns the closure point at `e` / `d`.
- static dimension_type max_space_dimension ()
  Returns the maximum space dimension a Generator can handle.
- static void initialize ()
  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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 $\mathbb{R}^0$.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Generator &g)

  Output operator.

- void swap (Parma_Polyhedra_Library::Generator &x, Parma_Polyhedra_Library::Generator &y)

  Specializes `std::swap`.

- 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`.

- template<typename To >
  bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, Rounding_Dir dir)

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

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

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

- template<typename To >
  bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, Rounding_Dir dir)

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

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

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

- template<typename To >
  bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, Rounding_Dir dir)

Computes the L_∞ distance between x and y.

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

  Computes the L_∞ distance between x and y.

- std::ostream & operator<< (std::ostream &s, const Generator::Type &t)

  Output operator.

10.23.1 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 (dimension_type i = g1.space_dimension(); i-- > 0; )
    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.

10.23.2 Member Enumeration Documentation

10.23.2.1 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.

10.23.3 Member Function Documentation

10.23.3.1 Generator line (const Linear_Expression & *e*) `[inline, static]`

Returns the line of direction `e`.

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.

10.23.3.2 Generator ray (const Linear_Expression & *e*) `[inline, static]`

Returns the ray of direction `e`.

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.

10.23.3.3 Generator point (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Returns the point at `e` / `d`.

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.

10.23.3.4 Generator closure_point (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Returns the closure point at `e` / `d`.

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.

10.23.3.5 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.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator.

10.23.3.6 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.

10.23.3.7 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.

10.23.4 Friends And Related Function Documentation

10.23.4.1 std::ostream & operator<< (std::ostream & *s*, const Generator & *g*) [related]

Output operator.

10.23.4.2 void swap (Parma_Polyhedra_Library::Generator & *x*, Parma_Polyhedra_Library::Generator & *y*) [related]

Specializes std::swap.

10.23.4.3 bool operator== (const Generator & *x*, const Generator & *y*) [related]

Returns `true` if and only if `x` is equivalent to `y`.

10.23.4.4 bool operator!= (const Generator & *x*, const Generator & *y*) [related]

Returns `true` if and only if `x` is not equivalent to `y`.

10.23.4.5 template<typename To > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, Rounding_Dir *dir*) [related]

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

Computes the euclidean 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>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

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<Temp, Extended_Number_Policy>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

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<Temp, Extended_Number_Policy>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.6 template< typename Temp , typename To > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, 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`.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.7 template< typename To > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, 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>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.8 template< typename Temp , typename To > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, 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`.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.9 template<typename To > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, 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>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

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<Temp, Extended_Number_-Policy>.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.10 template<typename Temp , typename To > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, 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`.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

10.23.4.11 std::ostream & operator<< (std::ostream & s, const Generator::Type & t) [related]

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.24 Parma_Polyhedra_Library::Generator_System Class Reference

A system of generators.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_System.

Inherited by Parma_Polyhedra_Library::Grid_Generator_System [private].

Classes

- class const_iterator

  An iterator over a system of generators.

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.*

- bool empty () const

  Returns `true` *if and only if* `*this` *has no generators.*

- 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.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`*.*

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`*.*

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`*.*

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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`.

10.24.1 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

$$\{ (x, 0)^{\mathrm{T}} \in \mathbb{R}^2 \mid x \geq 0 \},$$

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.

10.24.2 Member Function Documentation

10.24.2.1 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.

10.24.2.2 bool Parma_Polyhedra_Library::Generator_System::ascii_load (std::istream & *s*)

Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

Resizes the matrix of generators using the numbers of rows and columns read from `s`, then initializes the coordinates of each generator and its type reading the contents from `s`.

Reimplemented in Parma_Polyhedra_Library::Grid_Generator_System.

10.24.3 Friends And Related Function Documentation

10.24.3.1 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 ", ".

10.24.3.2 void swap (Parma_Polyhedra_Library::Generator_System & *x*, Parma_Polyhedra_Library::Generator_System & *y*) `[related]`

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.25 Parma_Polyhedra_Library::GMP_Integer Class Reference

Unbounded integers as provided by the GMP library.

`#include <ppl.hh>`

Related Functions

(Note that these are not member functions.)

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 abs_assign (GMP_Integer &x)
  Assigns to x *its absolute value.*
- void abs_assign (GMP_Integer &x, const GMP_Integer &y)
  Assigns to x *the absolute value of* y*.*
- void rem_assign (GMP_Integer &x, const GMP_Integer &y, const GMP_Integer &z)
  Assigns to x *the remainder of the division of* y *by* z*.*
- 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, GMP_Integer &s, GMP_Integer &t, const GMP_Integer &y, const GMP_Integer &z)
  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 mul_2exp_assign (GMP_Integer &x, const GMP_Integer &y, unsigned int exp)
  Assigns to x *the value* $y \cdot 2^{\mathrm{exp}}$*.*
- void div_2exp_assign (GMP_Integer &x, const GMP_Integer &y, unsigned int exp)
  Assigns to x *the value* $y/2^{\mathrm{exp}}$*.*
- 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.*

10.25.1 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/`

10.25.2 Friends And Related Function Documentation

10.25.2.1 const mpz_class & raw_value (const GMP_Integer & *x*) [related]

Returns a const reference to the underlying integer value.

10.25.2.2 mpz_class & raw_value (GMP_Integer & *x*) [related]

Returns a reference to the underlying integer value.

10.25.2.3 memory_size_type total_memory_in_bytes (const GMP_Integer & *x*) [related]

Returns the total size in bytes of the memory occupied by `x`.

10.25.2.4 memory_size_type external_memory_in_bytes (const GMP_Integer & *x*) [related]

Returns the size in bytes of the memory managed by `x`.

10.25.2.5 void neg_assign (GMP_Integer & *x*) [related]

Assigns to `x` its negation.

10.25.2.6 void neg_assign (GMP_Integer & *x*, const GMP_Integer & *y*) [related]

Assigns to `x` the negation of `y`.

10.25.2.7 void abs_assign (GMP_Integer & *x*) [related]

Assigns to `x` its absolute value.

10.25.2.8 void abs_assign (GMP_Integer & *x*, const GMP_Integer & *y*) [related]

Assigns to `x` the absolute value of `y`.

10.25.2.9 void rem_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) [related]

Assigns to x the remainder of the division of y by z.

10.25.2.10 void gcd_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) [related]

Assigns to x the greatest common divisor of y and z.

10.25.2.11 void gcdext_assign (GMP_Integer & *x*, GMP_Integer & *s*, GMP_Integer & *t*, const GMP_Integer & *y*, const GMP_Integer & *z*) [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.

10.25.2.12 void lcm_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) [related]

Assigns to x the least common multiple of y and z.

10.25.2.13 void add_mul_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) [related]

Assigns to x the value x + y * z.

10.25.2.14 void sub_mul_assign (GMP_Integer & *x*, const GMP_Integer & *y*, const GMP_Integer & *z*) [related]

Assigns to x the value x - y * z.

10.25.2.15 void mul_2exp_assign (GMP_Integer & *x*, const GMP_Integer & *y*, unsigned int *exp*) [related]

Assigns to x the value $y \cdot 2^{\mathrm{exp}}$.

10.25.2.16 void div_2exp_assign (GMP_Integer & *x*, const GMP_Integer & *y*, unsigned int *exp*) [related]

Assigns to x the value $y/2^{\mathtt{exp}}$.

10.25.2.17 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.

10.25.2.18 void sqrt_assign (GMP_Integer & *x*, const GMP_Integer & *y*) [related]

Assigns to x the integer square root of y.

The documentation for this class was generated from the following file:

- ppl.hh

10.26 Parma_Polyhedra_Library::Grid Class Reference

A grid.

`#include <ppl.hh>`

Public Types

- typedef Coefficient coefficient_type

  The numeric type of coefficients.

Public Member Functions

- Grid (dimension_type num_dimensions=0, 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, Recycle_Input dummy)

  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, Recycle_Input dummy)

  Builds a grid, recycling a system of constraints.

- Grid (const Grid_Generator_System &const_gs)

  Builds a grid, copying a system of grid generators.

- Grid (Grid_Generator_System &gs, Recycle_Input dummy)

  Builds a grid, recycling a system of grid generators.

- template<typename Interval >
  Grid (const Box< Interval > &box, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a grid out of a box.

- template<typename U >
  Grid (const BD_Shape< U > &bd, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a grid out of a bounded-difference shape.

- template<typename U >
  Grid (const Octagonal_Shape< U > &os, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a grid out of an octagonal shape.

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

  Builds a grid from a polyhedron using algorithms whose complexity does not exceed the one specified by `complexity`*. If* `complexity` *is* `ANY_COMPLEXITY`*, then the grid built is the smallest one containing* `ph`*.*

- Grid (const Grid &y, Complexity_Class complexity=ANY_COMPLEXITY)

  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`*.*

- Constraint_System constraints () const

  Returns a system of equality constraints satisfied by `*this` *with the same affine dimension as* `*this`*.*

- Constraint_System minimized_constraints () const

  Returns a minimal system of equality constraints satisfied by `*this` *with the same affine dimension as* `*this`*.*

- const Congruence_System & congruences () const

  Returns the system of congruences.

- const Congruence_System & minimized_congruences () const

Returns the system of congruences in minimal form.

- const Grid_Generator_System & grid_generators () const
  Returns the system of generators.
- const Grid_Generator_System & minimized_grid_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`.
- Poly_Gen_Relation relation_with (const Generator &g) const
  Returns the relations holding between `*this` and `g`.
- Poly_Con_Relation relation_with (const Constraint &c) const
  Returns the relations holding between `*this` and `c`.
- 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 contains_integer_point () const
  Returns `true` if and only if `*this` contains at least one integer point.
- bool constrains (Variable var) const
  Returns `true` if and only if `var` is constrained in `*this`.
- 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, 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 frequency (const Linear_Expression &expr, Coefficient &freq_n, Coefficient &freq_d, Coefficient &val_n, Coefficient &val_d) const

  Returns `true` *if and only if* `*this` *is not empty and frequency for* `*this` *with respect to* `expr` *is defined, in which case the frequency and the value for* `expr` *that is closest to zero 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`*.*

- 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_grid_generator (const Grid_Generator &g)

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

- void add_congruences (const Congruence_System &cgs)

  Adds a copy of each congruence in `cgs` *to* `*this`*.*

- void add_recycled_congruences (Congruence_System &cgs)

  Adds the congruences in `cgs` *to *this.*

- void add_constraint (const Constraint &c)

  Adds to `*this` *a congruence equivalent to constraint* `c`*.*

- void add_constraints (const Constraint_System &cs)

  Adds to `*this` *congruences equivalent to the constraints in* `cs`*.*

- void add_recycled_constraints (Constraint_System &cs)

  Adds to `*this` *congruences equivalent to the constraints in* `cs`*.*

- void refine_with_congruence (const Congruence &cg)

Uses a copy of the congruence `cg` *to refine* `*this`*.*

- void refine_with_congruences (const Congruence_System &cgs)

  Uses a copy of the congruences in `cgs` *to refine* `*this`*.*

- void refine_with_constraint (const Constraint &c)

  Uses a copy of the constraint `c` *to refine* `*this`*.*

- void refine_with_constraints (const Constraint_System &cs)

  Uses a copy of the constraints in `cs` *to refine* `*this`*.*

- void add_grid_generators (const Grid_Generator_System &gs)

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

- void add_recycled_grid_generators (Grid_Generator_System &gs)

  Adds the generators in `gs` *to the system of generators of* `this`*.*

- void unconstrain (Variable var)

  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`*.*

- void unconstrain (const Variables_Set &vars)

  Computes the cylindrification of `*this` *with respect to the set of space dimensions* `vars`*, assigning the result to* `*this`*.*

- void intersection_assign (const Grid &y)

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

- void upper_bound_assign (const Grid &y)

  Assigns to `*this` *the least upper bound of* `*this` *and* `y`*.*

- bool upper_bound_assign_if_exact (const Grid &y)

  If the upper bound of `*this` *and* `y` *is exact it is assigned to* `this` *and* `true` *is returned, otherwise* `false` *is returned.*

- void difference_assign (const Grid &y)

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

- bool simplify_using_context_assign (const Grid &y)

  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`*. If* `false` *is returned, then the intersection is empty.*

- 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, Relation_Symbol relsym, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one(), Coefficient_traits::const_reference modulus=Coefficient_zero())

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, Relation_Symbol relsym, const Linear_Expression &expr, Coefficient_traits::const_reference denominator=Coefficient_one(), Coefficient_traits::const_reference modulus=Coefficient_zero())

  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, Relation_Symbol relsym, const Linear_Expression &rhs, Coefficient_traits::const_reference modulus=Coefficient_zero())

  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, Relation_Symbol relsym, const Linear_Expression &rhs, Coefficient_traits::const_reference modulus=Coefficient_zero())

  Assigns to `*this` *the preimage of* `*this` *with respect to the generalized affine relation* $\mathrm{lhs}' = \mathrm{rhs} \pmod{\mathrm{modulus}}$.

- 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 Grid &y)

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

- void wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

  Wraps the specified dimensions of the vector space.

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping all points with non-integer coordinates.*

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping all points with non-integer coordinates for the space dimensions corresponding to* `vars`.

- void topological_closure_assign ()

  Assigns to `*this` *its topological closure.*

- void congruence_widening_assign (const Grid &y, unsigned *tp=NULL)

  Assigns to `*this` *the result of computing the Grid widening between* `*this` *and* `y` *using congruence systems.*

- void generator_widening_assign (const Grid &y, unsigned *tp=NULL)

  Assigns to `*this` *the result of computing the Grid widening between* `*this` *and* `y` *using generator systems.*

- 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_congruence_extrapolation_assign (const Grid &y, const Congruence_System &cgs, unsigned *tp=NULL)

  Improves the result of the congruence variant of Grid widening computation by also enforcing those congruences in `cgs` *that are satisfied by all the points of* `*this`.

- void limited_generator_extrapolation_assign (const Grid &y, const Congruence_System &cgs, unsigned *tp=NULL)

  Improves the result of the generator variant of the Grid widening computation by also enforcing those congruences in `cgs` *that are satisfied by all the points of* `*this`.

- 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 &vars)

  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 &vars, Variable dest)

  Folds the space dimensions in `vars` *into* `dest`.

Miscellaneous Member Functions

- ~Grid ()

  Destructor.

- void swap (Grid &y)

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

- void ascii_dump () const

  *Writes to std::cerr an ASCII representation of *this.*

- void ascii_dump (std::ostream &s) const

  *Writes to s an ASCII representation of *this.*

- void print () const

  *Prints *this to std::cerr using operator<<.*

- bool ascii_load (std::istream &s)

  *Loads from s an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets *this accordingly. Returns true if successful, false otherwise.*

- 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.*

- int32_t hash_code () const

  *Returns a 32-bit hash code for *this.*

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension all kinds of Grid can handle.

- static bool can_recycle_congruence_systems ()

  Returns true indicating that this domain has methods that can recycle congruences.

- static bool can_recycle_constraint_systems ()

  Returns true indicating that this domain has methods that can recycle constraints.

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`*.*

10.26.1 Detailed Description

A grid. An object of the class Grid represents a rational grid.

The domain of grids *optimally supports*:

- all (proper and non-proper) congruences;
- tautological and inconsistent constraints;
- linear equality constraints (i.e., non-proper congruences).

Depending on the method, using a constraint that is not optimally supported by the domain will either raise an exception or result in a (possibly non-optimal) upward approximation.

The domain of grids support a concept of double description similar to the one developed for polyhedra: hence, 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 a 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.

A key attribute 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_grid_generator(grid_point(0*x + 0*y));
gr.add_grid_generator(grid_line(x));
gr.add_grid_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

$$\left\{ (2, y)^{\mathrm{T}} \in \mathbb{R}^2 \;\middle|\; y \in \mathbb{R} \right\}.$$

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 $\left\{(2, 0)^{\mathrm{T}}\right\} \subseteq \mathbb{R}^2$.

Example 7

The following code shows the use of the function affine_image:

```
Grid gr(2, EMPTY);
gr.add_grid_generator(grid_point(0*x + 0*y));
gr.add_grid_generator(grid_point(4*x + 0*y));
gr.add_grid_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_grid_generator(grid_point(0*x + 0*y));
gr.add_grid_generator(grid_point(4*x + 0*y));
gr.add_grid_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 vars;
vars.insert(y);
vars.insert(z);
gr.remove_space_dimensions(vars);
```

The starting grid is the singleton set $\{(3,1,0,2)^{\mathrm{T}}\} \subseteq \mathbb{R}^4$, while the resulting grid is $\{(3,2)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$. 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> vars1;
vars1.insert(y);
gr.remove_space_dimensions(vars1);
set<Variable> vars2;
vars2.insert(z);
gr.remove_space_dimensions(vars2);
```

In this case, the result is the grid $\{(3,0)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$: when removing the set of dimensions `vars2` 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.

10.26.2 Constructor & Destructor Documentation

10.26.2.1 Parma_Polyhedra_Library::Grid::Grid (dimension_type *num_dimensions* = 0, Degenerate_Element *kind* = UNIVERSE) [inline, 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.

10.26.2.2 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.

10.26.2.3 Parma_Polyhedra_Library::Grid::Grid (Congruence_System & *cgs*, Recycle_Input *dummy*) `[inline]`

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 may be recycled to build the grid.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.26.2.4 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::invalid_argument Thrown if the constraint system `cs` contains inequality constraints.

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.26.2.5 Parma_Polyhedra_Library::Grid::Grid (Constraint_System & *cs*, Recycle_Input *dummy*)

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. Its data-structures may be recycled to build the grid.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

std::invalid_argument Thrown if the constraint system `cs` contains inequality constraints.

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.26.2.6 Parma_Polyhedra_Library::Grid::Grid (const Grid_Generator_System & *const_gs*) [inline, explicit]

Builds a grid, copying a system of grid 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.

10.26.2.7 Parma_Polyhedra_Library::Grid::Grid (Grid_Generator_System & *gs*, Recycle_Input *dummy*) [inline]

Builds a grid, recycling a system of grid generators.

The grid inherits the space dimension of the generator system.

Parameters

gs The system of generators defining the grid. Its data-structures may be recycled to build the grid.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

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.

10.26.2.8 template<typename Interval > Parma_Polyhedra_Library::Grid::Grid (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a grid out of a box.

The grid inherits the space dimension of the box. The built grid is the most precise grid that includes the box.

Parameters

box The box representing the grid to be built.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.26.2.9 template<typename U > Parma_Polyhedra_Library::Grid::Grid (const BD_Shape< U > & *bd*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a grid out of a bounded-difference shape.

The grid inherits the space dimension of the BDS. The built grid is the most precise grid that includes the BDS.

Parameters

bd The BDS representing the grid to be built.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `bd` exceeds the maximum allowed space dimension.

10.26.2.10 template<typename U > Parma_Polyhedra_Library::Grid::Grid (const Octagonal_Shape< U > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a grid out of an octagonal shape.

The grid inherits the space dimension of the octagonal shape. The built grid is the most precise grid that includes the octagonal shape.

Parameters

os The octagonal shape representing the grid to be built.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `os` exceeds the maximum allowed space dimension.

10.26.2.11 Parma_Polyhedra_Library::Grid::Grid (const Polyhedron & *ph*, Complexity_Class *complexity* = `ANY_COMPLEXITY`) `[explicit]`

Builds a grid from a polyhedron using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the grid built is the smallest one containing `ph`.

The grid inherits the space dimension of polyhedron.

Parameters

ph The polyhedron.

complexity The complexity class.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.26.2.12 Parma_Polyhedra_Library::Grid::Grid (const Grid & *y*, Complexity_Class *complexity* = `ANY_COMPLEXITY`)

Ordinary copy constructor.

The complexity argument is ignored.

10.26.3 Member Function Documentation

10.26.3.1 bool Parma_Polyhedra_Library::Grid::is_topologically_closed () const

Returns `true` if and only if `*this` is a topologically closed subset of the vector space.

A grid is always topologically closed.

10.26.3.2 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.

10.26.3.3 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.

10.26.3.4 bool Parma_Polyhedra_Library::Grid::constrains (Variable *var*) const

Returns true if and only if var is constrained in *this.

Exceptions

std::invalid_argument Thrown if var is not a space dimension of *this.

10.26.3.5 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.

10.26.3.6 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.

10.26.3.7 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.

10.26.3.8 bool Parma_Polyhedra_Library::Grid::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 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.

10.26.3.9 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.

10.26.3.10 bool Parma_Polyhedra_Library::Grid::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 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.

10.26.3.11 bool Parma_Polyhedra_Library::Grid::frequency (const Linear_Expression & *expr*, Coefficient & *freq_n*, Coefficient & *freq_d*, Coefficient & *val_n*, Coefficient & *val_d*) const

Returns true if and only if *this is not empty and frequency for *this with respect to expr is defined, in which case the frequency and the value for expr that is closest to zero are computed.

Parameters

expr The linear expression for which the frequency is needed;

freq_n The numerator of the maximum frequency of expr;

freq_d The denominator of the maximum frequency of expr;

val_n The numerator of them value of expr at a point in the grid that is closest to zero;

val_d The denominator of a value of expr at a point in the grid that is closest to zero;

Exceptions

std::invalid_argument Thrown if expr and *this are dimension-incompatible.

If *this is empty or frequency is undefined with respect to expr, then false is returned and freq_n, freq_d, val_n and val_d are left untouched.

10.26.3.12 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.

10.26.3.13 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.

10.26.3.14 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.

10.26.3.15 void Parma_Polyhedra_Library::Grid::add_congruence (const Congruence & *cg*) [inline]

Adds a copy of congruence `cg` to `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

10.26.3.16 void Parma_Polyhedra_Library::Grid::add_grid_generator (const Grid_Generator & *g*)

Adds a copy of grid 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.

10.26.3.17 void Parma_Polyhedra_Library::Grid::add_congruences (const Congruence_System & *cgs*) `[inline]`

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.

10.26.3.18 void Parma_Polyhedra_Library::Grid::add_recycled_congruences (Congruence_System & *cgs*)

Adds the congruences in `cgs` to *this.

Parameters

cgs The congruence system to be added to `*this`. The congruences in `cgs` may be recycled.

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.

10.26.3.19 void Parma_Polyhedra_Library::Grid::add_constraint (const Constraint & *c*) `[inline]`

Adds to `*this` a congruence equivalent to constraint `c`.

Parameters

c The constraint to be added.

Exceptions

std::invalid_argument Thrown if *this and c are dimension-incompatible or if constraint c is not optimally supported by the grid domain.

10.26.3.20 void Parma_Polyhedra_Library::Grid::add_constraints (const Constraint_System & *cs*)

Adds to *this congruences equivalent to the constraints in cs.

Parameters

cs The constraints to be added.

Exceptions

std::invalid_argument Thrown if *this and cs are dimension-incompatible or if cs contains a constraint whcih is not optimally supported by the grid domain.

10.26.3.21 void Parma_Polyhedra_Library::Grid::add_recycled_constraints (Constraint_System & *cs*) [inline]

Adds to *this congruences equivalent to the constraints in cs.

Parameters

cs The constraints to be added. They may be recycled.

Exceptions

std::invalid_argument Thrown if *this and cs are dimension-incompatible or if cs contains a constraint whcih is not optimally supported by the grid domain.

Warning

The only assumption that can be made about cs upon successful or exceptional return is that it can be safely destroyed.

10.26.3.22 void Parma_Polyhedra_Library::Grid::refine_with_congruence (const Congruence & *cg*) [inline]

Uses a copy of the congruence cg to refine *this.

Parameters

cg The congruence used.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

10.26.3.23 void Parma_Polyhedra_Library::Grid::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Uses a copy of the congruences in `cgs` to refine `*this`.

Parameters

cgs The congruences used.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.26.3.24 void Parma_Polyhedra_Library::Grid::refine_with_constraint (const Constraint & *c*)

Uses a copy of the constraint `c` to refine `*this`.

Parameters

c The constraint used. If it is not an equality, it will be ignored

Exceptions

std::invalid_argument Thrown if `*this` and `c` are dimension-incompatible.

10.26.3.25 void Parma_Polyhedra_Library::Grid::refine_with_constraints (const Constraint_System & *cs*)

Uses a copy of the constraints in `cs` to refine `*this`.

Parameters

cs The constraints used. Constraints that are not equalities are ignored.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.26.3.26 void Parma_Polyhedra_Library::Grid::add_grid_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.

10.26.3.27 void Parma_Polyhedra_Library::Grid::add_recycled_grid_generators (Grid_Generator_System & *gs*)

Adds the generators in gs to the system of generators of this.

Parameters

gs The generator system to be added to *this. The generators in gs may be recycled.

Exceptions

std::invalid_argument Thrown if *this and gs are dimension-incompatible.

Warning

The only assumption that can be made about gs upon successful or exceptional return is that it can be safely destroyed.

10.26.3.28 void Parma_Polyhedra_Library::Grid::unconstrain (Variable *var*)

Computes the cylindrification of *this with respect to space dimension var, assigning the result to *this.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if var is not a space dimension of *this.

10.26.3.29 void Parma_Polyhedra_Library::Grid::unconstrain (const Variables_Set & *vars*)

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.26.3.30 void Parma_Polyhedra_Library::Grid::intersection_assign (const Grid & *y*)

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.26.3.31 void Parma_Polyhedra_Library::Grid::upper_bound_assign (const Grid & *y*)

Assigns to `*this` the least upper bound of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.26.3.32 bool Parma_Polyhedra_Library::Grid::upper_bound_assign_if_exact (const Grid & *y*)

If the upper bound 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.

10.26.3.33 void Parma_Polyhedra_Library::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.

10.26.3.34 bool Parma_Polyhedra_Library::Grid::simplify_using_context_assign (const Grid & *y*)

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.26.3.35 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`.

10.26.3.36 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`.

10.26.3.37 void Parma_Polyhedra_Library::Grid::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`, Coefficient_traits::const_reference *modulus* = `Coefficient_zero()`)

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;

relsym The relation symbol where EQUAL is the symbol for a congruence 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 zero.

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`.

10.26.3.38 void Parma_Polyhedra_Library::Grid::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`, Coefficient_traits::const_reference *modulus* = `Coefficient_zero()`)

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;

relsym The relation symbol where EQUAL is the symbol for a congruence 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 zero.

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`.

10.26.3.39 void Parma_Polyhedra_Library::Grid::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*, Coefficient_traits::const_reference *modulus* = `Coefficient_zero()`)

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.

relsym The relation symbol where EQUAL is the symbol for a congruence relation;

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 zero.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with `lhs` or `rhs`.

10.26.3.40 void Parma_Polyhedra_Library::Grid::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*, Coefficient_traits::const_reference *modulus* = `Coefficient_zero()`)

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;

relsym The relation symbol where EQUAL is the symbol for a congruence relation;

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 zero.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with `lhs` or `rhs`.

10.26.3.41 void Parma_Polyhedra_Library::Grid::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`.

10.26.3.42 void Parma_Polyhedra_Library::Grid::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`.

10.26.3.43 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.

10.26.3.44 void Parma_Polyhedra_Library::Grid::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System * *pcs* = `0`, unsigned *complexity_threshold* = `16`, bool *wrap_individually* = `true`)

Wraps the specified dimensions of the vector space.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be wrapped.

w The width of the bounded integer type corresponding to all the dimensions to be wrapped.

r The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

o The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.

pcs Possibly null pointer to a constraint system. This argument is for compatibility with wrap_assign() for the other domains and only checked for dimension-compatibility.

complexity_threshold A precision parameter of the wrapping operator. This argument is for compatibility with wrap_assign() for the other domains and is ignored.

wrap_individually `true` if the dimensions should be wrapped individually. As wrapping dimensions collectively does not improve the precision, this argument is ignored.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars` or with `*pcs`.

Warning

It is assumed that variables in `Vars` represent integers. Thus, where the extra cost is negligible, the integrality of these variables is enforced; possibly causing a non-integral grid to become empty.

10.26.3.45 void Parma_Polyhedra_Library::Grid::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY)

Possibly tightens `*this` by dropping all points with non-integer coordinates.

Parameters

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.26.3.46 void Parma_Polyhedra_Library::Grid::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY)

Possibly tightens `*this` by dropping all points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.26.3.47 void Parma_Polyhedra_Library::Grid::congruence_widening_assign (const Grid & *y*, unsigned * *tp* = NULL)

Assigns to *this the result of computing the Grid widening between *this and y using congruence systems.

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.

10.26.3.48 void Parma_Polyhedra_Library::Grid::generator_widening_assign (const Grid & *y*, unsigned * *tp* = NULL)

Assigns to *this the result of computing the Grid widening between *this and y using generator systems.

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.

10.26.3.49 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.

This widening uses either the congruence or generator systems depending on which of the systems describing x and y are up to date and minimized.

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.

10.26.3.50 void Parma_Polyhedra_Library::Grid::limited_congruence_extrapolation_assign (const Grid & *y*, const Congruence_System & *cgs*, unsigned * *tp* = NULL)

Improves the result of the congruence variant of 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 cgs are dimension-incompatible.

10.26.3.51 void Parma_Polyhedra_Library::Grid::limited_generator_extrapolation_assign (const Grid & *y*, const Congruence_System & *cgs*, unsigned * *tp* = NULL)

Improves the result of the generator variant 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 cgs are dimension-incompatible.

10.26.3.52 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 `cgs` are dimension-incompatible.

10.26.3.53 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

$$\left\{ (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \;\middle|\; (x, y)^{\mathrm{T}} \in \mathcal{L} \right\}.$$

10.26.3.54 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

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

10.26.3.55 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()`.

10.26.3.56 void Parma_Polyhedra_Library::Grid::remove_space_dimensions (const Variables_Set & *vars*)

Removes all the specified dimensions from the vector space.

Parameters

vars 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 `vars`.

10.26.3.57 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`.

10.26.3.58 template<typename Partial_Function > void Parma_Polyhedra_Library::Grid::map_space_dimensions (const Partial_Function & *pfunc*) `[inline]`

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 type parameter 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.

10.26.3.59 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 $n, n+1, \dots, n+m-1$.

10.26.3.60 void Parma_Polyhedra_Library::Grid::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*)

Folds the space dimensions in `vars` into `dest`.

Parameters

- ***vars*** The set of Variable objects corresponding to the space dimensions to be folded;
- ***dest*** 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 `dest` or with one of the Variable objects contained in `vars`. Also thrown if `dest` is contained in `vars`.

If `*this` has space dimension n, with $n > 0$, `dest` has space dimension $k \leq n$, `vars` is a set of variables whose maximum space dimension is also less than or equal to n, and `dest` is not a member of `vars`, then the space dimensions corresponding to variables in `vars` are folded into the k-th space dimension.

10.26.3.61 int32_t Parma_Polyhedra_Library::Grid::hash_code () const [inline]

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.26.4 Friends And Related Function Documentation

10.26.4.1 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.

10.26.4.2 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.

10.26.4.3 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.

10.26.4.4 void swap (Parma_Polyhedra_Library::Grid & *x*, Parma_Polyhedra_Library::Grid & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.27 Parma_Polyhedra_Library::Grid_Certificate Class Reference

The convergence certificate for the Grid widening operator.

`#include <ppl.hh>`

Classes

- struct Compare

  A total ordering on Grid certificates.

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`.

10.27.1 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.

10.27.2 Member Function Documentation

10.27.2.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.28 Parma_Polyhedra_Library::Grid_Generator Class Reference

A grid line, parameter or grid point.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Generator.

Public Types

- enum Type { LINE, PARAMETER, POINT }

  The generator type.

Public Member Functions

- Grid_Generator (const Grid_Generator &g)

  Ordinary copy constructor.

- ~Grid_Generator ()

  Destructor.

- Grid_Generator & operator= (const Grid_Generator &g)

  Assignment operator.

- Grid_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_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.*

- Coefficient_traits::const_reference coefficient (Variable v) const

  Returns the coefficient of `v` *in* `*this`.

- 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 is_equal_at_dimension (dimension_type dim, const Grid_Generator &gg) const

  Returns `true` *if* `*this` *is equal to* `gg` *in dimension* `dim`.

- bool all_homogeneous_terms_are_zero () const

  Returns `true` *if and only if all the homogeneous terms of* `*this` *are* 0.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 grid_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())

  Returns the parameter of direction `e` *and size* `e/d`.

- static Grid_Generator grid_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one())

  Returns the point at `e` / `d`.

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Grid_Generator can handle.

- static void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- static const Grid_Generator & zero_dim_point ()

  Returns the origin of the zero-dimensional space $\mathbb{R}^0$.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Grid_Generator &g)

  Output operator.

- void swap (Parma_Polyhedra_Library::Grid_Generator &x, Parma_Polyhedra_Library::Grid_-Generator &y)

  Specializes `std::swap`.

- 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.

10.28.1 Detailed Description

A grid line, parameter or grid point. An object of the class Grid_Generator is one of the following:

- a grid_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 grid_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 grid_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 (`grid_line`, `parameter` or `grid_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 grid 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 grid 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 grid 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 grid 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 grid 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 grid 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 grid 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 grid 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 << "Grid point g1: " << g1 << endl;
  Linear_Expression e;
  for (dimension_type i = g1.space_dimension(); i-- > 0; )
    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 grid point." << endl;
```

Therefore, for the grid point

```
Grid_Generator g1 = grid_point(2*x - y + 3*z, 2);
```

we would obtain the following output:

```
Grid point g1: p((2*A - B + 3*C)/2)
Parameter g2: parameter((2*A - 2*B + 9*C)/2)
```

When working with grid 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.

10.28.2 Member Enumeration Documentation

10.28.2.1 enum Parma_Polyhedra_Library::Grid_Generator::Type

The generator type.

Enumerator:

LINE The generator is a grid line.

PARAMETER The generator is a parameter.

POINT The generator is a grid point.

Reimplemented from Parma_Polyhedra_Library::Generator.

10.28.3 Member Function Documentation

10.28.3.1 Grid_Generator grid_line (const Linear_Expression & *e*) `[inline, static]`

Returns the line of direction `e`.

Shorthand for Grid_Generator Grid_Generator::grid_line(const Linear_Expression& e).

Exceptions

std::invalid_argument Thrown if the homogeneous part of `e` represents the origin of the vector space.

10.28.3.2 Grid_Generator parameter (const Linear_Expression & *e* = `Linear_Expression::zero()`, Coefficient_traits::const_reference *d* = `Coefficient_one()`) `[inline, static]`

Returns the parameter of direction `e` and size `e/d`.

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.

10.28.3.3 Grid_Generator grid_point (const Linear_Expression & *e* = Linear_Expression::zero(), Coefficient_traits::const_reference *d* = Coefficient_one()) [inline, static]

Returns the point at e / d.

Shorthand for Grid_Generator Grid_Generator::grid_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.

10.28.3.4 Coefficient_traits::const_reference Parma_Polyhedra_Library::Grid_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.

Reimplemented from Parma_Polyhedra_Library::Generator.

10.28.3.5 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.

10.28.3.6 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.

10.28.3.7 void Parma_Polyhedra_Library::Grid_Generator::coefficient_swap (Grid_Generator & *y*)

Swaps `*this` with `y`, leaving `*this` with the original capacity.

All elements 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`.

10.28.4 Friends And Related Function Documentation

10.28.4.1 std::ostream & operator<< (std::ostream & *s*, const Grid_Generator & *g*) [related]

Output operator.

10.28.4.2 void swap (Parma_Polyhedra_Library::Grid_Generator & *x*, Parma_Polyhedra_Library::Grid_Generator & *y*) [related]

Specializes `std::swap`.

10.28.4.3 bool operator== (const Grid_Generator & *x*, const Grid_Generator & *y*) [related]

Returns `true` if and only if `x` is equivalent to `y`.

10.28.4.4 bool operator!= (const Grid_Generator & *x*, const Grid_Generator & *y*) [related]

Returns `true` if and only if `x` is not equivalent to `y`.

10.28.4.5 std::ostream & operator<< (std::ostream & *s*, const Grid_Generator::Type & *t*) [related]

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.29 Parma_Polyhedra_Library::Grid_Generator_System Class Reference

A system of grid generators.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Generator_System.

Classes

- class const_iterator

  An iterator over a system of grid generators.

Public Member Functions

- Grid_Generator_System ()

  Default constructor: builds an empty system of generators.

- Grid_Generator_System (const Grid_Generator &g)

  Builds the singleton system containing only generator `g`*.*

- Grid_Generator_System (dimension_type dim)

  Builds an empty system of generators of dimension `dim`*.*

- Grid_Generator_System (const Grid_Generator_System &gs)

  Ordinary copy constructor.

- ~Grid_Generator_System ()

  Destructor.

- Grid_Generator_System & operator= (const Grid_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 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.*

- bool empty () const

  Returns `true` *if and only if* `*this` *has no generators.*

- 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.

- dimension_type num_rows () const

  Returns the number of rows (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`*.*

- bool OK () const

  Checks if all the invariants are satisfied.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`*.*

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`*.*

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`*.*

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 (Grid_Generator_System &y)

  Swaps `*this` *with* `y`*.*

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Grid_Generator_System can handle.

- static void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- static const Grid_Generator_System & zero_dim_univ ()

  Returns the singleton system containing only Grid_Generator::zero_dim_point().

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`.

10.29.1 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

$$\left\{ (x,0)^{\mathrm{T}} \in \mathbb{R}^2 \;\middle|\; x \pmod{1}\, 0 \right\},$$

```
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.

10.29.2 Member Function Documentation

10.29.2.1 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`.

10.29.2.2 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.

10.29.2.3 bool Parma_Polyhedra_Library::Grid_Generator_System::ascii_load (std::istream & *s*)

Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

Resizes the matrix of generators using the numbers of rows and columns read from `s`, then initializes the coordinates of each generator and its type reading the contents from `s`.

Reimplemented from Parma_Polyhedra_Library::Generator_System.

10.29.3 Friends And Related Function Documentation

10.29.3.1 bool operator== (const Grid_Generator_System & *x*, const Grid_Generator_System & *y*) [friend]

Returns `true` if and only if `x` and `y` are identical.

10.29.3.2 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 ", ".

10.29.3.3 void swap (Parma_Polyhedra_Library::Grid_Generator_System & x, Parma_Polyhedra_Library::Grid_Generator_System & y) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.30 Parma_Polyhedra_Library::H79_Certificate Class Reference

A convergence certificate for the H79 widening operator.

`#include <ppl.hh>`

Classes

- struct Compare

  A total ordering on H79 certificates.

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`*.*

10.30.1 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.

10.30.2 Member Function Documentation

10.30.2.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.31 Parma_Polyhedra_Library::Interval< Boundary, Info > Class Template Reference

A generic, not necessarily closed, possibly restricted interval.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Interval_Base.

Public Member Functions

- void swap (Interval &y)

  Swaps `*this` *with* `y`*.*

- void topological_closure_assign ()

  Assigns to `*this` *its topological closure.*

- 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`*.*

- Interval (const char *s)

Builds the smallest interval containing the number whose textual representation is contained in `s`*.*

- template< typename From >
  Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type difference_assign (const From &x)

  Assigns to `*this` *the smallest interval containing the set-theoretic difference of* `*this` *and* `x`*.*

- template< typename From1 , typename From2 >
  Enable_If<((Is_Singleton< From1 >::value||Is_Interval< From1 >::value)&&(Is_Singleton< From2 >::value||Is_Interval< From2 >::value)), I_Result >::type difference_assign (const From1 &x, const From2 &y)

  Assigns to `*this` *the smallest interval containing the set-theoretic difference of* `x` *and* `y`*.*

- template< typename From >
  Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type lower_approximation_difference_assign (const From &x)

  Assigns to `*this` *the largest interval contained in the set-theoretic difference of* `*this` *and* `x`*.*

- template< typename From >
  Enable_If< Is_Interval< From >::value, bool >::type simplify_using_context_assign (const From &y)

  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`*.*

- template< typename From >
  Enable_If< Is_Interval< From >::value, void >::type empty_intersection_assign (const From &y)

  Assigns to `*this` *an interval having empty intersection with* `y`*. The assigned interval should be as large as possible.*

- template< typename From >
  Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type refine_existential (Relation_Symbol rel, const From &x)

  Refines `to` *according to the existential relation* `rel` *with* `x`*.*

- template< typename From >
  Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type refine_universal (Relation_Symbol rel, const From &x)

  Refines `to` *so that it satisfies the universal relation* `rel` *with* `x`*.*

- template< typename From1 , typename From2 >
  Enable_If<((Is_Singleton< From1 >::value||Is_Interval< From1 >::value)&&(Is_Singleton< From2 >::value||Is_Interval< From2 >::value)), I_Result >::type mul_assign (const From1 &x, const From2 &y)
- template< typename From1 , typename From2 >
  Enable_If<((Is_Singleton< From1 >::value||Is_Interval< From1 >::value)&&(Is_Singleton< From2 >::value||Is_Interval< From2 >::value)), I_Result >::type div_assign (const From1 &x, const From2 &y)

Related Functions

(Note that these are not member functions.)

- template<typename Boundary , typename Info >
  void swap (Parma_Polyhedra_Library::Interval< Boundary, Info > &x, Parma_Polyhedra_Library::Interval< Boundary, Info > &y)

10.31.1 Detailed Description

template<typename Boundary, typename Info> class Parma_Polyhedra_Library::Interval< Boundary, Info >

A generic, not necessarily closed, possibly restricted interval. The class template type parameter `Boundary` represents the type of the interval boundaries, and can be chosen, among other possibilities, within one of the following number families:

- a bounded precision native integer type (that is, from `signed char` to `long long` and from `int8_t` to `int64_t`);
- a bounded precision floating point type (`float`, `double` or `long double`);
- an unbounded integer or rational type, as provided by the C++ interface of GMP (`mpz_class` or `mpq_class`).

The class template type parameter `Info` allows to control a number of features of the class, among which:

- the ability to support open as well as closed boundaries;
- the ability to represent empty intervals in addition to nonempty ones;
- the ability to represent intervals of extended number families that contain positive and negative infinities;
- the ability to support (independently from the type of the boundaries) plain intervals of real numbers and intervals subject to generic *restrictions* (e.g., intervals of integer numbers).

10.31.2 Member Function Documentation

10.31.2.1 template<typename Boundary , typename Info > template<typename From > Enable_If< Is_Interval< From >::value, bool >::type Parma_Polyhedra_Library::Interval< Boundary, Info >::simplify_using_context_assign (const From & *y*) `[inline]`

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`.

Returns

`false` if and only if the meet of `*this` and `y` is empty.

10.31.2.2 template<typename Boundary , typename Info > template<typename From > Enable_If< Is_Interval< From >::value, void >::type Parma_Polyhedra_Library::Interval< Boundary, Info >::empty_intersection_assign (const From & *y*) `[inline]`

Assigns to `*this` an interval having empty intersection with `y`. The assigned interval should be as large as possible.

Note

Depending on interval restrictions, there could be many maximal intervals all inconsistent with respect to y.

10.31.2.3 template<typename To_Boundary , typename To_Info > template<typename From > Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type Parma_Polyhedra_Library::Interval< To_Boundary, To_Info >::refine_existential (Relation_Symbol *rel*, const From & *x*) `[inline]`

Refines `to` according to the existential relation `rel` with `x`.

The `to` interval is restricted to become, upon successful exit, the smallest interval of its type that contains the set

$$\{ a \in \mathtt{to} \mid \exists b \in \mathtt{x} \,.\, a \mathrel{\mathtt{rel}} b \}.$$

Returns

???

10.31.2.4 template<typename To_Boundary , typename To_Info > template<typename From > Enable_If< Is_Singleton< From >::value||Is_Interval< From >::value, I_Result >::type Parma_Polyhedra_Library::Interval< To_Boundary, To_Info >::refine_universal (Relation_Symbol *rel*, const From & *x*) `[inline]`

Refines `to` so that it satisfies the universal relation `rel` with `x`.

The `to` interval is restricted to become, upon successful exit, the smallest interval of its type that contains the set

$$\{ a \in \mathtt{to} \mid \forall b \in \mathtt{x} : a \mathrel{\mathtt{rel}} b \}.$$

Returns

???

10.31.2.5 template<typename To_Boundary , typename To_Info > template<typename From1 , typename From2 > Enable_If<((Is_Singleton< From1 >::value||Is_Interval< From1 >::value)&&(Is_Singleton< From2 >::value||Is_Interval< From2 >::value)), I_Result >::type Parma_Polyhedra_Library::Interval< To_Boundary, To_Info >::mul_assign (const From1 & *x*, const From2 & *y*) `[inline]`

+---------+-----------+-----------+-----------------+ | ∗ | yl > 0 | yu < 0 | yl < 0, yu > 0 | +---------+-----------+-----------+-----------------+ | xl > 0 |xl∗yl,xu∗yu|xu∗yl,xl∗yu| xu∗yl,xu∗yu | +---------+-----------+-----------+-----------------+ | xu < 0 |xl∗yu,xu∗yl|xu∗yu,xl∗yl| xl∗yu,xl∗yl | +---------+-----------+-----------+-----------------+ |xl<0 xu>0|xl∗yu,xu∗yu|xu∗yl,xl∗yl|min(xl∗yu,xu∗yl),| | | | |max(xl∗yl,xu∗yu) | +---------+-----------+-----------+-----------------+

10.31.2.6 template<typename To_Boundary , typename To_Info > template<typename From1 , typename From2 > Enable_If<((Is_Singleton< From1 >::value||Is_Interval< From1 >::value)&&(Is_Singleton< From2 >::value||Is_Interval< From2 >::value)), I_Result >::type Parma_Polyhedra_Library::Interval< To_Boundary, To_Info >::div_assign (const From1 & *x*, const From2 & *y*) `[inline]`

+-----------+-----------+-----------+ | / | yu < 0 | yl > 0 | +-----------+-----------+-----------+ | xu<=0 |xu/yl,xl/yu|xl/yl,xu/yu| +-----------+-----------+-----------+ |xl<=0 xu>=0|xu/yu,xl/yu|xl/yl,xu/yl| +-----------+-----------+-----------+ | xl>=0 |xu/yu,xl/yl|xl/yu,xu/yl| +-----------+-----------+-----------+

10.31.3 Friends And Related Function Documentation

10.31.3.1 template<typename Boundary , typename Info > void swap (Parma_Polyhedra_-Library::Interval< Boundary, Info > & *x*, Parma_Polyhedra_Library::Interval< Boundary, Info > & *y*) `[related]`

The documentation for this class was generated from the following file:

- ppl.hh

10.32 Parma_Polyhedra_Library::Is_Checked< T > Struct Template Reference

Inherits Parma_Polyhedra_Library::False.

10.32.1 Detailed Description

template<typename T> struct Parma_Polyhedra_Library::Is_Checked< T >

The documentation for this struct was generated from the following file:

- ppl.hh

10.33 Parma_Polyhedra_Library::Is_Checked< Checked_Number< T, P > > Struct Template Reference

Inherits Parma_Polyhedra_Library::True.

10.33.1 Detailed Description

template<typename T, typename P> struct Parma_Polyhedra_Library::Is_Checked< Checked_-Number< T, P > >

The documentation for this struct was generated from the following file:

- ppl.hh

10.34 Parma_Polyhedra_Library::Is_Native_Or_Checked< T > Struct Template Reference

Inherits Parma_Polyhedra_Library::Bool< Is_Native< T >::value||Is_Checked< T >::value >.

Inherited by Parma_Polyhedra_Library::Is_Singleton< T, Enable >.

10.34.1 Detailed Description

template<typename T> struct Parma_Polyhedra_Library::Is_Native_Or_Checked< T >

The documentation for this struct was generated from the following file:

- ppl.hh

10.35 Parma_Polyhedra_Library::Linear_Expression Class Reference

A linear expression.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::Linear_Row.

Inherited by Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter.

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 (Variable v)

  Builds the linear expression corresponding to the variable v.

- 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 Grid_Generator &g)

  Builds the linear expression corresponding to grid generator g *(for points, parameters and lines 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`.

- bool is_zero () const

  Returns `true` *if and only if* `*this` *is* 0.

- bool all_homogeneous_terms_are_zero () const

  Returns `true` *if and only if all the homogeneous terms of* `*this` *are* 0.

- 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`.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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 void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- 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+ (Variable v, const Linear_Expression &e)

  Returns the linear expression `v + e`.

- Linear_Expression operator+ (Variable v, Variable w)

  Returns the linear expression `v + w`.

- 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- (Variable v, Variable w)

  Returns the linear expression `v - w`.

- 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- (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* (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`*.*
- Linear_Expression & add_mul_assign (Linear_Expression &e, Coefficient_traits::const_reference n, Variable v)
  Returns the linear expression `e` + `n` $*$ `v` *and assigns it to* `e`*.*
- Linear_Expression & sub_mul_assign (Linear_Expression &e, Coefficient_traits::const_reference n, Variable v)
  Returns the linear expression `e` - `n` $*$ `v` *and assigns it to* `e`*.*

Related Functions

(Note that these are not member functions.)

- 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`*.*
- std::ostream & operator<< (std::ostream &s, const Linear_Expression &e)
  Output operator.
- void swap (Parma_Polyhedra_Library::Linear_Expression &x, Parma_Polyhedra_Library::Linear_-Expression &y)
  Specializes `std::swap`*.*

10.35.1 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.

10.35.2 Constructor & Destructor Documentation

10.35.2.1 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (Variable *v*)

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()`.

10.35.2.2 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const Constraint & *c*) [explicit]

Builds the linear expression corresponding to constraint `c`.

Given the constraint $c = \left(\sum_{i=0}^{n-1} a_i x_i + b \bowtie 0\right)$, 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.

10.35.2.3 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.

10.35.2.4 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const Grid_Generator & *g*) [explicit]

Builds the linear expression corresponding to grid generator `g` (for points, parameters and lines the divisor is not copied).

Given the grid generator $g = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^{\mathrm{T}}$ this builds the linear expression $\sum_{i=0}^{n-1} a_i x_i$. The inhomogeneous term of the linear expression is always 0.

10.35.2.5 Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const Congruence & *cg*) [explicit]

Builds the linear expression corresponding to congruence `cg`.

Given the congruence $cg = \left(\sum_{i=0}^{n-1} a_i x_i + b = 0 \pmod{m}\right)$, this builds the linear expression $\sum_{i=0}^{n-1} a_i x_i + b$.

10.35.3 Friends And Related Function Documentation

10.35.3.1 Linear_Expression operator+ (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the linear expression `e1 + e2`.

10.35.3.2 Linear_Expression operator+ (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the linear expression n + e.

10.35.3.3 Linear_Expression operator+ (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression e + n.

10.35.3.4 Linear_Expression operator+ (Variable *v*, const Linear_Expression & *e*) [friend]

Returns the linear expression v + e.

10.35.3.5 Linear_Expression operator+ (Variable *v*, Variable *w*) [friend]

Returns the linear expression v + w.

10.35.3.6 Linear_Expression operator- (const Linear_Expression & *e*) [friend]

Returns the linear expression - e.

10.35.3.7 Linear_Expression operator- (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the linear expression e1 - e2.

10.35.3.8 Linear_Expression operator- (Variable *v*, Variable *w*) [friend]

Returns the linear expression v - w.

10.35.3.9 Linear_Expression operator- (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the linear expression n - e.

10.35.3.10 Linear_Expression operator- (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression e - n.

10.35.3.11 Linear_Expression operator- (Variable *v*, const Linear_Expression & *e*) [friend]

Returns the linear expression v - e.

10.35.3.12 Linear_Expression operator- (const Linear_Expression & *e*, Variable *v*) [friend]

Returns the linear expression e - v.

10.35.3.13 Linear_Expression operator* (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [friend]

Returns the linear expression n * e.

10.35.3.14 Linear_Expression operator* (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression e * n.

10.35.3.15 Linear_Expression & operator+= (Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the linear expression e1 + e2 and assigns it to e1.

10.35.3.16 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().

10.35.3.17 Linear_Expression & operator+= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression e + n and assigns it to e.

10.35.3.18 Linear_Expression & operator-= (Linear_Expression & *e1*, const Linear_Expression & *e2*) [friend]

Returns the linear expression e1 - e2 and assigns it to e1.

10.35.3.19 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().

10.35.3.20 Linear_Expression & operator-= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression e - n and assigns it to e.

10.35.3.21 Linear_Expression & operator*= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [friend]

Returns the linear expression n * e and assigns it to e.

10.35.3.22 Linear_Expression & add_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, Variable *v*) [friend]

Returns the linear expression e + n * v and assigns it to e.

10.35.3.23 Linear_Expression & sub_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, Variable *v*) [friend]

Returns the linear expression e - n * v and assigns it to e.

10.35.3.24 Linear_Expression operator+ (const Linear_Expression & *e*, Variable *v*) [related]

Returns the linear expression e + v.

10.35.3.25 Linear_Expression operator+ (const Linear_Expression & *e*) [related]

Returns the linear expression e.

10.35.3.26 std::ostream & operator<< (std::ostream & *s*, const Linear_Expression & *e*) [related]

Output operator.

10.35.3.27 void swap (Parma_Polyhedra_Library::Linear_Expression & *x*, Parma_Polyhedra_Library::Linear_Expression & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.36 Parma_Polyhedra_Library::MIP_Problem Class Reference

A Mixed Integer (linear) Programming problem.

`#include <ppl.hh>`

Public Types

- enum Control_Parameter_Name { PRICING }

  Names of MIP problems' control parameters.

- enum Control_Parameter_Value { PRICING_STEEPEST_EDGE_FLOAT, PRICING_-STEEPEST_EDGE_EXACT, PRICING_TEXTBOOK }

  Possible values for MIP problem's control parameters.

- typedef Constraint_Sequence::const_iterator const_iterator

  A type alias for the read-only iterator on the constraints defining the feasible region.

Public Member Functions

- MIP_Problem (dimension_type dim=0)

  Builds a trivial MIP problem.

- template<typename In >
  MIP_Problem (dimension_type dim, In first, In last, const Variables_Set &int_vars, const Linear_-Expression &obj=Linear_Expression::zero(), Optimization_Mode mode=MAXIMIZATION)

  Builds an MIP problem having space dimension `dim` from the sequence of constraints in the range $[\mathrm{first}, \mathrm{last})$, the objective function `obj` and optimization mode `mode`; those dimensions whose indices occur in `int_vars` are constrained to take an integer value.

- template<typename In >
  MIP_Problem (dimension_type dim, In first, In last, const Linear_Expression &obj=Linear_-Expression::zero(), Optimization_Mode mode=MAXIMIZATION)

  Builds an MIP problem having space dimension `dim` from the sequence of constraints in the range $[\mathrm{first}, \mathrm{last})$, the objective function `obj` and optimization mode `mode`.

- MIP_Problem (dimension_type dim, const Constraint_System &cs, const Linear_Expression &obj=Linear_Expression::zero(), Optimization_Mode mode=MAXIMIZATION)

  Builds an MIP problem having space dimension `dim` from the constraint system `cs`, the objective function `obj` and optimization mode `mode`.

- MIP_Problem (const MIP_Problem &y)

  Ordinary copy constructor.

- ∼MIP_Problem ()

  Destructor.

- MIP_Problem & operator= (const MIP_Problem &y)

  Assignment operator.

- dimension_type space_dimension () const

  Returns the space dimension of the MIP problem.

- const Variables_Set & integer_space_dimensions () const

  Returns a set containing all the variables' indexes constrained to be integral.

- const_iterator constraints_begin () const

  Returns a read-only iterator to the first constraint defining the feasible region.

- const_iterator constraints_end () const

  Returns a past-the-end read-only iterator to the sequence of constraints defining the feasible region.

- const Linear_Expression & objective_function () const

  Returns the objective function.

- Optimization_Mode optimization_mode () const

  Returns the optimization mode.

- void clear ()

Resets `*this` *to be equal to the trivial MIP problem.*

- void add_space_dimensions_and_embed (dimension_type m)

  Adds `m` *new space dimensions and embeds the old MIP problem in the new vector space.*

- void add_to_integer_space_dimensions (const Variables_Set &i_vars)

  Sets the variables whose indexes are in set `i_vars` *to be integer space dimensions.*

- void add_constraint (const Constraint &c)

  Adds a copy of constraint `c` *to the MIP problem.*

- void add_constraints (const Constraint_System &cs)

  Adds a copy of the constraints in `cs` *to the MIP problem.*

- 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`.

- MIP_Problem_Status solve () const

  Optimizes the MIP problem.

- 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.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- 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 (MIP_Problem &y)

  Swaps `*this` with `y`.

- Control_Parameter_Value get_control_parameter (Control_Parameter_Name name) const

  Returns the value of the control parameter `name`.

- void set_control_parameter (Control_Parameter_Value value)

  Sets control parameter `value`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension an MIP_Problem can handle.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const MIP_Problem &lp)

  Output operator.

- void swap (Parma_Polyhedra_Library::MIP_Problem &x, Parma_Polyhedra_Library::MIP_-Problem &y)

  Specializes `std::swap`.

10.36.1 Detailed Description

A Mixed Integer (linear) Programming problem. An object of this class encodes a mixed integer (linear) programming problem. The MIP problem is specified by providing:

- the dimension of the vector space;
- the feasible region, by means of a finite set of linear equality and non-strict inequality constraints;
- the subset of the unknown variables that range over the integers (the other variables implicitly ranging over the reals);

- the objective function, described by a Linear_Expression;
- the optimization mode (either maximization or minimization).

The class provides support for the (incremental) solution of the MIP problem based on variations of the revised simplex method and on branch-and-bound techniques. The result of the resolution process is expressed in terms of an enumeration, encoding the feasibility and the unboundedness of the optimization problem. The class supports simple feasibility tests (i.e., no optimization), as well as the extraction of an optimal (resp., feasible) point, provided the MIP_Problem is optimizable (resp., feasible).

By exploiting the incremental nature of the solver, it is possible to reuse part of the computational work already done when solving variants of a given MIP_Problem: currently, incremental resolution supports the addition of space dimensions, the addition of constraints, the change of objective function and the change of optimization mode.

10.36.2 Member Enumeration Documentation

10.36.2.1 enum Parma_Polyhedra_Library::MIP_Problem::Control_Parameter_Name

Names of MIP problems' control parameters.

Enumerator:

PRICING The pricing rule.

10.36.2.2 enum Parma_Polyhedra_Library::MIP_Problem::Control_Parameter_Value

Possible values for MIP problem's control parameters.

Enumerator:

PRICING_STEEPEST_EDGE_FLOAT Steepest edge pricing method, using floating points (default).

PRICING_STEEPEST_EDGE_EXACT Steepest edge pricing method, using Coefficient.

PRICING_TEXTBOOK Textbook pricing method.

10.36.3 Constructor & Destructor Documentation

10.36.3.1 Parma_Polyhedra_Library::MIP_Problem::MIP_Problem (dimension_type *dim* = 0) [explicit]

Builds a trivial MIP problem.

A trivial MIP problem requires to maximize the objective function 0 on a vector space under no constraints at all: the origin of the vector space is an optimal solution.

Parameters

dim The dimension of the vector space enclosing `*this` (optional argument with default value 0).

Exceptions

std::length_error Thrown if dim exceeds max_space_dimension().

10.36.3.2 template<typename In > Parma_Polyhedra_Library::MIP_Problem::MIP_Problem (dimension_type *dim*, In *first*, In *last*, const Variables_Set & *int_vars*, const Linear_Expression & *obj* = Linear_Expression::zero(), Optimization_Mode *mode* = MAXIMIZATION) [inline]

Builds an MIP problem having space dimension dim from the sequence of constraints in the range $[\mathrm{first}, \mathrm{last})$, the objective function obj and optimization mode mode; those dimensions whose indices occur in int_vars are constrained to take an integer value.

Parameters

dim The dimension of the vector space enclosing *this.

first An input iterator to the start of the sequence of constraints.

last A past-the-end input iterator to the sequence of constraints.

int_vars The set of variables' indexes that are constrained to take integer values.

obj The objective function (optional argument with default value 0).

mode The optimization mode (optional argument with default value MAXIMIZATION).

Exceptions

std::length_error Thrown if dim exceeds max_space_dimension().

std::invalid_argument Thrown if a constraint in the sequence is a strict inequality, if the space dimension of a constraint (resp., of the objective function or of the integer variables) or the space dimension of the integer variable set is strictly greater than dim.

10.36.3.3 template<typename In > Parma_Polyhedra_Library::MIP_Problem::MIP_Problem (dimension_type *dim*, In *first*, In *last*, const Linear_Expression & *obj* = Linear_Expression::zero(), Optimization_Mode *mode* = MAXIMIZATION) [inline]

Builds an MIP problem having space dimension dim from the sequence of constraints in the range $[\mathrm{first}, \mathrm{last})$, the objective function obj and optimization mode mode.

Parameters

dim The dimension of the vector space enclosing *this.

first An input iterator to the start of the sequence of constraints.

last A past-the-end input iterator to the sequence of constraints.

obj The objective function (optional argument with default value 0).

mode The optimization mode (optional argument with default value MAXIMIZATION).

Exceptions

std::length_error Thrown if dim exceeds max_space_dimension().

std::invalid_argument Thrown if a constraint in the sequence is a strict inequality or if the space dimension of a constraint (resp., of the objective function or of the integer variables) is strictly greater than `dim`.

10.36.3.4 Parma_Polyhedra_Library::MIP_Problem::MIP_Problem (dimension_type *dim*, const Constraint_System & *cs*, const Linear_Expression & *obj* = `Linear_Expression::zero()`, Optimization_Mode *mode* = `MAXIMIZATION`)

Builds an MIP problem having space dimension `dim` from the constraint system `cs`, the objective function `obj` and optimization mode `mode`.

Parameters

dim The dimension of the vector space enclosing `*this`.

cs The constraint system defining the feasible region.

obj The objective function (optional argument with default value 0).

mode The optimization mode (optional argument with default value `MAXIMIZATION`).

Exceptions

std::length_error Thrown if `dim` exceeds `max_space_dimension()`.

std::invalid_argument Thrown if the constraint system contains any strict inequality or if the space dimension of the constraint system (resp., the objective function) is strictly greater than `dim`.

10.36.4 Member Function Documentation

10.36.4.1 void Parma_Polyhedra_Library::MIP_Problem::clear () `[inline]`

Resets `*this` to be equal to the trivial MIP problem.

The space dimension is reset to 0.

10.36.4.2 void Parma_Polyhedra_Library::MIP_Problem::add_space_dimensions_and_embed (dimension_type *m*)

Adds `m` new space dimensions and embeds the old MIP problem 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 MIP problem; they are initially unconstrained.

10.36.4.3 void Parma_Polyhedra_Library::MIP_Problem::add_to_integer_space_dimensions (const Variables_Set & *i_vars*)

Sets the variables whose indexes are in set `i_vars` to be integer space dimensions.

Exceptions

std::invalid_argument Thrown if some index in `i_vars` does not correspond to a space dimension in `*this`.

10.36.4.4 void Parma_Polyhedra_Library::MIP_Problem::add_constraint (const Constraint & *c*)

Adds a copy of constraint `c` to the MIP problem.

Exceptions

std::invalid_argument Thrown if the constraint `c` is a strict inequality or if its space dimension is strictly greater than the space dimension of `*this`.

10.36.4.5 void Parma_Polyhedra_Library::MIP_Problem::add_constraints (const Constraint_System & *cs*)

Adds a copy of the constraints in `cs` to the MIP problem.

Exceptions

std::invalid_argument Thrown if the constraint system `cs` contains any strict inequality or if its space dimension is strictly greater than the space dimension of `*this`.

10.36.4.6 void Parma_Polyhedra_Library::MIP_Problem::set_objective_function (const Linear_Expression & *obj*)

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`.

10.36.4.7 bool Parma_Polyhedra_Library::MIP_Problem::is_satisfiable () const

Checks satisfiability of `*this`.

Returns

true if and only if the MIP problem is satisfiable.

10.36.4.8 MIP_Problem_Status Parma_Polyhedra_Library::MIP_Problem::solve () const

Optimizes the MIP problem.

Returns

An MIP_Problem_Status flag indicating the outcome of the optimization attempt (unfeasible, unbounded or optimized problem).

10.36.4.9 void Parma_Polyhedra_Library::MIP_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.

10.36.4.10 const Generator& Parma_Polyhedra_Library::MIP_Problem::feasible_point () const

Returns a feasible point for *this, if it exists.

Exceptions

std::domain_error Thrown if the MIP problem is not satisfiable.

10.36.4.11 const Generator& Parma_Polyhedra_Library::MIP_Problem::optimizing_point () const

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 MIP problem is unbounded or not satisfiable.

10.36.4.12 void Parma_Polyhedra_Library::MIP_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 MIP problem is unbounded or not satisfiable.

10.36.5 Friends And Related Function Documentation

10.36.5.1 std::ostream & operator<< (std::ostream & *s*, const MIP_Problem & *lp*) [related]

Output operator.

10.36.5.2 void swap (Parma_Polyhedra_Library::MIP_Problem & *x*, Parma_Polyhedra_Library::MIP_Problem & *y*) [related]

Specializes std::swap.

The documentation for this class was generated from the following file:

- ppl.hh

10.37 Parma_Polyhedra_Library::NNC_Polyhedron Class Reference

A not necessarily closed convex polyhedron.

```
#include <ppl.hh>
```

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, Recycle_Input dummy)

  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, Recycle_Input dummy)

  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, Recycle_Input dummy)

  Builds an NNC polyhedron recycling a system of congruences.

- NNC_Polyhedron (const C_Polyhedron &y, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds an NNC polyhedron from the C polyhedron `y`*.*

- template<typename Interval >
  NNC_Polyhedron (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds an NNC polyhedron out of a box.

- NNC_Polyhedron (const Grid &grid, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds an NNC polyhedron out of a grid.

- template<typename U >
  NNC_Polyhedron (const BD_Shape< U > &bd, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a NNC polyhedron out of a BD shape.

- template<typename U >
  NNC_Polyhedron (const Octagonal_Shape< U > &os, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a NNC polyhedron out of an octagonal shape.

- NNC_Polyhedron (const NNC_Polyhedron &y, Complexity_Class complexity=ANY_-COMPLEXITY)

  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).

10.37.1 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.

10.37.2 Constructor & Destructor Documentation

10.37.2.1 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.

10.37.2.2 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.

10.37.2.3 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Constraint_System & *cs*, Recycle_Input *dummy*) [inline]

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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

10.37.2.4 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.

10.37.2.5 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Generator_System & *gs*, Recycle_Input *dummy*) [inline]

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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

std::invalid_argument Thrown if the system of generators is not empty but has no points.

10.37.2.6 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 may be recycled to build the polyhedron.

10.37.2.7 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (Congruence_System & *cgs*, Recycle_Input *dummy*)

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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

10.37.2.8 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (const C_Polyhedron & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [explicit]

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

Parameters

y The C polyhedron to be used;

complexity This argument is ignored.

10.37.2.9 template<typename Interval > Parma_Polyhedra_Library::NNC_Polyhedron::NNC_-Polyhedron (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds an NNC polyhedron out of a box.

The polyhedron inherits the space dimension of the box and is the most precise that includes the box.

Parameters

box The box representing the polyhedron to be built;

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.37.2.10 Parma_Polyhedra_Library::NNC_Polyhedron::NNC_Polyhedron (const Grid & *grid*, Complexity_Class *complexity* = ANY_COMPLEXITY) [explicit]

Builds an NNC polyhedron out of a grid.

The polyhedron inherits the space dimension of the grid and is the most precise that includes the grid.

Parameters

grid The grid used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.37.2.11 template<typename U > Parma_Polyhedra_Library::NNC_Polyhedron::NNC_-Polyhedron (const BD_Shape< U > & *bd*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a NNC polyhedron out of a BD shape.

The polyhedron inherits the space dimension of the BD shape and is the most precise that includes the BD shape.

Parameters

bd The BD shape used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.37.2.12 template<typename U > Parma_Polyhedra_Library::NNC_Polyhedron::NNC_-Polyhedron (const Octagonal_Shape< U > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a NNC polyhedron out of an octagonal shape.

The polyhedron inherits the space dimension of the octagonal shape and is the most precise that includes the octagonal shape.

Parameters

os The octagonal shape used to build the polyhedron.

complexity This argument is ignored as the algorithm used has polynomial complexity.

10.37.3 Member Function Documentation

10.37.3.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.38 Parma_Polyhedra_Library::PIP_Solution_Node::No_Constraints Struct Reference

A tag type to select the alternative copy constructor.

`#include <ppl.hh>`

10.38.1 Detailed Description

A tag type to select the alternative copy constructor.

The documentation for this struct was generated from the following file:

- ppl.hh

10.39 Parma_Polyhedra_Library::No_Reduction< D1, D2 > Class Template Reference

This class provides the reduction method for the Direct_Product domain.

`#include <ppl.hh>`

Public Member Functions

- No_Reduction ()

  Default constructor.

- void product_reduce (D1 &d1, D2 &d2)

  The null reduction operator.

- ∼No_Reduction ()

  Destructor.

10.39.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::No_Reduction< D1, D2 >

This class provides the reduction method for the Direct_Product domain. The reduction classes are used to instantiate the Partially_Reduced_Product domain template parameter `R`. This class does no reduction at all.

10.39.2 Member Function Documentation

10.39.2.1 template<typename D1 , typename D2 > void Parma_Polyhedra_Library::No_Reduction< D1, D2 >::product_reduce (D1 & *d1*, D2 & *d2*) `[inline]`

The null reduction operator.

The parameters `d1` and `d2` are ignored.

The documentation for this class was generated from the following file:

- ppl.hh

10.40 Parma_Polyhedra_Library::Octagonal_Shape< T > Class Template Reference

An octagonal shape.

`#include <ppl.hh>`

Public Types

- typedef T coefficient_type_base

  The numeric base type upon which OSs are built.

- typedef N coefficient_type

  The (extended) numeric type of the inhomogeneous term of the inequalities defining an OS.

Public Member Functions

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- 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`.

- int32_t hash_code () const

  Returns a 32-bit hash code for `*this`.

Constructors, Assignment, Swap and Destructor

- Octagonal_Shape (dimension_type num_dimensions=0, Degenerate_Element kind=UNIVERSE)
  Builds an universe or empty OS of the specified space dimension.

- Octagonal_Shape (const Octagonal_Shape &x, Complexity_Class complexity=ANY_-COMPLEXITY)
  Ordinary copy constructor.

- template< typename U >
  Octagonal_Shape (const Octagonal_Shape< U > &y, Complexity_Class complexity=ANY_-COMPLEXITY)
  Builds a conservative, upward approximation of `y`.

- Octagonal_Shape (const Constraint_System &cs)
  Builds an OS from the system of constraints `cs`.

- Octagonal_Shape (const Congruence_System &cgs)
  Builds an OS from a system of congruences.

- Octagonal_Shape (const Generator_System &gs)
  Builds an OS from the system of generators `gs`.

- Octagonal_Shape (const Polyhedron &ph, Complexity_Class complexity=ANY_-COMPLEXITY)
  Builds an OS from the polyhedron `ph`.

- template< typename Interval >
  Octagonal_Shape (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)
  Builds an OS out of a box.

- Octagonal_Shape (const Grid &grid, Complexity_Class complexity=ANY_COMPLEXITY)
  Builds an OS that approximates a grid.

- template< typename U >
  Octagonal_Shape (const BD_Shape< U > &bd, Complexity_Class complexity=ANY_-COMPLEXITY)
  Builds an OS from a BD shape.

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

- void swap (Octagonal_Shape &y)
  Swaps `*this` *with octagon* `y`*. (*`*this` *and* `y` *can be dimension-incompatible.).*

- ~Octagonal_Shape ()
  Destructor.

Member Functions that Do Not Modify the Octagonal_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 the system of constraints defining `*this`*.*

- Constraint_System minimized_constraints () const

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

- Congruence_System congruences () const

  Returns a system of (equality) congruences satisfied by `*this`*.*

- Congruence_System minimized_congruences () const

  Returns a minimal system of (equality) congruences satisfied by `*this` *with the same affine dimension as* `*this`*.*

- bool contains (const Octagonal_Shape &y) const

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

- bool strictly_contains (const Octagonal_Shape &y) const

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

- bool is_disjoint_from (const Octagonal_Shape &y) const

  Returns `true` *if and only if* `*this` *and* `y` *are disjoint.*

- Poly_Con_Relation relation_with (const Constraint &c) const

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

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between `*this` *and the congruence* `cg`*.*

- 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 OS.*

- bool is_universe () const

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

- 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 a bounded OS.*

- bool is_topologically_closed () const

  Returns `true` *if and only if* `*this` *is a topologically closed subset of the vector space.*

- bool contains_integer_point () const

  Returns `true` *if and only if* `*this` *contains (at least) an integer point.*

- bool constrains (Variable var) const

  Returns `true` *if and only if* `var` *is constrained in* `*this`*.*

- 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 &g) 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 &g) 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 frequency (const Linear_Expression &expr, Coefficient &freq_n, Coefficient &freq_d, Coefficient &val_n, Coefficient &val_d) const

  Returns `true` *if and only if there exist a unique value* `val` *such that* `*this` *saturates the equality* `expr = val`.

- bool OK () const

  Checks if all the invariants are satisfied.

Space-Dimension Preserving Member Functions that May Modify the Octagonal_Shape

- void add_constraint (const Constraint &c)

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

- void add_constraints (const Constraint_System &cs)

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

- void add_recycled_constraints (Constraint_System &cs)

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

- void add_congruence (const Congruence &cg)

  Adds to `*this` *a constraint equivalent to the congruence* `cg`.

- void add_congruences (const Congruence_System &cgs)

  Adds to `*this` *constraints equivalent to the congruences in* `cgs`.

- void add_recycled_congruences (Congruence_System &cgs)

  Adds to `*this` *constraints equivalent to the congruences in* `cgs`.

- void refine_with_constraint (const Constraint &c)

  Uses a copy of constraint `c` *to refine the system of octagonal constraints defining* `*this`.

- void refine_with_congruence (const Congruence &cg)

  Uses a copy of congruence `cg` *to refine the system of octagonal constraints of* `*this`.

- void refine_with_constraints (const Constraint_System &cs)

  Uses a copy of the constraints in `cs` *to refine the system of octagonal constraints defining* `*this`.

- void refine_with_congruences (const Congruence_System &cgs)

  Uses a copy of the congruences in `cgs` *to refine the system of octagonal constraints defining* `*this`.

- void unconstrain (Variable var)

  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`.

- void unconstrain (const Variables_Set &vars)

  Computes the cylindrification of `*this` *with respect to the set of space dimensions* `vars`*, assigning the result to* `*this`.

- void intersection_assign (const Octagonal_Shape &y)

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

- void upper_bound_assign (const Octagonal_Shape &y)

  Assigns to `*this` *the smallest OS that contains the convex union of* `*this` *and* `y`.

- bool upper_bound_assign_if_exact (const Octagonal_Shape &y)

  If the upper bound of `*this` *and* `y` *is exact, it is assigned to* `*this` *and* `true` *is returned, otherwise* `false` *is returned.*

- bool integer_upper_bound_assign_if_exact (const Octagonal_Shape &y)

  If the integer upper bound of `*this` *and* `y` *is exact, it is assigned to* `*this` *and* `true` *is returned; otherwise* `false` *is returned.*

- void difference_assign (const Octagonal_Shape &y)

  Assigns to `*this` *the smallest octagon containing the set difference of* `*this` *and* `y`.

- bool simplify_using_context_assign (const Octagonal_Shape &y)

  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`*. If* `false` *is returned, then the intersection is empty.*

- 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 generalized affine transfer function* $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$*, where* $\bowtie$ *is the relation symbol encoded by* `relsym`.

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

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

- void 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 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 generalized_affine_preimage (const Linear_Expression &lhs, 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_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 Octagonal_Shape &y)

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

- void wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

  Wraps the specified dimensions of the vector space.

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates.*

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates for the space dimensions corresponding to* `vars`.

- void topological_closure_assign ()

  Assigns to `*this` *its topological closure.*

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

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

- void widening_assign (const Octagonal_Shape &y, unsigned *tp=0)

  Same as BHMZ05_widening_assign(y, tp).

- void limited_BHMZ05_extrapolation_assign (const Octagonal_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 Octagonal_Shape &y)

  Restores from `y` the constraints of `*this`, lost by CC76-extrapolation applications.

- void limited_CC76_extrapolation_assign (const Octagonal_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`.

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 OS into the new space.

- void add_space_dimensions_and_project (dimension_type m)

  Adds `m` new dimensions to the OS and does not embed it in the new space.

- void concatenate_assign (const Octagonal_Shape &y)

  Assigns to `*this` the concatenation of `*this` and `y`, taken in this order.

- void remove_space_dimensions (const Variables_Set &vars)

  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 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 &vars, Variable dest)

  Folds the space dimensions in `vars` into `dest`.

- template<typename Interval_Info >
  void refine_fp_interval_abstract_store (Box< Interval< T, Interval_Info > > &store) const

  Refines `store` with the constraints defining `*this`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension that an OS can handle.

- static bool can_recycle_constraint_systems ()

  Returns false indicating that this domain cannot recycle constraints.

- static bool can_recycle_congruence_systems ()

  Returns false indicating that this domain cannot recycle congruences.

Friends

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

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

Related Functions

(Note that these are not member functions.)

- template<typename T >
  std::ostream & operator<< (std::ostream &s, const Octagonal_Shape< T > &oct)

  Output operator.

- template<typename T >
  bool operator!= (const Octagonal_Shape< T > &x, const Octagonal_Shape< T > &y)

  Returns `true` *if and only if* `x` *and* `y` *are different shapes.*

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

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

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

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

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

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

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

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

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

Computes the L_∞ distance between x and y.

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

  Computes the L_∞ distance between x and y.

- template<typename T >
  void swap (Parma_Polyhedra_Library::Octagonal_Shape< T > &x, Parma_Polyhedra_-Library::Octagonal_Shape< T > &y)

  Specializes `std::swap`.

10.40.1 Detailed Description

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

An octagonal shape. The class template Octagonal_Shape<T> allows for the efficient representation of a restricted kind of *topologically closed* convex polyhedra called *octagonal shapes* (OSs, for short). The name comes from the fact that, in a vector space of dimension 2, bounded OSs are polygons with at most eight sides. The closed affine half-spaces that characterize the OS can be expressed by constraints of the form

$$ax_i + bx_j \leq k$$

where $a, b \in \{-1, 0, 1\}$ and k is a rational number, which are called *octagonal constraints*.

Based on the class template type parameter `T`, a family of extended numbers is built and used to approximate the inhomogeneous term of octagonal constraints. 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 OSs is meant to be as similar as possible to the one developed for the polyhedron class C_Polyhedron.

The OS domain *optimally supports*:

- tautological and inconsistent constraints and congruences;
- octagonal constraints;
- non-proper congruences (i.e., equalities) that are expressible as octagonal constraints.

Depending on the method, using a constraint or congruence that is not optimally supported by the domain will either raise an exception or result in a (possibly non-optimal) upward approximation.

A constraint is octagonal if it has the form

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

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 octagonal Constraint object will be mapped into a *correct* and *optimal* approximation that, depending on the expressive power of the chosen template argument T, may loose some precision. Also note that strict constraints are not octagonal.

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, depending from the context, a Constraint object encoding $3x - y \leq 1$ will be either upward approximated (e.g., by safely ignoring it) or it will cause an exception.

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 an OS corresponding to a cube in $\mathbb{R}^3$, 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);
cs.insert(z >= 0);
cs.insert(z <= 3);
Octagonal_Shape<T> oct(cs);
```

In contrast, the following code will raise an exception, since constraints 7, 8, and 9 are not octagonal:

```
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 3);
cs.insert(y >= 0);
cs.insert(y <= 3);
cs.insert(z >= 0);
cs.insert(z <= 3);
cs.insert(x - 3*y <= 5);     // (7)
cs.insert(x - y + z <= 5);   // (8)
cs.insert(x + y + z <= 5);   // (9)
Octagonal_Shape<T> oct(cs);
```

10.40.2 Constructor & Destructor Documentation

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

Builds an universe or empty OS of the specified space dimension.

Parameters

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

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

10.40.2.2 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Octagonal_Shape< T > & *x*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Ordinary copy constructor.

The complexity argument is ignored.

10.40.2.3 template<typename T > template<typename U > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Octagonal_Shape< U > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a conservative, upward approximation of `y`.

The complexity argument is ignored.

10.40.2.4 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Constraint_System & *cs*) [inline, explicit]

Builds an OS from the system of constraints `cs`.

The OS inherits the space dimension of `cs`.

Parameters

cs A system of octagonal constraints.

Exceptions

std::invalid_argument Thrown if `cs` contains a constraint which is not optimally supported by the Octagonal shape domain.

10.40.2.5 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Congruence_System & *cgs*) [inline, explicit]

Builds an OS from a system of congruences.

The OS inherits the space dimension of `cgs`

Parameters

cgs A system of congruences.

Exceptions

std::invalid_argument Thrown if `cgs` contains a congruence which is not optimally supported by the Octagonal shape domain.

10.40.2.6 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Generator_System & *gs*) `[inline, explicit]`

Builds an OS from the system of generators `gs`.

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

Exceptions

std::invalid_argument Thrown if the system of generators is not empty but has no points.

10.40.2.7 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) `[inline, explicit]`

Builds an OS from the polyhedron `ph`.

Builds an OS containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the OS built is the smallest one containing `ph`.

10.40.2.8 template<typename T > template<typename Interval > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) `[inline, explicit]`

Builds an OS out of a box.

The OS inherits the space dimension of the box. The built OS is the most precise OS that includes the box.

Parameters

box The box representing the OS to be built.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.40.2.9 template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Grid & *grid*, Complexity_Class *complexity* = ANY_COMPLEXITY) `[inline, explicit]`

Builds an OS that approximates a grid.

The OS inherits the space dimension of the grid. The built OS is the most precise OS that includes the grid.

Parameters

grid The grid used to build the OS.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `grid` exceeds the maximum allowed space dimension.

10.40.2.10 template<typename T > template<typename U > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const BD_Shape< U > & *bd*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds an OS from a BD shape.

The OS inherits the space dimension of the BD shape. The built OS is the most precise OS that includes the BD shape.

Parameters

bd The BD shape used to build the OS.

complexity This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

std::length_error Thrown if the space dimension of `bd` exceeds the maximum allowed space dimension.

10.40.3 Member Function Documentation

10.40.3.1 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::contains (const Octagonal_Shape< T > & *y*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.40.3.2 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::strictly_contains (const Octagonal_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.

10.40.3.3 template< typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::is_disjoint_from (const Octagonal_Shape< T > & *y*) const `[inline]`

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.

10.40.3.4 template< typename T > Poly_Con_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Constraint & *c*) const `[inline]`

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

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.40.3.5 template< typename T > Poly_Con_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Congruence & *cg*) const `[inline]`

Returns the relations holding between `*this` and the congruence `cg`.

Exceptions

std::invalid_argument Thrown if `*this` and `cg` are dimension-incompatible.

10.40.3.6 template< typename T > Poly_Gen_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Generator & *g*) const `[inline]`

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

Exceptions

std::invalid_argument Thrown if `*this` and generator `g` are dimension-incompatible.

10.40.3.7 template< typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::constrains (Variable *var*) const `[inline]`

Returns `true` if and only if `var` is constrained in `*this`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.40.3.8 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::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.

10.40.3.9 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::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.

10.40.3.10 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::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.

10.40.3.11 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.40.3.12 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::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.

10.40.3.13 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.40.3.14 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::frequency (const Linear_Expression & *expr*, Coefficient & *freq_n*, Coefficient & *freq_d*, Coefficient & *val_n*, Coefficient & *val_d*) const `[inline]`

Returns `true` if and only if there exist a unique value `val` such that `*this` saturates the equality `expr = val`.

Parameters

expr The linear expression for which the frequency is needed;

freq_n If `true` is returned, the value is set to 0; Present for interface compatibility with class Grid, where the frequency can have a non-zero value;

freq_d If `true` is returned, the value is set to 1;

val_n The numerator of `val`;

val_d The denominator of `val`;

Exceptions

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `false` is returned, then `freq_n`, `freq_d`, `val_n` and `val_d` are left untouched.

10.40.3.15 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_constraint (const Constraint & *c*) `[inline]`

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

Parameters

c The constraint to be added.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible, or `c` is not optimally supported by the OS domain.

10.40.3.16 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_constraints (const Constraint_System & *cs*) `[inline]`

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

Parameters

cs The constraints that will be added.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible, or `cs` contains a constraint which is not optimally supported by the OS domain.

10.40.3.17 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_recycled_constraints (Constraint_System & *cs*) `[inline]`

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

Parameters

cs The constraint system to be added to `*this`. The constraints in `cs` may be recycled.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible, or `cs` contains a constraint which is not optimally supported by the OS domain.

Warning

The only assumption that can be made on `cs` upon successful or exceptional return is that it can be safely destroyed.

10.40.3.18 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_congruence (const Congruence & *cg*) `[inline]`

Adds to `*this` a constraint equivalent to the congruence `cg`.

Parameters

cg The congruence to be added.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible, or `cg` is not optimally supported by the OS domain.

10.40.3.19 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_congruences (const Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

Parameters

cgs The congruences to be added.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the OS domain.

10.40.3.20 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_recycled_congruences (Congruence_System & *cgs*) [inline]

Adds to `*this` constraints equivalent to the congruences in `cgs`.

Parameters

cgs The congruence system to be added to `*this`. The congruences in `cgs` may be recycled.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible, or `cgs` contains a congruence which is not optimally supported by the OS domain.

Warning

The only assumption that can be made on `cgs` upon successful or exceptional return is that it can be safely destroyed.

10.40.3.21 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_constraint (const Constraint & *c*) [inline]

Uses a copy of constraint `c` to refine the system of octagonal constraints defining `*this`.

Parameters

c The constraint. If it is not a octagonal constraint, it will be ignored.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.40.3.22 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_congruence (const Congruence & *cg*) [inline]

Uses a copy of congruence `cg` to refine the system of octagonal constraints of `*this`.

Parameters

cg The congruence. If it is not a octagonal equality, it will be ignored.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

10.40.3.23 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_constraints (const Constraint_System & *cs*) [inline]

Uses a copy of the constraints in `cs` to refine the system of octagonal constraints defining `*this`.

Parameters

cs The constraint system to be used. Constraints that are not octagonal are ignored.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.40.3.24 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Uses a copy of the congruences in `cgs` to refine the system of octagonal constraints defining `*this`.

Parameters

cgs The congruence system to be used. Congruences that are not octagonal equalities are ignored.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.40.3.25 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::unconstrain (Variable *var*) [inline]

Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.40.3.26 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::unconstrain (const Variables_Set & *vars*) [inline]

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.40.3.27 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::intersection_assign (const Octagonal_Shape< T > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.40.3.28 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::upper_bound_assign (const Octagonal_Shape< T > & *y*) [inline]

Assigns to `*this` the smallest OS that contains the convex union of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.40.3.29 template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::upper_bound_assign_if_exact (const Octagonal_Shape< T > & *y*) [inline]

If the upper bound 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.

Implementation is based on Theorem 6.3 of [BHZ09b].

10.40.3.30 template< typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::integer_upper_bound_assign_if_exact (const Octagonal_Shape< T > & y) `[inline]`

If the *integer* upper bound 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.

Note

This operator is only available when the class template parameter `T` is bound to an integer datatype. The integer upper bound of two rational OS is the smallest rational OS containing all the integral points in the two arguments. In general, the result is *not* an upper bound for the two input arguments, as it may cut away non-integral portions of the two rational shapes.

Implementation is based on Theorem 6.8 of [BHZ09b].

10.40.3.31 template< typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::difference_assign (const Octagonal_Shape< T > & y) `[inline]`

Assigns to `*this` the smallest octagon containing the set difference of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.40.3.32 template< typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::simplify_using_context_assign (const Octagonal_Shape< T > & y) `[inline]`

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.40.3.33 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.40.3.34 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.40.3.35 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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

Parameters

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

relsym The relation symbol.

expr The numerator of the right hand side affine expression.

denominator The denominator of the right hand side affine expression.

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.

10.40.3.36 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) `[inline]`

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

Parameters

lhs The left hand side affine expression.

relsym The relation symbol.

rhs The right hand side affine expression.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with `lhs` or `rhs` or if `relsym` is a strict relation symbol.

10.40.3.37 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.40.3.38 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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.

10.40.3.39 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) [inline]

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 `relsym` is a strict relation symbol.

10.40.3.40 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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`.

10.40.3.41 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::time_elapse_assign (const Octagonal_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.

10.40.3.42 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System * *pcs* = `0`, unsigned *complexity_threshold* = `16`, bool *wrap_individually* = `true`) `[inline]`

Wraps the specified dimensions of the vector space.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be wrapped.

w The width of the bounded integer type corresponding to all the dimensions to be wrapped.

r The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

o The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.

pcs Possibly null pointer to a constraint system whose variables are contained in `vars`. If `*pcs` depends on variables not in `vars`, the behavior is undefined. When non-null, the pointed-to constraint system is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation with the constraints in `*pcs`.

complexity_threshold A precision parameter of the wrapping operator: higher values result in possibly improved precision.

wrap_individually `true` if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

std::invalid_argument Thrown if `*pcs` is dimension-incompatible with `vars`, or if `*this` is dimension-incompatible `vars` or with `*pcs`.

10.40.3.43 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.40.3.44 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.40.3.45 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::CC76_extrapolation_assign (const Octagonal_Shape< T > & *y*, unsigned * *tp* = 0) [inline]

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

Parameters

y An OS 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.

10.40.3.46 template<typename T > template<typename Iterator > void Parma_Polyhedra_Library::Octagonal_Shape< T >::CC76_extrapolation_assign (const Octagonal_Shape< T > & *y*, Iterator *first*, Iterator *last*, unsigned * *tp* = 0) `[inline]`

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

Parameters

y An OS that *must* be contained in `*this`.

first An iterator that points to the first stop_point.

last An iterator that points to 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.

10.40.3.47 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::BHMZ05_widening_assign (const Octagonal_Shape< T > & *y*, unsigned * *tp* = 0) `[inline]`

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

Parameters

y An OS 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.

10.40.3.48 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::limited_BHMZ05_extrapolation_assign (const Octagonal_Shape< T > & *y*, const Constraint_System & *cs*, unsigned * *tp* = 0) `[inline]`

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 An OS that *must* be contained in `*this`.

cs The system of constraints used to improve the widened OS.

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 there is in `cs` a strict inequality.

10.40.3.49 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::CC76_narrowing_assign (const Octagonal_Shape< T > & *y*) [inline]

Restores from `y` the constraints of `*this`, lost by CC76-extrapolation applications.

Parameters

y An OS that *must* contain `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.40.3.50 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::limited_CC76_extrapolation_assign (const Octagonal_Shape< T > & *y*, const Constraint_System & *cs*, unsigned * *tp* = 0) [inline]

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 An OS that *must* be contained in `*this`.

cs The system of constraints used to improve the widened OS.

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.

10.40.3.51 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_space_dimensions_and_embed (dimension_type *m*) [inline]

Adds m new dimensions and embeds the old OS 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 OS, 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 OS $\mathcal{O} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the OS

$$\{ (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{O} \}.$$

10.40.3.52 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_space_dimensions_and_project (dimension_type *m*) [inline]

Adds m new dimensions to the OS and does not embed it in the new space.

Parameters

m The number of dimensions to add.

The new dimensions will be those having the highest indexes in the new OS, 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 OS $\mathcal{O} \subseteq \mathbb{R}^2$ and adding a third dimension, the result will be the OS

$$\{ (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{O} \}.$$

10.40.3.53 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::concatenate_assign (const Octagonal_Shape< T > & *y*) [inline]

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()`.

10.40.3.54 template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::remove_space_dimensions (const Variables_Set & *vars*) [inline]

Removes all the specified dimensions.

Parameters

vars 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 `vars`.

10.40.3.55 template<typename T > void Parma_Polyhedra_Library::Octagonal_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`.

10.40.3.56 template<typename T > template<typename Partial_Function > void Parma_Polyhedra_Library::Octagonal_Shape< T >::map_space_dimensions (const Partial_Function & *pfunc*) [inline]

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 type parameter 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.

```
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.

10.40.3.57 template< typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::expand_space_dimension (Variable *var*, dimension_type *m*) [inline]

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 $n, n+1, \dots, n+m-1$.

10.40.3.58 template< typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*) [inline]

Folds the space dimensions in vars into dest.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 dest or with one of the Variable objects contained in vars. Also thrown if dest is contained in vars.

If *this has space dimension n, with $n > 0$, dest has space dimension $k \leq n$, vars is a set of variables whose maximum space dimension is also less than or equal to n, and dest is not a member of vars, then the space dimensions corresponding to variables in vars are folded into the k-th space dimension.

10.40.3.59 template< typename T > template< typename Interval_Info > void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_fp_interval_abstract_store (Box< Interval< T, Interval_Info > > & *store*) const [inline]

Refines store with the constraints defining *this.

Parameters

store The interval floating point abstract store to refine.

10.40.3.60 template<typename T > int32_t Parma_Polyhedra_Library::Octagonal_Shape< T >::hash_code () const [inline]

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.40.4 Friends And Related Function Documentation

10.40.4.1 template<typename T > bool operator== (const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*) [friend]

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

Note that `x` and `y` may be dimension-incompatible shapes: in this case, the value `false` is returned.

10.40.4.2 template<typename T > std::ostream & operator<< (std::ostream & *s*, const Octagonal_Shape< T > & *x*) [related]

Output operator.

Writes a textual representation of `oct` on `s`: `false` is written if `oct` is an empty polyhedron; `true` is written if `oct` is a universe polyhedron; a system of constraints defining `oct` is written otherwise, all constraints separated by ", ".

10.40.4.3 template<typename T > bool operator!= (const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*) [related]

Returns `true` if and only if `x` and `y` are different shapes.

Note that `x` and `y` may be dimension-incompatible shapes: in this case, the value `true` is returned.

10.40.4.4 template<typename To , typename T > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.40.4.5 template<typename Temp , typename To , typename T > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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`.

10.40.4.6 template<typename To , typename T > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_-Policy>.

10.40.4.7 template<typename Temp , typename To , typename T > bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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`.

10.40.4.8 template<typename To , typename T > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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>.

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<Temp, Extended_Number_Policy>.

10.40.4.9 template<typename Temp , typename To , typename T > bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Octagonal_Shape< T > & *x*, const Octagonal_Shape< T > & *y*, 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`.

10.40.4.10 template<typename T > void swap (Parma_Polyhedra_Library::Octagonal_Shape< T > & *x*, Parma_Polyhedra_Library::Octagonal_Shape< T > & *y*) `[related]`

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.41 Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > Class Template Reference

The partially reduced product of two abstractions.

`#include <ppl.hh>`

Public Member Functions

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

  Builds an object having the specified properties.

- Partially_Reduced_Product (const Congruence_System &cgs)

  Builds a pair, copying a system of congruences.

- Partially_Reduced_Product (Congruence_System &cgs)

  Builds a pair, recycling a system of congruences.

- Partially_Reduced_Product (const Constraint_System &cs)

  Builds a pair, copying a system of constraints.

- Partially_Reduced_Product (Constraint_System &cs)

  Builds a pair, recycling a system of constraints.

- Partially_Reduced_Product (const C_Polyhedron &ph, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a product, from a C polyhedron.

- Partially_Reduced_Product (const NNC_Polyhedron &ph, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a product, from an NNC polyhedron.

- Partially_Reduced_Product (const Grid &gr, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a product, from a grid.

- template< typename Interval >
  Partially_Reduced_Product (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a product out of a box.

- template< typename U >
  Partially_Reduced_Product (const BD_Shape< U > &bd, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a product out of a BD shape.

- template< typename U >
  Partially_Reduced_Product (const Octagonal_Shape< U > &os, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a product out of an octagonal shape.

- Partially_Reduced_Product (const Partially_Reduced_Product &y, Complexity_Class complexity=ANY_COMPLEXITY)

  Ordinary copy constructor.

- template< typename E1 , typename E2 , typename S >
  Partially_Reduced_Product (const Partially_Reduced_Product< E1, E2, S > &y, Complexity_Class complexity=ANY_COMPLEXITY)

Builds a conservative, upward approximation of `y`*.*

- Partially_Reduced_Product & operator= (const Partially_Reduced_Product &y)

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

- bool reduce () const

  Reduce.

Member Functions that Do Not Modify the Partially_Reduced_Product

- dimension_type space_dimension () const

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

- dimension_type affine_dimension () const

  Returns the minimum affine dimension (see also grid affine dimension) of the components of `*this`*.*

- const D1 & domain1 () const

  Returns a constant reference to the first of the pair.

- const D2 & domain2 () const

  Returns a constant reference to the second of the pair.

- Constraint_System constraints () const

  Returns a system of constraints which approximates `*this`*.*

- Constraint_System minimized_constraints () const

  Returns a system of constraints which approximates `*this`*, in reduced form.*

- Congruence_System congruences () const

  Returns a system of congruences which approximates `*this`*.*

- Congruence_System minimized_congruences () const

  Returns a system of congruences which approximates `*this`*, in reduced form.*

- Poly_Con_Relation relation_with (const Constraint &c) const

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

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between `*this` *and* `cg`*.*

- Poly_Gen_Relation relation_with (const Generator &g) const

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

- bool is_empty () const

  Returns `true` *if and only if either of the components of* `*this` *are empty.*

- bool is_universe () const

  Returns `true` *if and only if both of the components of* `*this` *are the universe.*

- bool is_topologically_closed () const

  Returns `true` *if and only if both of the components of* `*this` *are topologically closed subsets of the vector space.*

- bool is_disjoint_from (const Partially_Reduced_Product &y) const
  Returns `true` *if and only if* `*this` *and* `y` *are componentwise disjoint.*
- bool is_discrete () const
  Returns `true` *if and only if a component of* `*this` *is discrete.*
- bool is_bounded () const
  Returns `true` *if and only if a component of* `*this` *is bounded.*
- bool constrains (Variable var) const
  Returns `true` *if and only if* `var` *is constrained in* `*this`*.*
- 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, 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 i* `*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 Partially_Reduced_Product &y) const
  Returns `true` *if and only if each component of* `*this` *contains the corresponding component of* `y`*.*
- bool strictly_contains (const Partially_Reduced_Product &y) const
  Returns `true` *if and only if each component of* `*this` *strictly contains the corresponding component of* `y`*.*
- bool OK () const
  Checks if all the invariants are satisfied.

Space Dimension Preserving Member Functions that May Modify the Partially_Reduced_-Product

- void add_constraint (const Constraint &c)
  Adds constraint `c` *to* `*this`*.*

- void refine_with_constraint (const Constraint &c)
  Use the constraint `c` *to refine* `*this`.
- void add_congruence (const Congruence &cg)
  Adds a copy of congruence `cg` *to* `*this`.
- void refine_with_congruence (const Congruence &cg)
  Use the congruence `cg` *to refine* `*this`.
- void add_congruences (const Congruence_System &cgs)
  Adds a copy of the congruences in `cgs` *to* `*this`.
- void refine_with_congruences (const Congruence_System &cgs)
  Use the congruences in `cgs` *to refine* `*this`.
- void add_recycled_congruences (Congruence_System &cgs)
  Adds the congruences in `cgs` *to *this.*
- void add_constraints (const Constraint_System &cs)
  Adds a copy of the constraint system in `cs` *to* `*this`.
- void refine_with_constraints (const Constraint_System &cs)
  Use the constraints in `cs` *to refine* `*this`.
- void add_recycled_constraints (Constraint_System &cs)
  Adds the constraint system in `cs` *to* `*this`.
- void unconstrain (Variable var)
  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`.
- void unconstrain (const Variables_Set &vars)
  Computes the cylindrification of `*this` *with respect to the set of space dimensions* `vars`*, assigning the result to* `*this`.
- void intersection_assign (const Partially_Reduced_Product &y)
  Assigns to `*this` *the componentwise intersection of* `*this` *and* `y`.
- void upper_bound_assign (const Partially_Reduced_Product &y)
  Assigns to `*this` *an upper bound of* `*this` *and* `y` *computed on the corresponding components.*
- bool upper_bound_assign_if_exact (const Partially_Reduced_Product &y)
  Assigns to `*this` *an upper bound of* `*this` *and* `y` *computed on the corresponding components. If it is exact on each of the components of* `*this`*,* `true` *is returned, otherwise* `false` *is returned.*
- void difference_assign (const Partially_Reduced_Product &y)
  Assigns to `*this` *an approximation of the set-theoretic difference of* `*this` *and* `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, 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` (see also generalized affine relation.).*

- 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 generalized affine relation $\mathrm{var}' \bowtie \frac{\mathrm{expr}}{\mathrm{denominator}}$, *where* $\bowtie$ *is the relation symbol encoded by `relsym`. (see also generalized affine relation.).*

- 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 generalized affine relation $\mathrm{lhs}' \bowtie \mathrm{rhs}$, *where* $\bowtie$ *is the relation symbol encoded by `relsym`. (see also generalized affine relation.).*

- void generalized_affine_preimage (const Linear_Expression &lhs, 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`. (see also generalized affine relation.).*

- 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 Partially_Reduced_Product &y)

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

- void topological_closure_assign ()

  Assigns to `*this` its topological closure.

- void widening_assign (const Partially_Reduced_Product &y, unsigned *tp=NULL)

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

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` by dropping some points with non-integer coordinates.

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

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 components of `*this` in the new vector space.

- void add_space_dimensions_and_project (dimension_type m)
  Adds `m` new space dimensions and does not embed the components in the new vector space.
- void concatenate_assign (const Partially_Reduced_Product &y)
  Assigns to the first (resp., second) component of `*this` the "concatenation" of the first (resp., second) components of `*this` and `y`, taken in this order. See also Concatenating Polyhedra.
- void remove_space_dimensions (const Variables_Set &vars)
  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 &vars, Variable dest)
  Folds the space dimensions in `vars` into `dest`.

Miscellaneous Member Functions

- ~Partially_Reduced_Product ()
  Destructor.
- void swap (Partially_Reduced_Product &y)
  Swaps `*this` with product `y`. (`*this` and `y` can be dimension-incompatible.).
- void ascii_dump () const
  Writes to `std::cerr` an ASCII representation of `*this`.
- void ascii_dump (std::ostream &s) const
  Writes to `s` an ASCII representation of `*this`.
- void print () const
  Prints `*this` to `std::cerr` using `operator<<`.
- bool ascii_load (std::istream &s)
  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.
- 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`.
- int32_t hash_code () const
  Returns a 32-bit hash code for `*this`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension this product can handle.

Protected Types

- typedef D1 Domain1

  The type of the first component.

- typedef D2 Domain2

  The type of the second component.

Protected Member Functions

- void clear_reduced_flag () const

  Clears the reduced flag.

- void set_reduced_flag () const

  Sets the reduced flag.

- bool is_reduced () const

  Return `true` if and only if the reduced flag is set.

Protected Attributes

- D1 d1

  The first component.

- D2 d2

  The second component.

- bool reduced

  Flag to record whether the components are reduced with respect to each other and the reduction class.

Friends

- bool operator== (const Partially_Reduced_Product< D1, D2, R > &x, const Partially_Reduced_Product< D1, D2, R > &y)

  Returns `true` if and only if the components of `x` and `y` are pairwise equal.

Related Functions

(Note that these are not member functions.)

- template<typename D1 , typename D2 , typename R >
  std::ostream & operator<< (std::ostream &s, const Partially_Reduced_Product< D1, D2, R > &dp)

  Output operator.

- template<typename D1 , typename D2 , typename R >
  bool operator!= (const Partially_Reduced_Product< D1, D2, R > &x, const Partially_Reduced_Product< D1, D2, R > &y)

  Returns `true` *if and only if the components of* `x` *and* `y` *are not pairwise equal.*

- template<typename D1 , typename D2 , typename R >
  void swap (Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > &x, Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > &y)

  Specializes `std::swap`.

10.41.1 Detailed Description

template<typename D1, typename D2, typename R> class Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >

The partially reduced product of two abstractions.

Warning

At present, the supported instantiations for the two domain templates `D1` and `D2` are the simple pointset domains: `C_Polyhedron`, `NNC_Polyhedron`, `Grid`, `Octagonal_Shape<T>`, `BD_Shape<T>`, `Box<T>`.

An object of the class `Partially_Reduced_Product<D1, D2, R>` represents the (partially reduced) product of two pointset domains `D1` and `D2` where the form of any reduction is defined by the reduction class `R`.

Suppose D_1 and D_2 are two abstract domains with concretization functions: $\gamma_1 \colon D_1 \to \mathbb{R}^n$ and $\gamma_2 \colon D_2 \to \mathbb{R}^n$, respectively.

The partially reduced product $D = D_1 \times D_2$, for any reduction class `R`, has a concretization $\gamma \colon D \to \mathbb{R}^n$ where, if $d = (d_1, d_2) \in D$

$$\gamma(d) = \gamma_1(d_1) \cap \gamma_2(d_2).$$

The operations are defined to be the result of applying the corresponding operations on each of the components provided the product is already reduced by the reduction method defined by `R`. In particular, if `R` is the `No_Reduction<D1, D2>` class, then the class `Partially_Reduced_Product<D1, D2, R>` domain is the direct product as defined in [CC79].

How the results on the components are interpreted and combined depend on the specific test. For example, the test for emptiness will first make sure the product is reduced (using the reduction method provided by `R` if it is not already known to be reduced) and then test if either component is empty; thus, if `R` defines no reduction between its components and $d = (G, P) \in (\mathbb{G} \times \mathbb{P})$ is a direct product in one dimension where G denotes the set of numbers that are integral multiples of 3 while P denotes the set of numbers between

1 and 2, then an operation that tests for emptiness should return false. However, the test for the universe returns true if and only if the test `is_universe()` on both components returns true.

In all the examples it is assumed that the template `R` is the `No_Reduction<D1, D2>` class and 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 direct product of a Grid and NNC Polyhedron, corresponding to the positive 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);
Partially_Reduced_Product<Grid, NNC_Polyhedron, No_Reduction<D1, D2> >
  dp(cgs);
dp.add_constraint(x >= 0);
dp.add_constraint(y >= 0);
```

Example 2

The following code builds the same product in $\mathbb{R}^2$:

```
Partially_Reduced_Product<Grid, NNC_Polyhedron, No_Reduction<D1, D2> > dp(2);
dp.add_constraint(x >= 0);
dp.add_constraint(y >= 0);
dp.add_congruence((x %= 0) / 2);
dp.add_congruence((y %= 0) / 2);
```

Example 3

The following code will write "dp is empty":

```
Partially_Reduced_Product<Grid, NNC_Polyhedron, No_Reduction<D1, D2> > dp(1);
dp.add_congruence((x %= 0) / 2);
dp.add_congruence((x %= 1) / 2);
if (dp.is_empty())
  cout << "dp is empty." << endl;
else
  cout << "dp is not empty." << endl;
```

Example 4

The following code will write "dp is not empty":

```
Partially_Reduced_Product<Grid, NNC_Polyhedron, No_Reduction<D1, D2> > dp(1);
dp.add_congruence((x %= 0) / 2);
dp.add_constraint(x >= 1);
dp.add_constraint(x <= 1);
if (dp.is_empty())
  cout << "dp is empty." << endl;
else
  cout << "dp is not empty." << endl;
```

10.41.2 Constructor & Destructor Documentation

10.41.2.1 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (dimension_type *num_dimensions* = 0, Degenerate_Element *kind* = UNIVERSE) [inline, explicit]

Builds an object having the specified properties.

Parameters

num_dimensions The number of dimensions of the vector space enclosing the pair;
kind Specifies whether a universe or an empty pair has to be built.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.41.2.2 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Congruence_System & *cgs*) `[inline, explicit]`

Builds a pair, copying a system of congruences.

The pair inherits the space dimension of the congruence system.

Parameters

cgs The system of congruences to be approximated by the pair.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.41.2.3 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (Congruence_System & *cgs*) `[inline, explicit]`

Builds a pair, recycling a system of congruences.

The pair inherits the space dimension of the congruence system.

Parameters

cgs The system of congruences to be approximates by the pair. Its data-structures may be recycled to build the pair.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.41.2.4 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Constraint_System & *cs*) `[inline, explicit]`

Builds a pair, copying a system of constraints.

The pair inherits the space dimension of the constraint system.

Parameters

cs The system of constraints to be approximated by the pair.

Exceptions

std::length_error Thrown if `num_dimensions` exceeds the maximum allowed space dimension.

10.41.2.5 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (Constraint_System & *cs*) [inline, explicit]

Builds a pair, recycling a system of constraints.

The pair inherits the space dimension of the constraint system.

Parameters

cs The system of constraints to be approximated by the pair.

Exceptions

std::length_error Thrown if the space dimension of `cs` exceeds the maximum allowed space dimension.

10.41.2.6 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const C_Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a product, from a C polyhedron.

Builds a product containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the built product is the smallest one containing `ph`. The product inherits the space dimension of the polyhedron.

Parameters

ph The polyhedron to be approximated by the product.

complexity The complexity that will not be exceeded.

Exceptions

std::length_error Thrown if the space dimension of `ph` exceeds the maximum allowed space dimension.

10.41.2.7 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const NNC_Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a product, from an NNC polyhedron.

Builds a product containing `ph` using algorithms whose complexity does not exceed the one specified by `complexity`. If `complexity` is `ANY_COMPLEXITY`, then the built product is the smallest one containing `ph`. The product inherits the space dimension of the polyhedron.

Parameters

ph The polyhedron to be approximated by the product.

complexity The complexity that will not be exceeded.

Exceptions

std::length_error Thrown if the space dimension of `ph` exceeds the maximum allowed space dimension.

10.41.2.8 template<typename D1 , typename D2 , typename R > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Grid & *gr*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a product, from a grid.

Builds a product containing `gr`. The product inherits the space dimension of the grid.

Parameters

gr The grid to be approximated by the product.

complexity The complexity is ignored.

Exceptions

std::length_error Thrown if the space dimension of `gr` exceeds the maximum allowed space dimension.

10.41.2.9 template<typename D1 , typename D2 , typename R > template<typename Interval > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Builds a product out of a box.

Builds a product containing `box`. The product inherits the space dimension of the box.

Parameters

box The box representing the pair to be built.

complexity The complexity is ignored.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.41.2.10 template< typename D1 , typename D2 , typename R > template< typename U > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const BD_Shape< U > & *bd*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Builds a product out of a BD shape.

Builds a product containing bd. The product inherits the space dimension of the BD shape.

Parameters

bd The BD shape representing the product to be built.

complexity The complexity is ignored.

Exceptions

std::length_error Thrown if the space dimension of bd exceeds the maximum allowed space dimension.

10.41.2.11 template< typename D1 , typename D2 , typename R > template< typename U > Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Octagonal_Shape< U > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Builds a product out of an octagonal shape.

Builds a product containing os. The product inherits the space dimension of the octagonal shape.

Parameters

os The octagonal shape representing the product to be built.

complexity The complexity is ignored.

Exceptions

std::length_error Thrown if the space dimension of os exceeds the maximum allowed space dimension.

10.41.2.12 template< typename D1 , typename D2 , typename R > template< typename E1 , typename E2 , typename S > Parma_Polyhedra_Library::Partially_-Reduced_Product< D1, D2, R >::Partially_Reduced_Product (const Partially_Reduced_Product< E1, E2, S > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a conservative, upward approximation of y.

The complexity argument is ignored.

10.41.3 Member Function Documentation

10.41.3.1 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::is_disjoint_from (const Partially_Reduced_Product< D1, D2, R > & *y*) const [inline]

Returns `true` if and only if `*this` and `y` are componentwise disjoint.

Exceptions

std::invalid_argument Thrown if `x` and `y` are dimension-incompatible.

10.41.3.2 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::constrains (Variable *var*) const [inline]

Returns `true` if and only if `var` is constrained in `*this`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.41.3.3 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::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.

10.41.3.4 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::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.

10.41.3.5 template<typename D1 , typename D2 , typename R > bool Parma_-Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::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`.

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.

10.41.3.6 template<typename D1 , typename D2 , typename R > bool Parma_-Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::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 the supremum value can be reached in `this`.

point When maximization succeeds, will be assigned a generator 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.

10.41.3.7 template<typename D1 , typename D2 , typename R > bool Parma_- Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::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 i `*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 infimum value can be reached in `this`.

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.

10.41.3.8 template<typename D1 , typename D2 , typename R > bool Parma_- Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::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 the infimum value can be reached in `this`.

point When minimization succeeds, will be assigned a generator 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.

10.41.3.9 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::contains (const Partially_Reduced_Product< D1, D2, R > & y) const [inline]

Returns true if and only if each component of *this contains the corresponding component of y.

Exceptions

std::invalid_argument Thrown if *this and y are dimension-incompatible.

10.41.3.10 template<typename D1 , typename D2 , typename R > bool Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::strictly_contains (const Partially_Reduced_Product< D1, D2, R > & y) const [inline]

Returns true if and only if each component of *this strictly contains the corresponding component of y.

Exceptions

std::invalid_argument Thrown if *this and y are dimension-incompatible.

10.41.3.11 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_constraint (const Constraint & c) [inline]

Adds constraint c to *this.

Exceptions

std::invalid_argument Thrown if *this and c are dimension-incompatible.

10.41.3.12 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::refine_with_constraint (const Constraint & c) [inline]

Use the constraint c to refine *this.

Parameters

c The constraint to be used for refinement.

Exceptions

std::invalid_argument Thrown if *this and c are dimension-incompatible.

10.41.3.13 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_congruence (const Congruence & *cg*) [inline]

Adds a copy of congruence cg to *this.

Exceptions

std::invalid_argument Thrown if *this and congruence cg are dimension-incompatible.

10.41.3.14 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::refine_with_congruence (const Congruence & *cg*) [inline]

Use the congruence cg to refine *this.

Parameters

cg The congruence to be used for refinement.

Exceptions

std::invalid_argument Thrown if *this and cg are dimension-incompatible.

10.41.3.15 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_congruences (const Congruence_System & *cgs*) [inline]

Adds a copy of the congruences in cgs to *this.

Parameters

cgs The congruence system to be added.

Exceptions

std::invalid_argument Thrown if *this and cgs are dimension-incompatible.

10.41.3.16 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Use the congruences in cgs to refine *this.

Parameters

cgs The congruences to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.41.3.17 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::add_recycled_congruences (Congruence_System & *cgs*) `[inline]`

Adds the congruences in `cgs` to *this.

Parameters

cgs The congruence system to be added that may be recycled.

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.

10.41.3.18 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::add_constraints (const Constraint_System & *cs*) `[inline]`

Adds a copy of the constraint system in `cs` to `*this`.

Parameters

cs The constraint system to be added.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.41.3.19 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::refine_with_constraints (const Constraint_System & *cs*) `[inline]`

Use the constraints in `cs` to refine `*this`.

Parameters

cs The constraints to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.41.3.20 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_recycled_constraints (Constraint_System & *cs*) `[inline]`

Adds the constraint system in `cs` to `*this`.

Parameters

cs The constraint system to be added that may be recycled.

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.

10.41.3.21 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::unconstrain (Variable *var*) `[inline]`

Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.41.3.22 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::unconstrain (const Variables_Set & *vars*) `[inline]`

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.41.3.23 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::intersection_assign (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.41.3.24 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::upper_bound_assign (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

Assigns to `*this` an upper bound of `*this` and `y` computed on the corresponding components.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.41.3.25 template< typename D1 , typename D2 , typename R > bool Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::upper_bound_assign_if_exact (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

Assigns to `*this` an upper bound of `*this` and `y` computed on the corresponding components. If it is exact on each of the components of `*this`, `true` is returned, otherwise `false` is returned.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.41.3.26 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::difference_assign (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

Assigns to `*this` an approximation of the set-theoretic difference of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.41.3.27 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.41.3.28 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.41.3.29 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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` (see also generalized affine relation.).

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.

10.41.3.30 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`. (see also generalized affine relation.).

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.

10.41.3.31 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) `[inline]`

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`. (see also generalized affine relation.).

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.

10.41.3.32 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) `[inline]`

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`. (see also generalized affine relation.).

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.

10.41.3.33 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.41.3.34 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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`.

10.41.3.35 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::time_elapse_assign (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.41.3.36 template< typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::widening_assign (const Partially_Reduced_Product< D1, D2, R > & *y*, unsigned * *tp* = NULL) [inline]

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

This widening uses either the congruence or generator systems depending on which of the systems describing x and y are up to date and minimized.

Parameters

y A product 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.

10.41.3.37 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.41.3.38 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.41.3.39 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_space_dimensions_and_embed (dimension_type *m*) [inline]

Adds `m` new space dimensions and embeds the components of `*this` 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()`.

10.41.3.40 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_space_dimensions_and_project (dimension_type *m*) [inline]

Adds `m` new space dimensions and does not embed the components 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()`.

10.41.3.41 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::concatenate_assign (const Partially_Reduced_Product< D1, D2, R > & *y*) [inline]

Assigns to the first (resp., second) component of `*this` the "concatenation" of the first (resp., second) components of `*this` and `y`, taken in this order. See also Concatenating Polyhedra.

Exceptions

std::length_error Thrown if the concatenation would cause the vector space to exceed dimension `max_space_dimension()`.

10.41.3.42 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::remove_space_dimensions (const Variables_Set & *vars*) [inline]

Removes all the specified dimensions from the vector space.

Parameters

vars 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 `vars`.

10.41.3.43 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::remove_higher_space_dimensions (dimension_type *new_dimension*) `[inline]`

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`.

10.41.3.44 template<typename D1 , typename D2 , typename R > template<typename Partial_Function > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::map_space_dimensions (const Partial_Function & *pfunc*) `[inline]`

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 `*this`.

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

10.41.3.45 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::expand_space_dimension (Variable *var*, dimension_type *m*) `[inline]`

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 $n, n+1, \dots, n+m-1$.

10.41.3.46 template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_-Library::Partially_Reduced_Product< D1, D2, R >::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*) [inline]

Folds the space dimensions in `vars` into `dest`.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 `dest` or with one of the Variable objects contained in `vars`. Also thrown if `dest` is contained in `vars`.

If `*this` has space dimension n, with $n > 0$, `dest` has space dimension $k \leq n$, `vars` is a set of variables whose maximum space dimension is also less than or equal to n, and `dest` is not a member of `vars`, then the space dimensions corresponding to variables in `vars` are folded into the k-th space dimension.

10.41.3.47 template<typename D1 , typename D2 , typename R > int32_t Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::hash_code () const [inline]

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.41.4 Friends And Related Function Documentation

10.41.4.1 template<typename D1 , typename D2 , typename R > bool operator== (const Partially_Reduced_Product< D1, D2, R > & *x*, const Partially_Reduced_Product< D1, D2, R > & *y*) [friend]

Returns `true` if and only if the components of `x` and `y` are pairwise equal.

Note that `x` and `y` may be dimension-incompatible: in those cases, the value `false` is returned.

10.41.4.2 template<typename D1 , typename D2 , typename R > std::ostream & operator<< (std::ostream & *s*, const Partially_Reduced_Product< D1, D2, R > & *pd*) [related]

Output operator.

Writes a textual representation of `dp` on `s`.

10.41.4.3 template<typename D1 , typename D2 , typename R > bool operator!= (const Partially_Reduced_Product< D1, D2, R > & *x*, const Partially_Reduced_Product< D1, D2, R > & *y*) [related]

Returns `true` if and only if the components of `x` and `y` are not pairwise equal.

Note that `x` and `y` may be dimension-incompatible: in those cases, the value `true` is returned.

10.41.4.4 template<typename D1 , typename D2 , typename R > void swap (Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > & *x*, Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.42 Parma_Polyhedra_Library::PIP_Decision_Node Class Reference

A tree node representing a decision in the space of solutions.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::PIP_Tree_Node.

Public Member Functions

- virtual PIP_Tree_Node ∗ clone () const

  Returns a pointer to a dynamically-allocated copy of `*this`.

- virtual ∼PIP_Decision_Node ()

  Destructor.

- virtual bool OK () const

  Returns `true` *if and only if* `*this` *is well formed.*

- virtual const PIP_Decision_Node ∗ as_decision () const

  Returns `this`.

- const PIP_Tree_Node * child_node (bool b) const

  Returns a const pointer to the `b` *(true or false) branch of* `*this`*.*

- PIP_Tree_Node * child_node (bool b)

  Returns a pointer to the `b` *(true or false) branch of* `*this`*.*

- void ascii_dump (std::ostream &s) const

  Dumps to `s` *an ASCII representation of* `*this`*.*

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- virtual memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`*.*

- virtual memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`*.*

Protected Member Functions

- PIP_Decision_Node (const PIP_Decision_Node &y)

  Copy constructor.

- virtual void update_tableau (const PIP_Problem &pip, dimension_type external_space_dim, dimension_type first_pending_constraint, const Constraint_Sequence &input_cs, const Variables_-Set &parameters)

  Implements pure virtual method PIP_Tree_Node::update_tableau.

- virtual PIP_Tree_Node * solve (const PIP_Problem &pip, bool check_feasible_context, const Matrix &context, const Variables_Set &params, dimension_type space_dim, unsigned indent_level)

  Implements pure virtual method PIP_Tree_Node::solve.

- virtual void print_tree (std::ostream &s, unsigned indent, const std::vector< bool > &pip_dim_is_-param, dimension_type first_art_dim) const

  Prints on `s` *the tree rooted in* `*this`*.*

10.42.1 Detailed Description

A tree node representing a decision in the space of solutions.

The documentation for this class was generated from the following file:

- ppl.hh

10.43 Parma_Polyhedra_Library::PIP_Problem Class Reference

A Parametric Integer (linear) Programming problem.

`#include <ppl.hh>`

Public Types

- enum Control_Parameter_Name { CUTTING_STRATEGY, PIVOT_ROW_STRATEGY }

  Possible names for PIP_Problem control parameters.

- enum Control_Parameter_Value {
  CUTTING_STRATEGY_FIRST, CUTTING_STRATEGY_DEEPEST, CUTTING_STRATEGY_-ALL, PIVOT_ROW_STRATEGY_FIRST,
  PIVOT_ROW_STRATEGY_MAX_COLUMN }

  Possible values for PIP_Problem control parameters.

- typedef Constraint_Sequence::const_iterator const_iterator

  A type alias for the read-only iterator on the constraints defining the feasible region.

Public Member Functions

- PIP_Problem (dimension_type dim=0)

  Builds a trivial PIP problem.

- template<typename In >
  PIP_Problem (dimension_type dim, In first, In last, const Variables_Set &p_vars)

  Builds a PIP problem having space dimension `dim` *from the sequence of constraints in the range* $[\mathrm{first}, \mathrm{last})$*; those dimensions whose indices occur in* `p_vars` *are interpreted as parameters.*

- PIP_Problem (const PIP_Problem &y)

  Ordinary copy-constructor.

- ∼PIP_Problem ()

  Destructor.

- PIP_Problem & operator= (const PIP_Problem &y)

  Assignment operator.

- dimension_type space_dimension () const

  Returns the space dimension of the PIP problem.

- const Variables_Set & parameter_space_dimensions () const

  Returns a set containing all the variables' indexes representing the parameters of the PIP problem.

- const_iterator constraints_begin () const

  Returns a read-only iterator to the first constraint defining the feasible region.

- const_iterator constraints_end () const

  Returns a past-the-end read-only iterator to the sequence of constraints defining the feasible region.

- void clear ()

  Resets `*this` *to be equal to the trivial PIP problem.*

- void add_space_dimensions_and_embed (dimension_type m_vars, dimension_type m_params)

Adds `m_vars + m_params` new space dimensions and embeds the old PIP problem in the new vector space.

- void add_to_parameter_space_dimensions (const Variables_Set &p_vars)

  Sets the space dimensions whose indexes which are in set `p_vars` to be parameter space dimensions.

- void add_constraint (const Constraint &c)

  Adds a copy of constraint `c` to the PIP problem.

- void add_constraints (const Constraint_System &cs)

  Adds a copy of the constraints in `cs` to the PIP problem.

- bool is_satisfiable () const

  Checks satisfiability of `*this`.

- PIP_Problem_Status solve () const

  Optimizes the PIP problem.

- PIP_Tree solution () const

  Returns a feasible solution for `*this`, if it exists.

- PIP_Tree optimizing_solution () const

  Returns an optimizing solution for `*this`, if it exists.

- bool OK () const

  Checks if all the invariants are satisfied.

- void print_solution (std::ostream &s, unsigned indent=0) const

  Prints on `s` the solution computed for `*this`.

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- 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 (PIP_Problem &y)

Swaps `*this` with `y`.

- Control_Parameter_Value get_control_parameter (Control_Parameter_Name name) const

  Returns the value of control parameter `name`.

- void set_control_parameter (Control_Parameter_Value value)

  Sets control parameter `value`.

- void set_big_parameter_dimension (dimension_type big_dim)

  Sets the dimension for the big parameter to `big_dim`.

- dimension_type get_big_parameter_dimension () const

  Returns the space dimension for the big parameter.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a PIP_Problem can handle.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const PIP_Problem &p)

  Output operator.

- void swap (Parma_Polyhedra_Library::PIP_Problem &x, Parma_Polyhedra_Library::PIP_Problem &y)

  Specializes `std::swap`.

10.43.1 Detailed Description

A Parametric Integer (linear) Programming problem. An object of this class encodes a parametric integer (linear) programming problem. The PIP problem is specified by providing:

- the dimension of the vector space;
- the subset of those dimensions of the vector space that are interpreted as integer parameters (the other space dimensions are interpreted as non-parameter integer variables);
- a finite set of linear equality and (strict or non-strict) inequality constraints involving variables and/or parameters; these constraints are used to define:
  - the *feasible region*, if they involve one or more problem variable (and maybe some parameters);
  - the *initial context*, if they only involve the parameters;
- optionally, the so-called *big parameter*, i.e., a problem parameter to be considered arbitrarily big.

Note that all problem variables and problem parameters are assumed to take non-negative integer values, so that there is no need to specify non-negativity constraints.

The class provides support for the (incremental) solution of the PIP problem based on variations of the revised simplex method and on Gomory cut generation techniques.

The solution for a PIP problem is the lexicographic minimum of the integer points of the feasible region, expressed in terms of the parameters. As the problem to be solved only involves non-negative variables and parameters, the problem will always be either unfeasible or optimizable.

As the feasibility and the solution value of a PIP problem depend on the values of the parameters, the solution is a binary decision tree, dividing the context parameter set into subsets. The tree nodes are of two kinds:

- *Decision* nodes. These are internal tree nodes encoding one or more linear tests on the parameters; if all the tests are satisfied, then the solution is the node's *true* child; otherwise, the solution is the node's *false* child;
- *Solution* nodes. These are leaf nodes in the tree, encoding the solution of the problem in the current context subset, where each variable is defined in terms of a linear expression of the parameters. Solution nodes also optionally embed a set of parameter constraints: if all these constraints are satisfied, the solution is described by the node, otherwise the problem has no solution.

It may happen that a decision node has no *false* child. This means that there is no solution if at least one of the corresponding constraints is not satisfied. Decision nodes having two or more linear tests on the parameters cannot have a *false* child. Decision nodes always have a *true* child.

Both kinds of tree nodes may also contain the definition of extra parameters which are artificially introduced by the solver to enforce an integral solution. Such artificial parameters are defined by the integer division of a linear expression on the parameters by an integer coefficient.

By exploiting the incremental nature of the solver, it is possible to reuse part of the computational work already done when solving variants of a given PIP_Problem: currently, incremental resolution supports the addition of space dimensions, the addition of parameters and the addition of constraints.

Example problem

An example PIP problem can be defined the following:

```
3*j >= -2*i+8
j <= 4*i - 4
i <= n
j <= m
```

where `i` and `j` are the problem variables and `n` and `m` are the problem parameters. This problem can be optimized; the resulting solution tree may be represented as follows:

```
if 7*n >= 10 then
  if 7*m >= 12 then
    {i = 2 ; j = 2}
  else
    Parameter P = (m) div 2
    if 2*n + 3*m >= 8 then
      {i = -m - P + 4 ; j = m}
    else
      _|_
else
  _|_
```

The solution tree starts with a decision node depending on the context constraint `7*n >= 10`. If this constraint is satisfied by the values assigned to the problem parameters, then the (textually first) `then` branch is taken, reaching the *true* child of the root node (which in this case is another decision node); otherwise, the (textually last) `else` branch is taken, for which there is no corresponding *false* child.

The $\perp$ notation, also called *bottom*, denotes the lexicographic minimum of an empty set of solutions, here meaning the corresponding subproblem is unfeasible.

Notice that a tree node may introduce new (non-problem) parameters, as is the case for parameter `P` in the (textually first) `else` branch above. These *artificial* parameters are only meaningful inside the subtree where they are defined and are used to define the parametric values of the problem variables in solution nodes (e.g., the `{i, j}` vector in the textually third `then` branch).

Context restriction

The above solution is correct in an unrestricted initial context, meaning all possible values are allowed for the parameters. If we restrict the context with the following parameter inequalities:

```
m >= n
n >= 5
```

then the resulting optimizing tree will be a simple solution node:

```
{i = 2 ; j = 2}
```

Creating the PIP_Problem object

The PIP_Problem object corresponding to the above example can be created as follows:

```
Variable i(0);
Variable j(1);
Variable n(2);
Variable m(3);
Variables_Set params(n, m);
Constraint_System cs;
cs.insert(3*j >= -2*i+8);
cs.insert(j <= 4*i - 4);
cs.insert(j <= m);
cs.insert(i <= n);
PIP_Problem pip(cs.space_dimension(), cs.begin(), cs.end(), params);
```

If you want to restrict the initial context, simply add the parameter constraints the same way as for normal constraints.

```
cs.insert(m >= n);
cs.insert(n >= 5);
```

Solving the problem

Once the PIP_Problem object has been created, you can start the resolution of the problem by calling the solve() method:

```
PIP_Problem_Status status = pip.solve();
```

where the returned `status` indicates if the problem has been optimized or if it is unfeasible for any possible configuration of the parameter values. The resolution process is also started if an attempt is made to get its solution, as follows:

```
const PIP_Tree_Node* node = pip.solution();
```

In this case, an unfeasible problem will result in an empty solution tree, i.e., assigning a null pointer to `node`.

Printing the solution tree

A previously computed solution tree may be printed as follows:

```
pip.print_solution(std::cout);
```

This will produce the following output (note: variables and parameters are printed according to the default output function; see Variable::set_output_function):

```
if 7*C >= 10 then
  if 7*D >= 12 then
    {2 ; 2}
  else
    Parameter E = (D) div 2
    if 2*C + 3*D >= 8 then
      {-D - E + 4 ; D}
    else
      _|_
else
  _|_
```

Spanning the solution tree

A parameter assignment for a PIP problem binds each of the problem parameters to a non-negative integer value. After fixing a parameter assignment, the "spanning" of the PIP problem solution tree refers to the process whereby the solution tree is navigated, starting from the root node: the value of artificial parameters is computed according to the parameter assignment and the node's contraints are evaluated, thereby descending in either the true or the false subtree of decision nodes and eventually reaching a solution node or a bottom node. If a solution node is found, each of the problem variables is provided with a parametric expression, which can be evaluated to a fixed value using the given parameter assignment and the computed values for artificial parameters.

The coding of the spanning process can be done as follows. First, the root of the PIP solution tree is retrieved:

```
const PIP_Tree_Node* node = pip.solution();
```

If `node` represents an unfeasible solution (i.e., $\perp$), its value will be `0`. For a non-null tree node, the virtual methods `PIP_Tree_Node::as_decision()` and `PIP_Tree_Node::as_solution()` can be used to check whether the node is a decision or a solution node:

```
const PIP_Solution_Node* sol = node->as_solution();
if (sol != 0) {
  // The node is a solution node
  ...
}
else {
  // The node is a decision node
  const PIP_Decision_Node* dec = node->as_decision();
  ...
}
```

The true (resp., false) child node of a Decision Node may be accessed by using method `PIP_Decision_Node::child_node(bool)`, passing `true` (resp., `false`) as the input argument.

Artificial parameters

A PIP_Tree_Node::Artificial_Parameter object represents the result of the integer division of a Linear_Expression (on the other parameters, including the previously-defined artificials) by an integer denominator (a Coefficient object). The dimensions of the artificial parameters (if any) in a tree node have consecutive indices starting from `dim+1`, where the value of `dim` is computed as follows:

- for the tree root node, `dim` is the space dimension of the PIP_Problem;
- for any other node of the tree, it is recusrively obtained by adding the value of `dim` computed for the parent node to the number of artificial parameters defined in the parent node.

Since the numbering of dimensions for artificial parameters follows the rule above, the addition of new problem variables and/or new problem parameters to an already solved PIP_Problem object (as done when incrementally solving a problem) will result in the systematic renumbering of all the existing artificial parameters.

Node constraints

All kind of tree nodes can contain context constraints. Decision nodes always contain at least one of them. The node's local constraint system can be obtained using method PIP_Tree_Node::constraints. These constraints only involve parameters, including both the problem parameters and the artificial parameters that have been defined in nodes occurring on the path from the root node to the current node. The meaning of these constraints is as follows:

- On a decision node, if all tests in the constraints are true, then the solution is the *true* child; otherwise it is the *false* child.
- On a solution node, if the (possibly empty) system of constraints evaluates to true for a given parameter assignment, then the solution is described by the node; otherwise the solution is $\perp$ (i.e., the problem is unfeasible for that parameter assignment).

Getting the optimal values for the variables

After spanning the solution tree using the given parameter assignment, if a solution node has been reached, then it is possible to retrieve the parametric expression for each of the problem variables using method PIP_Solution_Node::parametric_values. The retrieved expression will be defined in terms of all the parameters (problem parameters and artificial parameters defined along the path).

Solving maximization problems

You can solve a lexicographic maximization problem by reformulating its constraints using variable substitution. Proceed the following steps:

- Create a big parameter (see PIP_Problem::set_big_parameter_dimension), which we will call M.
- Reformulate each of the maximization problem constraints by substituting each x_i variable with an expression of the form $M - x'_i$, where the x'_i variables are positive variables to be minimized.
- Solve the lexicographic minimum for the x' variable vector.
- In the solution expressions, the values of the x' variables will be expressed in the form: $x'_i = M - x_i$. To get back the value of the expression of each x_i variable, just apply the formula: $x_i = M - x'_i$.

Note that if the resulting expression of one of the x'_i variables is not in the $x'_i = M - x_i$ form, this means that the sign-unrestricted problem is unbounded.

You can choose to maximize only a subset of the variables while minimizing the other variables. In that case, just apply the variable substitution method on the variables you want to be maximized. The variable optimization priority will still be in lexicographic order.

Example: consider you want to find the lexicographic maximum of the (x, y) vector, under the constraints:

$$\begin{cases} y \geq 2x - 4 \\ y \leq -x + p \end{cases}$$

where p is a parameter.

After variable substitution, the constraints become:

$$\begin{cases} M - y \geq 2M - 2x - 4 \\ M - y \leq -M + x + p \end{cases}$$

The code for creating the corresponding problem object is the following:

```
Variable x(0);
Variable y(1);
Variable p(2);
Variable M(3);
Variables_Set params(p, M);
Constraint_System cs;
cs.insert(M - y >= 2*M - 2*x - 4);
cs.insert(M - y <= -M + x + p);
PIP_Problem pip(cs.space_dimension(), cs.begin(), cs.end(), params);
pip.set_big_parameter_dimension(3);     // M is the big parameter
```

Solving the problem provides the following solution:

```
Parameter E = (C + 1) div 3
{D - E - 1 ; -C + D + E + 1}
```

Under the notations above, the solution is:

$$\begin{cases} x' = M - \left\lfloor \frac{p+1}{3} \right\rfloor - 1 \\ y' = M - p + \left\lfloor \frac{p+1}{3} \right\rfloor + 1 \end{cases}$$

Performing substitution again provides us with the values of the original variables:

$$\begin{cases} x = \left\lfloor \frac{p+1}{3} \right\rfloor + 1 \\ y = p - \left\lfloor \frac{p+1}{3} \right\rfloor - 1 \end{cases}$$

Allowing variables to be arbitrarily signed

You can deal with arbitrarily signed variables by reformulating the constraints using variable substitution. Proceed the following steps:

- Create a big parameter (see PIP_Problem::set_big_parameter_dimension), which we will call M.
- Reformulate each of the maximization problem constraints by substituting each x_i variable with an expression of the form $x'_i - M$, where the x'_i variables are positive.
- Solve the lexicographic minimum for the x' variable vector.
- The solution expression can be read in the form:
- In the solution expressions, the values of the x' variables will be expressed in the form: $x'_i = x_i + M$. To get back the value of the expression of each signed x_i variable, just apply the formula: $x_i = x'_i - M$.

Note that if the resulting expression of one of the x'_i variables is not in the $x'_i = x_i + M$ form, this means that the sign-unrestricted problem is unbounded.

You can choose to define only a subset of the variables to be sign-unrestricted. In that case, just apply the variable substitution method on the variables you want to be sign-unrestricted.

Example: consider you want to find the lexicographic minimum of the (x, y) vector, where the x and y variables are sign-unrestricted, under the constraints:

$$\begin{cases} y \geq -2x - 4 \\ 2y \leq x + 2p \end{cases}$$

where p is a parameter.

After variable substitution, the constraints become:

$$\begin{cases} y' - M \geq -2x' + 2M - 4 \\ 2y' - 2M \leq x' - M + 2p \end{cases}$$

The code for creating the corresponding problem object is the following:

```
Variable x(0);
Variable y(1);
Variable p(2);
Variable M(3);
Variables_Set params(p, M);
Constraint_System cs;
cs.insert(y - M >= -2*x + 2*M - 4);
cs.insert(2*y - 2*M <= x - M + 2*p);
PIP_Problem pip(cs.space_dimension(), cs.begin(), cs.end(), params);
pip.set_big_parameter_dimension(3);     // M is the big parameter
```

Solving the problem provides the following solution:

```
Parameter E = (2*C + 3) div 5
{D - E - 1 ; D + 2*E - 2}
```

Under the notations above, the solution is:

$$\begin{cases} x' = M - \left\lfloor \frac{2p+3}{5} \right\rfloor - 1 \\ y' = M + 2\left\lfloor \frac{2p+3}{5} \right\rfloor - 2 \end{cases}$$

Performing substitution again provides us with the values of the original variables:

$$\begin{cases} x = -\left\lfloor \frac{2p+3}{5} \right\rfloor - 1 \\ y = 2\left\lfloor \frac{2p+3}{5} \right\rfloor - 2 \end{cases}$$

Allowing parameters to be arbitrarily signed

You can consider a parameter p arbitrarily signed by replacing p with $p^+ - p^-$, where both p^+ and p^- are positive parameters. To represent a set of arbitrarily signed parameters, replace each parameter p_i with $p_i^+ - p^-$, where $-p^-$ is the minimum negative value of all parameters.

Minimizing a linear cost function

Lexicographic solving can be used to find the parametric minimum of a linear cost function.

Suppose the variables are named $x_1, x_2, \ldots, x_n$, and the parameters $p_1, p_2, \ldots, p_m$. You can minimize a linear cost function $f(x_2, \ldots, x_n, p_1, \ldots, p_m)$ by simply adding the constraint $x_1 \geq f(x_2, \ldots, x_n, p_1, \ldots, p_m)$ to the constraint system. As lexicographic minimization ensures x_1 is minimized in priority, and because x_1 is forced by a constraint to be superior or equal to the cost function, optimal solutions of the problem necessarily ensure that the solution value of x_1 is the optimal value of the cost function.

10.43.2 Member Enumeration Documentation

10.43.2.1 enum Parma_Polyhedra_Library::PIP_Problem::Control_Parameter_Name

Possible names for PIP_Problem control parameters.

Enumerator:

CUTTING_STRATEGY Cutting strategy.

PIVOT_ROW_STRATEGY Pivot row strategy.

10.43.2.2 enum Parma_Polyhedra_Library::PIP_Problem::Control_Parameter_Value

Possible values for PIP_Problem control parameters.

Enumerator:

CUTTING_STRATEGY_FIRST Choose the first non-integer row.

CUTTING_STRATEGY_DEEPEST Choose row which generates the deepest cut.

CUTTING_STRATEGY_ALL Always generate all possible cuts.

PIVOT_ROW_STRATEGY_FIRST Choose the first row with negative parameter sign.

PIVOT_ROW_STRATEGY_MAX_COLUMN Choose the row which generates the lexico-maximal pivot column.

10.43.3 Constructor & Destructor Documentation

10.43.3.1 Parma_Polyhedra_Library::PIP_Problem::PIP_Problem (dimension_type *dim* = 0) `[explicit]`

Builds a trivial PIP problem.

A trivial PIP problem requires to compute the lexicographic minimum on a vector space under no constraints and with no parameters: due to the implicit non-negativity constraints, the origin of the vector space is an optimal solution.

Parameters

dim The dimension of the vector space enclosing *this (optional argument with default value 0).

Exceptions

std::length_error Thrown if dim exceeds max_space_dimension().

10.43.3.2 template<typename In > Parma_Polyhedra_Library::PIP_Problem::PIP_Problem (dimension_type *dim*, In *first*, In *last*, const Variables_Set & *p_vars*) [inline]

Builds a PIP problem having space dimension dim from the sequence of constraints in the range $[\mathrm{first}, \mathrm{last})$; those dimensions whose indices occur in p_vars are interpreted as parameters.

Parameters

dim The dimension of the vector space (variables and parameters) enclosing *this.

first An input iterator to the start of the sequence of constraints.

last A past-the-end input iterator to the sequence of constraints.

p_vars The set of variables' indexes that are interpreted as parameters.

Exceptions

std::length_error Thrown if dim exceeds max_space_dimension().

std::invalid_argument Thrown if the space dimension of a constraint in the sequence (resp., the parameter variables) is strictly greater than dim.

10.43.4 Member Function Documentation

10.43.4.1 void Parma_Polyhedra_Library::PIP_Problem::clear ()

Resets *this to be equal to the trivial PIP problem.

The space dimension is reset to 0.

10.43.4.2 void Parma_Polyhedra_Library::PIP_Problem::add_space_dimensions_and_embed (dimension_type *m_vars*, dimension_type *m_params*)

Adds m_vars + m_params new space dimensions and embeds the old PIP problem in the new vector space.

Parameters

m_vars The number of space dimensions to add that are interpreted as PIP problem variables (i.e., non parameters). These are added *before* adding the m_params parameters.

m_params The number of space dimensions to add that are interpreted as PIP problem parameters. These are added *after* having added the m_vars problem variables.

Exceptions

std::length_error Thrown if adding `m_vars + m_params` 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 PIP problem; they are initially unconstrained.

10.43.4.3 void Parma_Polyhedra_Library::PIP_Problem::add_to_parameter_space_dimensions (const Variables_Set & *p_vars*)

Sets the space dimensions whose indexes which are in set `p_vars` to be parameter space dimensions.

Exceptions

std::invalid_argument Thrown if some index in `p_vars` does not correspond to a space dimension in `*this`.

10.43.4.4 void Parma_Polyhedra_Library::PIP_Problem::add_constraint (const Constraint & *c*)

Adds a copy of constraint `c` to the PIP problem.

Exceptions

std::invalid_argument Thrown if the space dimension of `c` is strictly greater than the space dimension of `*this`.

10.43.4.5 void Parma_Polyhedra_Library::PIP_Problem::add_constraints (const Constraint_System & *cs*)

Adds a copy of the constraints in `cs` to the PIP problem.

Exceptions

std::invalid_argument Thrown if the space dimension of constraint system `cs` is strictly greater than the space dimension of `*this`.

10.43.4.6 bool Parma_Polyhedra_Library::PIP_Problem::is_satisfiable () const

Checks satisfiability of `*this`.

Returns

`true` if and only if the PIP problem is satisfiable.

10.43.4.7 PIP_Problem_Status Parma_Polyhedra_Library::PIP_Problem::solve () const

Optimizes the PIP problem.

Returns

A PIP_Problem_Status flag indicating the outcome of the optimization attempt (unfeasible or optimized problem).

10.43.4.8 PIP_Tree Parma_Polyhedra_Library::PIP_Problem::solution () const

Returns a feasible solution for `*this`, if it exists.

A null pointer is returned for an unfeasible PIP problem.

10.43.4.9 PIP_Tree Parma_Polyhedra_Library::PIP_Problem::optimizing_solution () const

Returns an optimizing solution for `*this`, if it exists.

A null pointer is returned for an unfeasible PIP problem.

10.43.4.10 void Parma_Polyhedra_Library::PIP_Problem::print_solution (std::ostream & *s*, unsigned *indent* = 0) const

Prints on `s` the solution computed for `*this`.

Parameters

s The output stream.

indent An indentation parameter (default value 0).

Exceptions

std::logic_error Thrown if trying to print the solution when the PIP problem still has to be solved.

10.43.4.11 dimension_type Parma_Polyhedra_Library::PIP_Problem::get_big_parameter_-dimension () const [inline]

Returns the space dimension for the big parameter.

If a big parameter was not set, returns `not_a_dimension()`.

10.43.5 Friends And Related Function Documentation

10.43.5.1 std::ostream & operator<< (std::ostream & *s*, const PIP_Problem & *p*) [related]

Output operator.

10.43.5.2 void swap (Parma_Polyhedra_Library::PIP_Problem & *x*, Parma_Polyhedra_Library::PIP_Problem & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.44 Parma_Polyhedra_Library::PIP_Solution_Node Class Reference

A tree node representing part of the space of solutions.

`#include <ppl.hh>`

Inherits Parma_Polyhedra_Library::PIP_Tree_Node.

Classes

- struct No_Constraints

  A tag type to select the alternative copy constructor.

Public Member Functions

- PIP_Solution_Node (const PIP_Problem ∗owner)

  Constructor: builds a solution node owned by `*owner`.

- virtual PIP_Tree_Node ∗ clone () const

  Returns a pointer to a dynamically-allocated copy of `*this`.

- virtual ∼PIP_Solution_Node ()

  Destructor.

- virtual bool OK () const

  Returns `true` *if and only if* `*this` *is well formed.*

- virtual const PIP_Solution_Node ∗ as_solution () const

  Returns `this`.

- const Linear_Expression & parametric_values (Variable var) const

  Returns a parametric expression for the values of problem variable `var`.

- void ascii_dump (std::ostream &s) const

  Dumps to `s` *an ASCII representation of* `*this`.

- bool ascii_load (std::istream &s)

  Loads from `s` *an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets* `*this` *accordingly. Returns* `true` *if successful,* `false` *otherwise.*

- virtual memory_size_type total_memory_in_bytes () const

  Returns the total size in bytes of the memory occupied by `*this`.

- virtual memory_size_type external_memory_in_bytes () const

  Returns the size in bytes of the memory managed by `*this`.

Protected Member Functions

- PIP_Solution_Node (const PIP_Solution_Node &y)

  Copy constructor.

- PIP_Solution_Node (const PIP_Solution_Node &y, No_Constraints)

  Alternative copy constructor.

- virtual void set_owner (const PIP_Problem *owner)

  Sets the pointer to the PIP_Problem owning object.

- virtual bool check_ownership (const PIP_Problem *owner) const

  Returns `true` *if and only if all the nodes in the subtree rooted in* `*this` *is owned by* `*pip`.

- virtual void update_tableau (const PIP_Problem &pip, dimension_type external_space_dim, dimension_type first_pending_constraint, const Constraint_Sequence &input_cs, const Variables_-Set &parameters)

  Implements pure virtual method PIP_Tree_Node::update_tableau.

- void update_solution (const std::vector< bool > &pip_dim_is_param) const

  Update the solution values.

- void update_solution () const

  Helper method.

- virtual PIP_Tree_Node * solve (const PIP_Problem &pip, bool check_feasible_context, const Matrix &context, const Variables_Set &params, dimension_type space_dim, unsigned indent_level)

  Implements pure virtual method PIP_Tree_Node::solve.

- void generate_cut (dimension_type i, Variables_Set &parameters, Matrix &context, dimension_type &space_dimension, unsigned indent_level)

  Generate a Gomory cut using non-integer tableau row `i`.

- virtual void print_tree (std::ostream &s, unsigned indent, const std::vector< bool > &pip_dim_is_-param, dimension_type first_art_dim) const

  Prints on `s` *the tree rooted in* `*this`.

10.44.1 Detailed Description

A tree node representing part of the space of solutions.

10.44.2 Constructor & Destructor Documentation

10.44.2.1 Parma_Polyhedra_Library::PIP_Solution_Node::PIP_Solution_Node (const PIP_Solution_Node & *y*, No_Constraints) `[protected]`

Alternative copy constructor.

This constructor differs from the default copy constructor in that it will not copy the constraint system, nor the artificial parameters.

10.44.3 Member Function Documentation

10.44.3.1 const Linear_Expression& Parma_Polyhedra_Library::PIP_Solution_-Node::parametric_values (Variable *var*) const

Returns a parametric expression for the values of problem variable `var`.

The returned linear expression may involve problem parameters as well as artificial parameters.

Parameters

var The problem variable which is queried about.

Exceptions

std::invalid_argument Thrown if `var` is dimension-incompatible with the PIP_Problem owning this solution node, or if `var` is a problem parameter.

10.44.3.2 void Parma_Polyhedra_Library::PIP_Solution_Node::update_solution (const std::vector< bool > & *pip_dim_is_param*) const `[protected]`

Update the solution values.

Parameters

pip_dim_is_param A vector of Boolean flags telling which PIP problem dimensions are problem parameters. The size of the vector is equal to the PIP problem internal space dimension (i.e., no artificial parameters).

10.44.3.3 void Parma_Polyhedra_Library::PIP_Solution_Node::generate_cut (dimension_type *i*, Variables_Set & *parameters*, Matrix & *context*, dimension_type & *space_dimension*, unsigned *indent_level*) `[protected]`

Generate a Gomory cut using non-integer tableau row `i`.

Parameters

i Row index in simplex tableau from which the cut is generated

parameters A std::set of the current parameter dimensions (including artificials); to be updated if a new artificial parameter is to be created

context A set of linear inequalities on the parameters, in matrix form; to be updated if a new artificial parameter is to be created

space_dimension The current space dimension, including variables and all parameters; to be updated if an extra parameter is to be created

indent_level The indentation level (for debugging output only).

The documentation for this class was generated from the following file:

- ppl.hh

10.45 Parma_Polyhedra_Library::PIP_Tree_Node Class Reference

A node of the PIP solution tree.

`#include <ppl.hh>`

Inherited by Parma_Polyhedra_Library::PIP_Decision_Node, and Parma_Polyhedra_Library::PIP_-Solution_Node.

Classes

- class Artificial_Parameter

  Artificial parameters in PIP solution trees.

Public Types

- typedef std::vector< Artificial_Parameter > Artificial_Parameter_Sequence

  A type alias for a sequence of Artificial_Parameter's.

Public Member Functions

- virtual PIP_Tree_Node ∗ clone () const =0

  Returns a pointer to a dynamically-allocated copy of `*this`.

- virtual ~PIP_Tree_Node ()

  Destructor.

- virtual bool OK () const

  Returns `true` *if and only if* `*this` *is well formed.*

- virtual const PIP_Solution_Node ∗ as_solution () const

Returns `this` if `*this` is a solution node, 0 otherwise.

- virtual const PIP_Decision_Node * as_decision () const

  Returns `this` if `*this` is a decision node, 0 otherwise.

- const Constraint_System & constraints () const

  Returns the system of parameter constraints controlling `*this`.

- Artificial_Parameter_Sequence::const_iterator art_parameter_begin () const

  Returns a const_iterator to the beginning of local artificial parameters.

- Artificial_Parameter_Sequence::const_iterator art_parameter_end () const

  Returns a const_iterator to the end of local artificial parameters.

- dimension_type art_parameter_count () const

  Returns the number of local artificial parameters.

- void print (std::ostream &s, unsigned indent=0) const

  Prints on `s` the tree rooted in `*this`.

- void ascii_dump (std::ostream &s) const

  Dumps to `s` an ASCII representation of `*this`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- virtual memory_size_type total_memory_in_bytes () const =0

  Returns the total size in bytes of the memory occupied by `*this`.

- virtual memory_size_type external_memory_in_bytes () const =0

  Returns the size in bytes of the memory managed by `*this`.

Protected Types

- typedef std::vector< Constraint > Constraint_Sequence

  A type alias for a sequence of constraints.

Protected Member Functions

- PIP_Tree_Node (const PIP_Problem *owner)

  Constructor: builds a node owned by `*owner`.

- PIP_Tree_Node (const PIP_Tree_Node &y)

  Copy constructor.

- const PIP_Problem * get_owner () const

Returns a pointer to the PIP_Problem owning object.

- virtual void set_owner (const PIP_Problem ∗owner)=0

    Sets the pointer to the PIP_Problem owning object.

- virtual bool check_ownership (const PIP_Problem ∗owner) const =0

    Returns `true` if and only if all the nodes in the subtree rooted in `*this` is owned by `*pip`.

- const PIP_Decision_Node ∗ parent () const

    Returns a pointer to this node's parent.

- void set_parent (const PIP_Decision_Node ∗p)

    Set this node's parent to `*p`.

- virtual void update_tableau (const PIP_Problem &pip, dimension_type external_space_dim, dimension_type first_pending_constraint, const Constraint_Sequence &input_cs, const Variables_Set &parameters)=0

    Populates the parametric simplex tableau using external data.

- virtual PIP_Tree_Node ∗ solve (const PIP_Problem &pip, bool check_feasible_context, const Matrix &context, const Variables_Set &params, dimension_type space_dim, unsigned indent_level)=0

    Executes a parametric simplex on the tableau, under specified context.

- void add_constraint (const Row &x, const Variables_Set &parameters)

    Inserts a new parametric constraint in internal Row format.

- void parent_merge ()

    Merges parent's artificial parameters into `*this`.

- virtual void print_tree (std::ostream &s, unsigned indent, const std::vector< bool > &pip_dim_is_param, dimension_type first_art_dim) const

    Prints on `s` the tree rooted in `*this`.

Static Protected Member Functions

- static void indent_and_print (std::ostream &s, unsigned indent, const char ∗str)

    A helper function used when printing PIP trees.

- static bool compatibility_check (Matrix &s)

    Checks whether a context matrix is satisfiable.

- static bool compatibility_check (const Matrix &context, const Row &row)

    Helper method: checks for satisfiability of the restricted context obtained by adding `row` to `context`.

Protected Attributes

- const PIP_Problem * owner_

  A pointer to the PIP_Problem object owning this node.

- const PIP_Decision_Node * parent_

  A pointer to the parent of `*this`, null if `*this` is the root.

- Constraint_System constraints_

  The local system of parameter constraints.

- Artificial_Parameter_Sequence artificial_parameters

  The local sequence of expressions for local artificial parameters.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &os, const PIP_Tree_Node &x)

  Output operator: prints the solution tree rooted in x.

10.45.1 Detailed Description

A node of the PIP solution tree. This is the base class for the nodes of the binary trees representing the solutions of PIP problems. From this one, two classes are derived:

- PIP_Decision_Node, for the internal nodes of the tree;
- PIP_Solution_Node, for the leaves of the tree.

10.45.2 Member Function Documentation

10.45.2.1 const Constraint_System & Parma_Polyhedra_Library::PIP_Tree_Node::constraints () const [inline]

Returns the system of parameter constraints controlling `*this`.

The indices in the constraints are the same as the original variables and parameters. Coefficients in indices corresponding to variables always are zero.

10.45.2.2 void Parma_Polyhedra_Library::PIP_Tree_Node::print (std::ostream & *s*, unsigned *indent* = 0) const

Prints on `s` the tree rooted in `*this`.

Parameters

s The output stream.

indent The amount of indentation.

10.45.2.3 virtual void Parma_Polyhedra_Library::PIP_Tree_Node::update_tableau (const PIP_Problem & *pip*, dimension_type *external_space_dim*, dimension_type *first_pending_constraint*, const Constraint_Sequence & *input_cs*, const Variables_Set & *parameters*) `[protected, pure virtual]`

Populates the parametric simplex tableau using external data.

Parameters

pip The PIP_Problem object containing this node.

external_space_dim The number of all problem variables and problem parameters (excluding artificial parameters).

first_pending_constraint The first element in `input_cs` to be added to the tableau, which already contains the previous elements.

input_cs All the constraints of the PIP problem.

parameters The set of indices of the problem parameters.

Implemented in Parma_Polyhedra_Library::PIP_Solution_Node, and Parma_Polyhedra_Library::PIP_-Decision_Node.

10.45.2.4 virtual PIP_Tree_Node* Parma_Polyhedra_Library::PIP_Tree_Node::solve (const PIP_Problem & *pip*, bool *check_feasible_context*, const Matrix & *context*, const Variables_Set & *params*, dimension_type *space_dim*, unsigned *indent_level*) `[protected, pure virtual]`

Executes a parametric simplex on the tableau, under specified context.

Returns

The root of the PIP tree solution, or 0 if unfeasible.

Parameters

pip The PIP_Problem object containing this node.

check_feasible_context Whether the resolution process should (re-)check feasibility of context (since the initial context may have been modified).

context The context, being a set of constraints on the parameters.

params The local parameter set, including parent's artificial parameters.

space_dim The space dimension of parent, including artificial parameters.

indent_level The indentation level (for debugging output only).

Implemented in Parma_Polyhedra_Library::PIP_Solution_Node, and Parma_Polyhedra_Library::PIP_-Decision_Node.

10.45.2.5 virtual void Parma_Polyhedra_Library::PIP_Tree_Node::print_tree (std::ostream & *s*, unsigned *indent*, const std::vector< bool > & *pip_dim_is_param*, dimension_type *first_art_dim*) const [protected, virtual]

Prints on `s` the tree rooted in `*this`.

Parameters

s The output stream.

indent The amount of indentation.

pip_dim_is_param A vector of Boolean flags telling which PIP problem dimensions are problem parameters. The size of the vector is equal to the PIP problem internal space dimension (i.e., no artificial parameters).

first_art_dim The first space dimension corresponding to an artificial parameter that was created in this node (if any).

Reimplemented in Parma_Polyhedra_Library::PIP_Solution_Node, and Parma_Polyhedra_Library::PIP_-Decision_Node.

10.45.2.6 static bool Parma_Polyhedra_Library::PIP_Tree_Node::compatibility_check (Matrix & *s*) [static, protected]

Checks whether a context matrix is satisfiable.

The satisfiability check is implemented by the revised dual simplex algorithm on the context matrix. The algorithm ensures the feasible solution is integer by applying a cut generation method when intermediate non-integer solutions are found.

10.45.3 Friends And Related Function Documentation

10.45.3.1 std::ostream & operator<< (std::ostream & *os*, const PIP_Tree_Node & *x*) [related]

Output operator: prints the solution tree rooted in `x`.

The documentation for this class was generated from the following file:

- ppl.hh

10.46 Parma_Polyhedra_Library::Pointset_Powerset< PSET > Class Template Reference

The powerset construction instantiated on PPL pointset domains.

`#include <ppl.hh>`

Inherits Powerset< Parma_Polyhedra_Library::Determinate< PSET > >.

Public Member Functions

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

Constructors

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

  Builds a universe (top) or empty (bottom) Pointset_Powerset.

- Pointset_Powerset (const Pointset_Powerset &y, Complexity_Class complexity=ANY_-COMPLEXITY)

  Ordinary copy constructor.

- template<typename QH >
  Pointset_Powerset (const Pointset_Powerset< QH > &y, Complexity_Class complexity=ANY_-COMPLEXITY)

  Conversion constructor: the type `QH` of the disjuncts in the source powerset is different from `PSET`.

- template<typename QH1 , typename QH2 , typename R >
  Pointset_Powerset (const Partially_Reduced_Product< QH1, QH2, R > &prp, Complexity_Class complexity=ANY_COMPLEXITY)

  Creates a Pointset_Powerset from a product This will be created as a single disjunct of type PSET that approximates the product.

- Pointset_Powerset (const Constraint_System &cs)

  Creates a Pointset_Powerset with a single disjunct approximating the system of constraints `cs`.

- Pointset_Powerset (const Congruence_System &cgs)

  Creates a Pointset_Powerset with a single disjunct approximating the system of congruences `cgs`.

- Pointset_Powerset (const C_Polyhedron &ph, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a pointset_powerset out of a closed polyhedron.

- Pointset_Powerset (const NNC_Polyhedron &ph, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a pointset_powerset out of an nnc polyhedron.

- Pointset_Powerset (const Grid &gr, Complexity_Class complexity=ANY_COMPLEXITY)

  Builds a pointset_powerset out of a grid.

- template<typename T >
  Pointset_Powerset (const Octagonal_Shape< T > &os, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a pointset_powerset out of an octagonal shape.
- template<typename T >
  Pointset_Powerset (const BD_Shape< T > &bds, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a pointset_powerset out of a bd shape.
- template<typename Interval >
  Pointset_Powerset (const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

  Builds a pointset_powerset out of a box.

Member Functions that Do Not Modify the Pointset_Powerset

- dimension_type space_dimension () const

  Returns the dimension of the vector space enclosing `*this`.
- dimension_type affine_dimension () const

  Returns the dimension of the vector space enclosing `*this`.
- bool is_empty () const

  Returns `true` if and only if `*this` is an empty powerset.
- bool is_universe () const

  Returns `true` if and only if `*this` is the top element of the powerser lattice.
- bool is_topologically_closed () const

  Returns `true` if and only if all the disjuncts in `*this` are topologically closed.
- bool is_bounded () const

  Returns `true` if and only if all elements in `*this` are bounded.
- bool is_disjoint_from (const Pointset_Powerset &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 constrains (Variable var) const

  Returns `true` if and only if `var` is constrained in `*this`.
- 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 &g) 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 &g) 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 geometrically_covers (const Pointset_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 Pointset_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.*

- bool contains (const Pointset_Powerset &y) const

  Returns `true` *if and only if each disjunct of* `y` *is contained in a disjunct of* `*this`*.*

- bool strictly_contains (const Pointset_Powerset &y) const

  Returns `true` *if and only if each disjunct of* `y` *is strictly contained in a disjunct of* `*this`*.*

- bool contains_integer_point () const

  Returns `true` *if and only if* `*this` *contains at least one integer point.*

- Poly_Con_Relation relation_with (const Constraint &c) const

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

- Poly_Gen_Relation relation_with (const Generator &g) const

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

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between the powerset `*this` *and the congruence* `c`*.*

- 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`*.*

- int32_t hash_code () const

  Returns a 32-bit hash code for `*this`*.*

- bool OK () const

  Checks if all the invariants are satisfied.

Space Dimension Preserving Member Functions that May Modify the Pointset_Powerset

- void add_disjunct (const PSET &ph)
  Adds to `*this` the disjunct `ph`.
- void add_constraint (const Constraint &c)
  Intersects `*this` with constraint `c`.
- void refine_with_constraint (const Constraint &c)
  Use the constraint `c` to refine `*this`.
- void add_constraints (const Constraint_System &cs)
  Intersects `*this` with the constraints in `cs`.
- void refine_with_constraints (const Constraint_System &cs)
  Use the constraints in `cs` to refine `*this`.
- void add_congruence (const Congruence &c)
  Intersects `*this` with congruence `c`.
- void refine_with_congruence (const Congruence &cg)
  Use the congruence `cg` to refine `*this`.
- void add_congruences (const Congruence_System &cgs)
  Intersects `*this` with the congruences in `cgs`.
- void refine_with_congruences (const Congruence_System &cgs)
  Use the congruences in `cgs` to refine `*this`.
- void unconstrain (Variable var)
  Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.
- void unconstrain (const Variables_Set &vars)
  Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.
- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)
  Possibly tightens `*this` by dropping some points with non-integer coordinates.
- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)
  Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.
- void topological_closure_assign ()
  Assigns to `*this` its topological closure.
- void intersection_assign (const Pointset_Powerset &y)
  Assigns to `*this` the intersection of `*this` and `y`.
- void difference_assign (const Pointset_Powerset &y)
  Assigns to `*this` an (a smallest) over-approximation as a powerset of the disjunct domain of the set-theoretical difference of `*this` and `y`.

- bool simplify_using_context_assign (const Pointset_Powerset &y)

  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`. *If* `false` *is returned, then the intersection is empty.*

- 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, 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, 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, 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, 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 Pointset_Powerset &y)

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

- void wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

  Wraps the specified dimensions of the vector space.

- void pairwise_reduce ()

  Assign to `*this` *the result of (recursively) merging together the pairs of disjuncts whose upper-bound is the same as their set-theoretical union.*

- template<typename Widening >
  void BGP99_extrapolation_assign (const Pointset_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 Pointset_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

- Pointset_Powerset & operator= (const Pointset_Powerset &y)

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

- template<typename QH >
  Pointset_Powerset & operator= (const Pointset_Powerset< QH > &y)

  Conversion assignment: the type `QH` of the disjuncts in the source powerset is different from `PSET` (`*this` and `y` can be dimension-incompatible).

- void swap (Pointset_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 disjunct 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 disjuncts in `*this` in the new space.

- void concatenate_assign (const Pointset_Powerset &y)

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

- void remove_space_dimensions (const Variables_Set &vars)

  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.

- 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 &vars, Variable dest)

  Folds the space dimensions in `vars` into `dest`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Pointset_Powerset<PSET> can handle.

Related Functions

(Note that these are not member functions.)

- template< typename PSET >
  Widening_Function< PSET > widen_fun_ref (void(PSET::∗wm)(const PSET &, unsigned ∗))

  Wraps a widening method into a function object.

- template< typename PSET , typename CSYS >
  Limited_Widening_Function< PSET, CSYS > widen_fun_ref (void(PSET::∗lwm)(const PSET &, const CSYS &, unsigned ∗), const CSYS &cs)

  Wraps a limited widening method into a function object.

- template< typename PSET >
  std::pair< PSET, Pointset_Powerset< NNC_Polyhedron > > linear_partition (const PSET &p, const PSET &q)

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

- bool check_containment (const NNC_Polyhedron &ph, const Pointset_Powerset< NNC_Polyhedron > &ps)

  Returns `true` *if and only if the union of the NNC polyhedra in* `ps` *contains the NNC polyhedron* `ph`.

- std::pair< Grid, Pointset_Powerset< Grid > > approximate_partition (const Grid &p, const Grid &q, bool &finite_partition)

  Partitions the grid `q` *with respect to grid* `p` *if and only if such a partition is finite.*

- bool check_containment (const Grid &ph, const Pointset_Powerset< Grid > &ps)

  Returns `true` *if and only if the union of the grids* `ps` *contains the grid* `g`.

- template< typename PSET >
  bool check_containment (const PSET &ph, const Pointset_Powerset< PSET > &ps)

  Returns `true` *if and only if the union of the objects in* `ps` *contains* `ph`.

- template< typename PSET >
  void swap (Parma_Polyhedra_Library::Pointset_Powerset< PSET > &x, Parma_Polyhedra_Library::Pointset_Powerset< PSET > &y)

  Specializes `std::swap`.

- template<>
  bool check_containment (const C_Polyhedron &ph, const Pointset_Powerset< C_Polyhedron > &ps)

10.46.1 Detailed Description

template<typename PSET> class Parma_Polyhedra_Library::Pointset_Powerset< PSET >

The powerset construction instantiated on PPL pointset domains.

Warning

At present, the supported instantiations for the disjunct domain template `PSET` are the simple pointset domains: `C_Polyhedron`, `NNC_Polyhedron`, `Grid`, `Octagonal_Shape<T>`, `BD_Shape<T>`, `Box<T>`.

10.46.2 Constructor & Destructor Documentation

10.46.2.1 template<typename PSET > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (dimension_type *num_dimensions* = 0, Degenerate_Element *kind* = UNIVERSE) [inline, explicit]

Builds a universe (top) or empty (bottom) Pointset_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.

10.46.2.2 template<typename PSET > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Pointset_Powerset< PSET > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Ordinary copy constructor.

The complexity argument is ignored.

10.46.2.3 template<typename PSET > template<typename QH > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Pointset_Powerset< QH > & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Conversion constructor: the type `QH` of the disjuncts in the source powerset is different from `PSET`.

Parameters

y The powerset to be used to build the new powerset.

complexity The maximal complexity of any algorithms used.

10.46.2.4 template<typename PSET > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const C_Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of a closed polyhedron.

Builds a powerset that is either empty (if the polyhedron is found to be empty) or contains a single disjunct approximating the polyhedron; this must only use algorithms that do not exceed the specified complexity. The powerset inherits the space dimension of the polyhedron.

Parameters

ph The closed polyhedron to be used to build the powerset.

complexity The maximal complexity of any algorithms used.

Exceptions

std::length_error Thrown if the space dimension of `ph` exceeds the maximum allowed space dimension.

10.46.2.5 template<typename PSET > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const NNC_Polyhedron & *ph*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of an nnc polyhedron.

Builds a powerset that is either empty (if the polyhedron is found to be empty) or contains a single disjunct approximating the polyhedron; this must only use algorithms that do not exceed the specified complexity. The powerset inherits the space dimension of the polyhedron.

Parameters

ph The closed polyhedron to be used to build the powerset.

complexity The maximal complexity of any algorithms used.

Exceptions

std::length_error Thrown if the space dimension of `ph` exceeds the maximum allowed space dimension.

10.46.2.6 template<typename PSET > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Grid & *gr*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of a grid.

If the grid is nonempty, builds a powerset containing a single disjunct approximating the grid. Builds the empty powerset otherwise. The powerset inherits the space dimension of the grid.

Parameters

gr The grid to be used to build the powerset.

complexity This argument is ignored.

Exceptions

std::length_error Thrown if the space dimension of `gr` exceeds the maximum allowed space dimension.

10.46.2.7 template< typename PSET > template< typename T > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Octagonal_Shape< T > & *os*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of an octagonal shape.

If the octagonal shape is nonempty, builds a powerset containing a single disjunct approximating the octagonal shape. Builds the empty powerset otherwise. The powerset inherits the space dimension of the octagonal shape.

Parameters

os The octagonal shape to be used to build the powerset.

complexity This argument is ignored.

Exceptions

std::length_error Thrown if the space dimension of `os` exceeds the maximum allowed space dimension.

10.46.2.8 template< typename PSET > template< typename T > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const BD_Shape< T > & *bds*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of a bd shape.

If the bd shape is nonempty, builds a powerset containing a single disjunct approximating the bd shape. Builds the empty powerset otherwise. The powerset inherits the space dimension of the bd shape.

Parameters

bds The bd shape to be used to build the powerset.

complexity This argument is ignored.

Exceptions

std::length_error Thrown if the space dimension of `bdss` exceeds the maximum allowed space dimension.

10.46.2.9 template<typename PSET > template<typename Interval > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, explicit]

Builds a pointset_powerset out of a box.

If the box is nonempty, builds a powerset containing a single disjunct approximating the box. Builds the empty powerset otherwise. The powerset inherits the space dimension of the box.

Parameters

box The box to be used to build the powerset.

complexity This argument is ignored.

Exceptions

std::length_error Thrown if the space dimension of `box` exceeds the maximum allowed space dimension.

10.46.3 Member Function Documentation

10.46.3.1 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::is_disjoint_from (const Pointset_Powerset< PSET > & *y*) const [inline]

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.

10.46.3.2 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::constrains (Variable *var*) const [inline]

Returns `true` if and only if `var` is constrained in `*this`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

Note

A variable is constrained if there exists a non-redundant disjunct that is constraining the variable: this definition relies on the powerset lattice structure and may be somewhat different from the geometric intuition. For instance, variable x is constrained in the powerset

$$ps = \big\{\{x \geq 0\}, \{x \leq 0\}\big\},$$

even though ps is geometrically equal to the whole vector space.

10.46.3.3 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

10.46.3.4 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

10.46.3.5 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

10.46.3.6 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.46.3.7 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

10.46.3.8 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.46.3.9 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::geometrically_covers (const Pointset_Powerset< PSET > & *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 dimension-incompatible.

Warning

This may be *really* expensive!

10.46.3.10 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::geometrically_equals (const Pointset_Powerset< PSET > & *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 dimension-incompatible.

Warning

This may be *really* expensive!

10.46.3.11 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::contains (const Pointset_Powerset< PSET > & *y*) const [inline]

Returns `true` if and only if each disjunct of `y` is contained in a disjunct of `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.46.3.12 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::strictly_contains (const Pointset_Powerset< PSET > & *y*) const [inline]

Returns `true` if and only if each disjunct of `y` is strictly contained in a disjunct of `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.46.3.13 template<typename PSET > Poly_Con_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Constraint & *c*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are dimension-incompatible.

10.46.3.14 template<typename PSET > Poly_Gen_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Generator & *g*) const [inline]

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

Exceptions

std::invalid_argument Thrown if `*this` and generator `g` are dimension-incompatible.

10.46.3.15 template<typename PSET > Poly_Con_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Congruence & *cg*) const [inline]

Returns the relations holding between the powerset `*this` and the congruence `c`.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `c` are dimension-incompatible.

10.46.3.16 template<typename PSET > int32_t Parma_Polyhedra_Library::Pointset_Powerset< PSET >::hash_code () const [inline]

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.46.3.17 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_disjunct (const PSET & *ph*) [inline]

Adds to `*this` the disjunct `ph`.

Exceptions

std::invalid_argument Thrown if `*this` and `ph` are dimension-incompatible.

10.46.3.18 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_constraint (const Constraint & *c*) [inline]

Intersects `*this` with constraint `c`.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are topology-incompatible or dimension-incompatible.

10.46.3.19 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_constraint (const Constraint & *c*) [inline]

Use the constraint `c` to refine `*this`.

Parameters

c The constraint to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `c` are dimension-incompatible.

10.46.3.20 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_constraints (const Constraint_System & *cs*) [inline]

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.

10.46.3.21 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_constraints (const Constraint_System & *cs*) `[inline]`

Use the constraints in `cs` to refine `*this`.

Parameters

cs The constraints to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.46.3.22 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_congruence (const Congruence & *c*) `[inline]`

Intersects `*this` with congruence `c`.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `c` are topology-incompatible or dimension-incompatible.

10.46.3.23 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_congruence (const Congruence & *cg*) `[inline]`

Use the congruence `cg` to refine `*this`.

Parameters

cg The congruence to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cg` are dimension-incompatible.

10.46.3.24 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_congruences (const Congruence_System & *cgs*) [inline]

Intersects `*this` with the congruences in `cgs`.

Parameters

cgs The congruences to intersect with.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are topology-incompatible or dimension-incompatible.

10.46.3.25 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_congruences (const Congruence_System & *cgs*) [inline]

Use the congruences in `cgs` to refine `*this`.

Parameters

cgs The congruences to be used for refinement.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.46.3.26 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::unconstrain (Variable *var*) [inline]

Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.46.3.27 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::unconstrain (const Variables_Set & *vars*) [inline]

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.46.3.28 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.46.3.29 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.46.3.30 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::intersection_assign (const Pointset_Powerset< PSET > & *y*) [inline]

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

The result is obtained by intersecting each disjunct in `*this` with each disjunct in `y` and collecting all these intersections.

10.46.3.31 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::difference_assign (const Pointset_Powerset< PSET > & *y*) `[inline]`

Assigns to `*this` an (a smallest) over-approximation as a powerset of the disjunct domain of the set-theoretical difference of `*this` and `y`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.46.3.32 template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::simplify_using_context_assign (const Pointset_Powerset< PSET > & *y*) `[inline]`

Assigns to `*this` a meet-preserving simplification of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.46.3.33 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.46.3.34 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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 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`.

10.46.3.35 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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.

10.46.3.36 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::generalized_affine_preimage (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one()) [inline]

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.

10.46.3.37 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::generalized_affine_image (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) `[inline]`

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.

10.46.3.38 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::generalized_affine_preimage (const Linear_Expression & *lhs*, Relation_Symbol *relsym*, const Linear_Expression & *rhs*) `[inline]`

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.

10.46.3.39 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::bounded_affine_image (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.46.3.40 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::bounded_affine_preimage (Variable *var*, const Linear_Expression & *lb_expr*, const Linear_Expression & *ub_expr*, Coefficient_traits::const_reference *denominator* = `Coefficient_one()`) `[inline]`

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`.

10.46.3.41 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::time_elapse_assign (const Pointset_Powerset< PSET > & *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 disjunct in `*this` with each disjunct in `y`.

10.46.3.42 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System ∗ *pcs* = 0, unsigned *complexity_threshold* = 16, bool *wrap_individually* = true) [inline]

Wraps the specified dimensions of the vector space.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be wrapped.

w The width of the bounded integer type corresponding to all the dimensions to be wrapped.

r The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

o The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.

pcs Possibly null pointer to a constraint system whose variables are contained in `vars`. If `*pcs` depends on variables not in `vars`, the behavior is undefined. When non-null, the pointed-to constraint system is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation with the constraints in `*pcs`.

complexity_threshold A precision parameter of the wrapping operator: higher values result in possibly improved precision.

wrap_individually `true` if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

std::invalid_argument Thrown if `*pcs` is dimension-incompatible with `vars`, or if `*this` is dimension-incompatible `vars` or with `*pcs`.

10.46.3.43 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::pairwise_reduce () [inline]

Assign to `*this` the result of (recursively) merging together the pairs of disjuncts whose upper-bound is the same as their set-theoretical union.

On exit, for all the pairs $\mathcal{P}$, $\mathcal{Q}$ of different disjuncts in `*this`, we have $\mathcal{P} \uplus \mathcal{Q} \neq \mathcal{P} \cup \mathcal{Q}$.

10.46.3.44 template< typename PSET > template< typename Widening > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::BGP99_extrapolation_assign (const Pointset_Powerset< PSET > & *y*, Widening *wf*, unsigned *max_disjuncts*) [inline]

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 powerset that *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 dimension-incompatible.

For a description of the extrapolation operator, see [BGP99] and [BHZ03b].

10.46.3.45 template< typename PSET > template< typename Cert , typename Widening > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::BHZ03_widening_assign (const Pointset_Powerset< PSET > & *y*, Widening *wf*) `[inline]`

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 computed in the previous iteration step. It *must* definitely entail `*this`;

wf The widening function to be used on disjuncts. 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 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.

10.46.3.46 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::concatenate_assign (const Pointset_Powerset< PSET > & *y*) `[inline]`

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

The result is obtained by computing the pairwise concatenation of each disjunct in `*this` with each disjunct in `y`.

10.46.3.47 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::remove_space_dimensions (const Variables_Set & *vars*) [inline]

Removes all the specified space dimensions.

Parameters

vars 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 `vars`.

10.46.3.48 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::remove_higher_space_dimensions (dimension_type *new_dimension*) [inline]

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`.

10.46.3.49 template< typename PSET > template< typename Partial_Function > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::map_space_dimensions (const Partial_Function & *pfunc*) [inline]

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

See also Polyhedron::map_space_dimensions.

10.46.3.50 template< typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::expand_space_dimension (Variable *var*, dimension_type *m*) [inline]

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 $n, n + 1, \ldots, n + m - 1$.

10.46.3.51 template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*) [inline]

Folds the space dimensions in vars into dest.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 dest or with one of the Variable objects contained in vars. Also thrown if dest is contained in vars.

If *this has space dimension n, with $n > 0$, dest has space dimension $k \leq n$, vars is a set of variables whose maximum space dimension is also less than or equal to n, and dest is not a member of vars, then the space dimensions corresponding to variables in vars are folded into the k-th space dimension.

10.46.4 Friends And Related Function Documentation

10.46.4.1 template<typename PSET > Widening_Function< PSET > widen_fun_ref (void(PSET::*)(const PSET &, unsigned *) *wm*) [related]

Wraps a widening method into a function object.

Parameters

wm The widening method.

10.46.4.2 template<typename PSET , typename CSYS > Limited_Widening_Function< PSET, CSYS > widen_fun_ref (void(PSET::*)(const PSET &, const CSYS &, unsigned *) *lwm*, const CSYS & *cs*) [related]

Wraps a limited widening method into a function object.

Parameters

lwm The limited widening method.

cs The constraint system limiting the widening.

10.46.4.3 template< typename PSET > std::pair< PSET, Pointset_Powerset< NNC_Polyhedron > > linear_partition (const PSET & *p*, const PSET & *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<PSET, Pointset_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 set-theoretical union of r.first with all the elements of r.second gives q (i.e., r is the representation of a partition of q).

10.46.4.4 template< typename PSET > std::pair< Grid, Pointset_Powerset< Grid > > approximate_partition (const Grid & *p*, const Grid & *q*, bool & *finite_partition*) [related]

Partitions the grid q with respect to grid p if and only if such a partition is finite.

Let p and q be two grids. The function returns an object r of type std::pair<PSET, Pointset_Powerset<Grid> > such that

- r.first is the intersection of p and q;
- If there is a finite partition of q wrt p the Boolean finite_partition is set to true and r.second has the property that all its elements are pairwise disjoint and disjoint from p and the set-theoretical union of r.first with all the elements of r.second gives q (i.e., r is the representation of a partition of q).
- Otherwise the Boolean finite_partition is set to false and the singleton set that contains q is stored in r.secondr.

10.46.4.5 template< typename PSET > bool check_containment (const PSET & *ph*, const Pointset_Powerset< PSET > & *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 PSET can be converted without precision loss into an NNC_Polyhedron; otherwise, an incorrect result might be obtained.

10.46.4.6 template< typename PSET > void swap (Parma_Polyhedra_Library::Pointset_Powerset< PSET > & *x*, Parma_Polyhedra_Library::Pointset_Powerset< PSET > & *y*) [related]

Specializes std::swap.

10.46.4.7 bool check_containment (const C_Polyhedron & *ph*, const Pointset_Powerset< C_Polyhedron > & *ps*) [related]

The documentation for this class was generated from the following file:

- ppl.hh

10.47 Parma_Polyhedra_Library::Poly_Con_Relation Class Reference

The relation between a polyhedron and a constraint.

`#include <ppl.hh>`

Public Member Functions

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`*.*

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`*.*

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`*.*

- 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*.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Poly_Con_Relation &r)

  Output operator.

10.47.1 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.

10.47.2 Friends And Related Function Documentation

10.47.2.1 bool operator== (const Poly_Con_Relation & *x*, const Poly_Con_Relation & *y*) `[friend]`

True if and only if x and y are logically equivalent.

10.47.2.2 bool operator!= (const Poly_Con_Relation & *x*, const Poly_Con_Relation & *y*) `[friend]`

True if and only if x and y are not logically equivalent.

10.47.2.3 Poly_Con_Relation operator&& (const Poly_Con_Relation & *x*, const Poly_Con_Relation & *y*) `[friend]`

Yields the logical conjunction of x and y.

10.47.2.4 Poly_Con_Relation operator- (const Poly_Con_Relation & *x*, const Poly_Con_Relation & *y*) [friend]

Yields the assertion with all the conjuncts of x that are not in y.

10.47.2.5 std::ostream & operator<< (std::ostream & *s*, const Poly_Con_Relation & *r*) [related]

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.48 Parma_Polyhedra_Library::Poly_Gen_Relation Class Reference

The relation between a polyhedron and a generator.

`#include <ppl.hh>`

Public Member Functions

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

- 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*.*

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Poly_Gen_Relation &r)
  Output operator.

10.48.1 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.

10.48.2 Friends And Related Function Documentation

10.48.2.1 bool operator== (const Poly_Gen_Relation & *x*, const Poly_Gen_Relation & *y*) [friend]

True if and only if x and y are logically equivalent.

10.48.2.2 bool operator!= (const Poly_Gen_Relation & *x*, const Poly_Gen_Relation & *y*) [friend]

True if and only if x and y are not logically equivalent.

10.48.2.3 Poly_Gen_Relation operator&& (const Poly_Gen_Relation & *x*, const Poly_Gen_Relation & *y*) [friend]

Yields the logical conjunction of x and y.

10.48.2.4 Poly_Gen_Relation operator- (const Poly_Gen_Relation & *x*, const Poly_Gen_Relation & *y*) [friend]

Yields the assertion with all the conjuncts of x that are not in y.

10.48.2.5 std::ostream & operator<< (std::ostream & *s*, const Poly_Gen_Relation & *r*) [related]

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

10.49 Parma_Polyhedra_Library::Polyhedron Class Reference

The base class for convex polyhedra.

`#include <ppl.hh>`

Inherited by Parma_Polyhedra_Library::C_Polyhedron, and Parma_Polyhedra_Library::NNC_-Polyhedron.

Public Types

- typedef Coefficient coefficient_type

  The numeric type of coefficients.

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.

- Congruence_System congruences () const

  Returns a system of (equality) congruences satisfied by `*this`.

- Congruence_System minimized_congruences () const

  Returns a system of (equality) congruences satisfied by `*this`*, with no redundant congruences and having the same affine dimension as* `*this`.

- Grid_Generator_System grid_generators () const

  Returns a universe system of grid generators.

- Grid_Generator_System minimized_grid_generators () const

  Returns a universe system of grid generators.

- 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`.

- Poly_Con_Relation relation_with (const Congruence &cg) const

  Returns the relations holding between the polyhedron `*this` *and the congruence* `c`.

- 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_discrete () const

  Returns `true` *if and only if* `*this` *is discrete.*

- bool is_bounded () const

  Returns `true` *if and only if* `*this` *is a bounded polyhedron.*

- bool contains_integer_point () const

  Returns `true` *if and only if* `*this` *contains at least one integer point.*

- bool constrains (Variable var) const

  Returns `true` *if and only if* `var` *is constrained in* `*this`.

- 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 &g) 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 &g) 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 frequency (const Linear_Expression &expr, Coefficient &freq_n, Coefficient &freq_d, Coefficient &val_n, Coefficient &val_d) const

  Returns `true` *if and only if there exist a unique value* `val` *such that* `*this` *saturates the equality* `expr = val`*.*

- 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`*.*

- 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).*

- void add_generator (const Generator &g)

  Adds a copy of generator `g` *to the system of generators of* `*this` *(without minimizing the result).*

- void add_congruence (const Congruence &cg)

  Adds a copy of congruence `cg` *to* `*this`*, if* `cg` *can be exactly represented by a polyhedron.*

- 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).*

- 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).*
- void add_congruences (const Congruence_System &cgs)
  Adds a copy of the congruences in `cgs` *to* `*this`*, if all the congruences can be exactly represented by a polyhedron.*
- void add_recycled_congruences (Congruence_System &cgs)
  Adds the congruences in `cgs` *to* `*this`*, if all the congruences can be exactly represented by a polyhedron.*
- void refine_with_constraint (const Constraint &c)
  Uses a copy of constraint `c` *to refine* `*this`*.*
- void refine_with_congruence (const Congruence &cg)
  Uses a copy of congruence `cg` *to refine* `*this`*.*
- void refine_with_constraints (const Constraint_System &cs)
  Uses a copy of the constraints in `cs` *to refine* `*this`*.*
- void refine_with_congruences (const Congruence_System &cgs)
  Uses a copy of the congruences in `cgs` *to refine* `*this`*.*
- void unconstrain (Variable var)
  Computes the cylindrification of `*this` *with respect to space dimension* `var`*, assigning the result to* `*this`*.*
- void unconstrain (const Variables_Set &vars)
  Computes the cylindrification of `*this` *with respect to the set of space dimensions* `vars`*, assigning the result to* `*this`*.*
- void intersection_assign (const Polyhedron &y)
  Assigns to `*this` *the intersection of* `*this` *and* `y`*.*
- void poly_hull_assign (const Polyhedron &y)
  Assigns to `*this` *the poly-hull of* `*this` *and* `y`*.*
- 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`*.*
- void difference_assign (const Polyhedron &y)
  Same as poly_difference_assign(y).
- bool simplify_using_context_assign (const Polyhedron &y)
  Assigns to `*this` *a meet-preserving simplification of* `*this` *with respect to* `y`*. If* `false` *is returned, then the intersection is empty.*
- 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, 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, 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, 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, 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 wrap_assign (const Variables_Set &vars, Bounded_Integer_Type_Width w, Bounded_-Integer_Type_Representation r, Bounded_Integer_Type_Overflow o, const Constraint_System *pcs=0, unsigned complexity_threshold=16, bool wrap_individually=true)

  Wraps the specified dimensions of the vector space.

- void drop_some_non_integer_points (Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates.*

- void drop_some_non_integer_points (const Variables_Set &vars, Complexity_Class complexity=ANY_COMPLEXITY)

  Possibly tightens `*this` *by dropping some points with non-integer coordinates for the space dimensions corresponding to* `vars`*.*

- 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)

  Assigns to `*this` the result of computing the limited extrapolation between `*this` and `y` using the BHRZ03-widening operator.

- void bounded_BHRZ03_extrapolation_assign (const Polyhedron &y, const Constraint_System &cs, unsigned *tp=0)

  Assigns to `*this` the result of computing the bounded extrapolation between `*this` and `y` using the BHRZ03-widening operator.

- 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 widening_assign (const Polyhedron &y, unsigned *tp=0)

  Same as H79_widening_assign(y, tp).

- void limited_H79_extrapolation_assign (const Polyhedron &y, const Constraint_System &cs, unsigned *tp=0)

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

- void bounded_H79_extrapolation_assign (const Polyhedron &y, const Constraint_System &cs, unsigned *tp=0)

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

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 &vars)

  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 &vars, Variable dest)

Folds the space dimensions in `vars` into `dest`.

Miscellaneous Member Functions

- ~Polyhedron ()

  Destructor.

- void swap (Polyhedron &y)

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

- void ascii_dump () const

  Writes to `std::cerr` an ASCII representation of `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` an ASCII representation of `*this`.

- void print () const

  Prints `*this` to `std::cerr` using `operator<<`.

- bool ascii_load (std::istream &s)

  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.

- 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`.

- int32_t hash_code () const

  Returns a 32-bit hash code for `*this`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension all kinds of Polyhedron can handle.

- static bool can_recycle_constraint_systems ()

  Returns `true` indicating that this domain has methods that can recycle constraints.

- static void initialize ()

  Initializes the class.

- static void finalize ()

  Finalizes the class.

- static bool can_recycle_congruence_systems ()

  Returns `false` indicating that this domain cannot recycle congruences.

Protected Member Functions

- Polyhedron (Topology topol, dimension_type num_dimensions, Degenerate_Element kind)

    Builds a polyhedron having the specified properties.

- Polyhedron (const Polyhedron &y, Complexity_Class complexity=ANY_COMPLEXITY)

    Ordinary copy constructor.

- Polyhedron (Topology topol, const Constraint_System &cs)

    Builds a polyhedron from a system of constraints.

- Polyhedron (Topology topol, Constraint_System &cs, Recycle_Input dummy)

    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, Recycle_Input dummy)

    Builds a polyhedron recycling a system of generators.

- template<typename Interval >
  Polyhedron (Topology topol, const Box< Interval > &box, Complexity_Class complexity=ANY_-COMPLEXITY)

    Builds a polyhedron from a box.

- Polyhedron & operator= (const Polyhedron &y)

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

- void drop_some_non_integer_points (const Variables_Set *pvars, Complexity_Class complexity)

    Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `*pvars`.

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`.

10.49.1 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.

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

$$\bigl\{\, (2, y)^{\mathrm{T}} \in \mathbb{R}^2 \bigm| y \in \mathbb{R} \,\bigr\}.$$

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 $\bigl\{ (2,0)^{\mathrm{T}} \bigr\} \subseteq \mathbb{R}^2$.

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 vars;
vars.insert(y);
vars.insert(z);
ph.remove_space_dimensions(vars);
```

The starting polyhedron is the singleton set $\{(3,1,0,2)^{\mathrm{T}}\} \subseteq \mathbb{R}^4$, while the resulting polyhedron is $\{(3,2)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$. Be careful when removing 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> vars1;
vars1.insert(y);
ph.remove_space_dimensions(vars1);
set<Variable> vars2;
vars2.insert(z);
ph.remove_space_dimensions(vars2);
```

In this case, the result is the polyhedron $\{(3,0)^{\mathrm{T}}\} \subseteq \mathbb{R}^2$: when removing the set of dimensions `vars2` 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.

10.49.2 Constructor & Destructor Documentation

10.49.2.1 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.

10.49.2.2 Parma_Polyhedra_Library::Polyhedron::Polyhedron (const Polyhedron & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) `[protected]`

Ordinary copy constructor.

The complexity argument is ignored.

10.49.2.3 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`.

10.49.2.4 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, Constraint_System & *cs*, Recycle_Input *dummy*) `[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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

std::invalid_argument Thrown if the topology of `cs` is incompatible with `topol`.

10.49.2.5 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.

10.49.2.6 Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, Generator_System & *gs*, Recycle_Input *dummy*) `[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 may be recycled to build the polyhedron.

dummy A dummy tag to syntactically differentiate this one from the other constructors.

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.

10.49.2.7 template<typename Interval > Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline, protected]

Builds a polyhedron from a box.

This will use an algorithm whose complexity is polynomial and build the smallest polyhedron with topology `topol` containing `box`.

Parameters

topol The topology of the polyhedron;

box The box representing the polyhedron to be built;

complexity This argument is ignored.

10.49.3 Member Function Documentation

10.49.3.1 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.

10.49.3.2 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.

10.49.3.3 Poly_Con_Relation Parma_Polyhedra_Library::Polyhedron::relation_with (const Congruence & *cg*) const

Returns the relations holding between the polyhedron `*this` and the congruence `c`.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `c` are dimension-incompatible.

10.49.3.4 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.

10.49.3.5 bool Parma_Polyhedra_Library::Polyhedron::constrains (Variable *var*) const

Returns `true` if and only if `var` is constrained in `*this`.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.49.3.6 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.

10.49.3.7 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.

10.49.3.8 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.

10.49.3.9 bool Parma_Polyhedra_Library::Polyhedron::maximize (const Linear_Expression & *expr*, Coefficient & *sup_n*, Coefficient & *sup_d*, bool & *maximum*, Generator & *g*) 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;
- ***g*** 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 `g` are left untouched.

10.49.3.10 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.

10.49.3.11 bool Parma_Polyhedra_Library::Polyhedron::minimize (const Linear_Expression & *expr*, Coefficient & *inf_n*, Coefficient & *inf_d*, bool & *minimum*, Generator & *g*) 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;

g 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 `g` are left untouched.

10.49.3.12 bool Parma_Polyhedra_Library::Polyhedron::frequency (const Linear_Expression & *expr*, Coefficient & *freq_n*, Coefficient & *freq_d*, Coefficient & *val_n*, Coefficient & *val_d*) const

Returns `true` if and only if there exist a unique value `val` such that `*this` saturates the equality `expr = val`.

Parameters

expr The linear expression for which the frequency is needed;

freq_n If `true` is returned, the value is set to 0; Present for interface compatibility with class Grid, where the frequency can have a non-zero value;

freq_d If `true` is returned, the value is set to 1;

val_n The numerator of `val`;

val_d The denominator of `val`;

Exceptions

std::invalid_argument Thrown if `expr` and `*this` are dimension-incompatible.

If `false` is returned, then `freq_n`, `freq_d`, `val_n` and `val_d` are left untouched.

10.49.3.13 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.

10.49.3.14 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.

10.49.3.15 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.

10.49.3.16 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).

Parameters

c The constraint that will be added to the system of constraints of `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and constraint `c` are topology-incompatible or dimension-incompatible.

10.49.3.17 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.

10.49.3.18 void Parma_Polyhedra_Library::Polyhedron::add_congruence (const Congruence & *cg*)

Adds a copy of congruence `cg` to `*this`, if `cg` can be exactly represented by a polyhedron.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible, of if `cg` is a proper congruence which is neither a tautology, nor a contradiction.

10.49.3.19 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.

10.49.3.20 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 to be added to `*this`. The constraints in `cs` may be recycled.

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.

10.49.3.21 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.

10.49.3.22 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 to be added to `*this`. The generators in `gs` may be recycled.

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.

10.49.3.23 void Parma_Polyhedra_Library::Polyhedron::add_congruences (const Congruence_System & *cgs*)

Adds a copy of the congruences in cgs to *this, if all the congruences can be exactly represented by a polyhedron.

Parameters

cgs The congruences to be added.

Exceptions

std::invalid_argument Thrown if *this and cgs are dimension-incompatible, of if there exists in cgs a proper congruence which is neither a tautology, nor a contradiction.

10.49.3.24 void Parma_Polyhedra_Library::Polyhedron::add_recycled_congruences (Congruence_System & *cgs*) [inline]

Adds the congruences in cgs to *this, if all the congruences can be exactly represented by a polyhedron.

Parameters

cgs The congruences to be added. Its elements may be recycled.

Exceptions

std::invalid_argument Thrown if *this and cgs are dimension-incompatible, of if there exists in cgs a proper congruence which is neither a tautology, nor a contradiction

Warning

The only assumption that can be made on cgs upon successful or exceptional return is that it can be safely destroyed.

10.49.3.25 void Parma_Polyhedra_Library::Polyhedron::refine_with_constraint (const Constraint & *c*)

Uses a copy of constraint c to refine *this.

Exceptions

std::invalid_argument Thrown if *this and constraint c are dimension-incompatible.

10.49.3.26 void Parma_Polyhedra_Library::Polyhedron::refine_with_congruence (const Congruence & *cg*)

Uses a copy of congruence `cg` to refine `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and congruence `cg` are dimension-incompatible.

10.49.3.27 void Parma_Polyhedra_Library::Polyhedron::refine_with_constraints (const Constraint_System & *cs*)

Uses a copy of the constraints in `cs` to refine `*this`.

Parameters

cs Contains the constraints used to refine the system of constraints of `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and `cs` are dimension-incompatible.

10.49.3.28 void Parma_Polyhedra_Library::Polyhedron::refine_with_congruences (const Congruence_System & *cgs*)

Uses a copy of the congruences in `cgs` to refine `*this`.

Parameters

cgs Contains the congruences used to refine the system of constraints of `*this`.

Exceptions

std::invalid_argument Thrown if `*this` and `cgs` are dimension-incompatible.

10.49.3.29 void Parma_Polyhedra_Library::Polyhedron::unconstrain (Variable *var*)

Computes the cylindrification of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

var The space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `var` is not a space dimension of `*this`.

10.49.3.30 void Parma_Polyhedra_Library::Polyhedron::unconstrain (const Variables_Set & *vars*)

Computes the cylindrification of `*this` with respect to the set of space dimensions `vars`, assigning the result to `*this`.

Parameters

vars The set of space dimension that will be unconstrained.

Exceptions

std::invalid_argument Thrown if `*this` is dimension-incompatible with one of the Variable objects contained in `vars`.

10.49.3.31 void Parma_Polyhedra_Library::Polyhedron::intersection_assign (const Polyhedron & *y*)

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.49.3.32 void Parma_Polyhedra_Library::Polyhedron::poly_hull_assign (const Polyhedron & *y*)

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.49.3.33 void Parma_Polyhedra_Library::Polyhedron::poly_difference_assign (const Polyhedron & *y*)

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

Exceptions

std::invalid_argument Thrown if `*this` and `y` are topology-incompatible or dimension-incompatible.

10.49.3.34 bool Parma_Polyhedra_Library::Polyhedron::simplify_using_context_assign (const Polyhedron & *y*)

Assigns to *this a meet-preserving simplification of *this with respect to y. If false is returned, then the intersection is empty.

Exceptions

std::invalid_argument Thrown if *this and y are topology-incompatible or dimension-incompatible.

10.49.3.35 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.

10.49.3.36 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.

10.49.3.37 void Parma_Polyhedra_Library::Polyhedron::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 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.

10.49.3.38 void Parma_Polyhedra_Library::Polyhedron::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 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.

10.49.3.39 void Parma_Polyhedra_Library::Polyhedron::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 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.

10.49.3.40 void Parma_Polyhedra_Library::Polyhedron::generalized_affine_preimage (const Linear_Expression & *lhs*, 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.

10.49.3.41 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`.

10.49.3.42 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`.

10.49.3.43 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.

10.49.3.44 void Parma_Polyhedra_Library::Polyhedron::wrap_assign (const Variables_Set & *vars*, Bounded_Integer_Type_Width *w*, Bounded_Integer_Type_Representation *r*, Bounded_Integer_Type_Overflow *o*, const Constraint_System * *pcs* = `0`, unsigned *complexity_threshold* = `16`, bool *wrap_individually* = `true`)

Wraps the specified dimensions of the vector space.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be wrapped.

w The width of the bounded integer type corresponding to all the dimensions to be wrapped.

r The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

o The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.

pcs Possibly null pointer to a constraint system whose variables are contained in `vars`. If `*pcs` depends on variables not in `vars`, the behavior is undefined. When non-null, the pointed-to constraint system is assumed to represent the conditional or looping construct guard with respect to which wrapping is performed. Since wrapping requires the computation of upper bounds and due to non-distributivity of constraint refinement over upper bounds, passing a constraint system in this way can be more precise than refining the result of the wrapping operation with the constraints in `*pcs`.

complexity_threshold A precision parameter of the wrapping operator: higher values result in possibly improved precision.

wrap_individually `true` if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

std::invalid_argument Thrown if `*pcs` is dimension-incompatible with `vars`, or if `*this` is dimension-incompatible `vars` or with `*pcs`.

10.49.3.45 void Parma_Polyhedra_Library::Polyhedron::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates.

Parameters

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.49.3.46 void Parma_Polyhedra_Library::Polyhedron::drop_some_non_integer_points (const Variables_Set & *vars*, Complexity_Class *complexity* = ANY_COMPLEXITY) [inline]

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `vars`.

Parameters

vars Points with non-integer coordinates for these variables/space-dimensions can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.49.3.47 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.

10.49.3.48 void Parma_Polyhedra_Library::Polyhedron::limited_BHRZ03_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0)

Assigns to `*this` the result of computing the limited extrapolation between `*this` and `y` using the BHRZ03-widening operator.

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.

10.49.3.49 void Parma_Polyhedra_Library::Polyhedron::bounded_BHRZ03_extrapolation_-assign (const Polyhedron & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0)

Assigns to `*this` the result of computing the bounded extrapolation between `*this` and `y` using the BHRZ03-widening operator.

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.

10.49.3.50 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.

10.49.3.51 void Parma_Polyhedra_Library::Polyhedron::limited_H79_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0)

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

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.

10.49.3.52 void Parma_Polyhedra_Library::Polyhedron::bounded_H79_extrapolation_assign (const Polyhedron & *y*, const Constraint_System & *cs*, unsigned ∗ *tp* = 0)

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

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.

10.49.3.53 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

$$\{\, (x, y, z)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{P} \,\}.$$

10.49.3.54 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

$$\{ (x, y, 0)^{\mathrm{T}} \in \mathbb{R}^3 \mid (x, y)^{\mathrm{T}} \in \mathcal{P} \}.$$

10.49.3.55 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()`.

10.49.3.56 void Parma_Polyhedra_Library::Polyhedron::remove_space_dimensions (const Variables_Set & *vars*)

Removes all the specified dimensions from the vector space.

Parameters

vars 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 `vars`.

10.49.3.57 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`.

10.49.3.58 template<typename Partial_Function > void Parma_Polyhedra_Library::Polyhedron::map_space_dimensions (const Partial_Function & *pfunc*) [inline]

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 type parameter 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.

10.49.3.59 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 $n, n+1, \dots, n+m-1$.

10.49.3.60 void Parma_Polyhedra_Library::Polyhedron::fold_space_dimensions (const Variables_Set & *vars*, Variable *dest*)

Folds the space dimensions in `vars` into `dest`.

Parameters

vars The set of Variable objects corresponding to the space dimensions to be folded;

dest 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 `dest` or with one of the Variable objects contained in `vars`. Also thrown if `dest` is contained in `vars`.

If `*this` has space dimension n, with $n > 0$, `dest` has space dimension $k \leq n$, `vars` is a set of variables whose maximum space dimension is also less than or equal to n, and `dest` is not a member of `vars`, then the space dimensions corresponding to variables in `vars` are folded into the k-th space dimension.

10.49.3.61 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.

10.49.3.62 int32_t Parma_Polyhedra_Library::Polyhedron::hash_code () const `[inline]`

Returns a 32-bit hash code for `*this`.

If `x` and `y` are such that `x == y`, then `x.hash_code() == y.hash_code()`.

10.49.3.63 void Parma_Polyhedra_Library::Polyhedron::drop_some_non_integer_points (const Variables_Set * *pvars*, Complexity_Class *complexity*) `[protected]`

Possibly tightens `*this` by dropping some points with non-integer coordinates for the space dimensions corresponding to `*pvars`.

Parameters

pvars When nonzero, points with non-integer coordinates for the variables/space-dimensions contained in `*pvars` can be discarded.

complexity The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if `complexity` is `ANY_COMPLEXITY`.

10.49.4 Friends And Related Function Documentation

10.49.4.1 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 ", ".

10.49.4.2 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.

10.49.4.3 void swap (Parma_Polyhedra_Library::Polyhedron & *x*, Parma_Polyhedra_Library::Polyhedron & *y*) [related]

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.50 Parma_Polyhedra_Library::Powerset< D > Class Template Reference

The powerset construction on a base-level domain.

`#include <ppl.hh>`

Public Types

- typedef iterator_to_const< Sequence > iterator

  Alias for a read-only bidirectional iterator on the disjuncts of a Powerset element.

- typedef const_iterator_to_const< Sequence > 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 Object

- 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 in `*this`.

- 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 Object

- 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`.

- bool upper_bound_assign_if_exact (const Powerset &y)

  Assigns to `*this` the least upper bound of `*this` and `y` and returns `true`.

- 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_preserve_reduction (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_preserve_reduction (const D &d)

  Adds to `*this` the disjunct `d`, assuming `d` is not the bottom element and preserving Omega-reduction.

- 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.)

- template<typename D >
  bool operator== (const Powerset< D > &x, const Powerset< D > &y)

  Returns `true` *if and only if* `x` *and* `y` *are equivalent.*

- template<typename D >
  bool operator!= (const Powerset< D > &x, const Powerset< D > &y)

  Returns `true` *if and only if* `x` *and* `y` *are not equivalent.*

- template<typename D >
  std::ostream & operator<< (std::ostream &s, const Powerset< D > &x)

  Output operator.

- template<typename D >
  void swap (Parma_Polyhedra_Library::Powerset< D > &x, Parma_Polyhedra_Library::Powerset< D > &y)

  Specializes `std::swap`*.*

10.50.1 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 template 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.

10.50.2 Member Typedef Documentation

10.50.2.1 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.

10.50.2.2 template<typename D> typedef iterator_to_const<Sequence> 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.

10.50.3 Member Function Documentation

10.50.3.1 template<typename D > void Parma_Polyhedra_Library::Powerset< D >::omega_reduce () const [inline]

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.

10.50.3.2 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`.

10.50.3.3 template<typename D > bool Parma_Polyhedra_Library::Powerset< D >::upper_bound_assign_if_exact (const Powerset< D > & *y*) [inline]

Assigns to `*this` the least upper bound of `*this` and `y` and returns `true`.

Exceptions

std::invalid_argument Thrown if `*this` and `y` are dimension-incompatible.

10.50.3.4 template<typename D> Powerset< D >::iterator Parma_Polyhedra_Library::Powerset< D >::add_non_bottom_disjunct_preserve_reduction (const D & *d*, iterator *first*, iterator *last*) [inline, 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. If `*this` is reduced before an invocation of this method, it will be reduced upon successful return from the method.

10.50.3.5 template<typename D> void Parma_Polyhedra_Library::Powerset< D >::add_non_bottom_disjunct_preserve_reduction (const D & *d*) [inline, protected]

Adds to `*this` the disjunct `d`, assuming `d` is not the bottom element and preserving Omega-reduction.

If `*this` is reduced before an invocation of this method, it will be reduced upon successful return from the method.

10.50.3.6 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*) `[inline, 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.

10.50.4 Friends And Related Function Documentation

10.50.4.1 template< typename D > bool operator== (const Powerset< D > & *x*, const Powerset< D > & *y*) `[related]`

Returns `true` if and only if `x` and `y` are equivalent.

10.50.4.2 template< typename D > bool operator!= (const Powerset< D > & *x*, const Powerset< D > & *y*) `[related]`

Returns `true` if and only if `x` and `y` are not equivalent.

10.50.4.3 template< typename D > std::ostream & operator<< (std::ostream & *s*, const Powerset< D > & *x*) `[related]`

Output operator.

10.50.4.4 template< typename D > void swap (Parma_Polyhedra_Library::Powerset< D > & *x*, Parma_Polyhedra_Library::Powerset< D > & *y*) `[related]`

Specializes `std::swap`.

The documentation for this class was generated from the following file:

- ppl.hh

10.51 Parma_Polyhedra_Library::Recycle_Input Struct Reference

A tag class.

`#include <ppl.hh>`

10.51.1 Detailed Description

A tag class. Tag class to distinguish those constructors that recycle the data structures of their arguments, instead of taking a copy.

The documentation for this struct was generated from the following file:

- ppl.hh

10.52 Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 > Class Template Reference

This class provides the reduction method for the Shape_Preserving_Product domain.

```
#include <ppl.hh>
```

Public Member Functions

- Shape_Preserving_Reduction ()

  Default constructor.

- void product_reduce (D1 &d1, D2 &d2)

  The congruences reduction operator for detect emptiness or any equalities implied by each of the congruences defining one of the components and the bounds of the other component. It is assumed that the components are already constraints reduced.

- ~Shape_Preserving_Reduction ()

  Destructor.

10.52.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 >

This class provides the reduction method for the Shape_Preserving_Product domain. The reduction classes are used to instantiate the Partially_Reduced_Product domain.

This reduction method includes the congruences reduction. This class uses the minimized constraints defining each of the components. For each of the constraints, it checks the frequency and value for the same linear expression in the other component. If the constraint does not satisfy the implied congruence, the inhomogeneous term is adjusted so that it does. Note that unless the congruences reduction adds equalitites the shapes of the domains are unaltered.

10.52.2 Member Function Documentation

10.52.2.1 template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 >::product_reduce (D1 & *d1*, D2 & *d2*) `[inline]`

The congruences reduction operator for detect emptiness or any equalities implied by each of the congruences defining one of the components and the bounds of the other component. It is assumed that the components are already constraints reduced.

The minimized congruence system defining the domain element `d1` is used to check if `d2` intersects none, one or more than one of the hyperplanes defined by the congruences: if it intersects none, then product is set empty; if it intersects one, then the equality defining this hyperplane is added to both components; otherwise, the product is unchanged. In each case, the donor domain must provide a congruence system in minimal form.

Parameters

d1 A pointset domain element;

d2 A pointset domain element;

The documentation for this class was generated from the following file:

- ppl.hh

10.53 Parma_Polyhedra_Library::Smash_Reduction< D1, D2 > Class Template Reference

This class provides the reduction method for the Smash_Product domain.

`#include <ppl.hh>`

Public Member Functions

- Smash_Reduction ()

  Default constructor.

- void product_reduce (D1 &d1, D2 &d2)

  The smash reduction operator for propagating emptiness between the domain elements `d1` and `d2`.

- ~Smash_Reduction ()

  Destructor.

10.53.1 Detailed Description

template<typename D1, typename D2> class Parma_Polyhedra_Library::Smash_Reduction< D1, D2 >

This class provides the reduction method for the Smash_Product domain. The reduction classes are used to instantiate the Partially_Reduced_Product domain. This class propagates emptiness between its components.

10.53.2 Member Function Documentation

10.53.2.1 template<typename D1 , typename D2 > void Parma_Polyhedra_-Library::Smash_Reduction< D1, D2 >::product_reduce (D1 & *d1*, D2 & *d2*) `[inline]`

The smash reduction operator for propagating emptiness between the domain elements `d1` and `d2`.

If either of the the domain elements `d1` or `d2` is empty then the other is also set empty.

Parameters

d1 A pointset domain element;

d2 A pointset domain element;

The documentation for this class was generated from the following file:

- ppl.hh

10.54 Parma_Polyhedra_Library::Throwable Class Reference

User objects the PPL can throw.

`#include <ppl.hh>`

Public Member Functions

- virtual void throw_me () const =0

  Throws the user defined exception object.

- virtual ~Throwable ()

  Virtual destructor.

10.54.1 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.

The documentation for this class was generated from the following file:

- ppl.hh

10.55 Parma_Polyhedra_Library::Variable Class Reference

A dimension of the vector space.

`#include <ppl.hh>`

Classes

- struct Compare

    Binary predicate defining the total ordering on variables.

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.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Variable &v)

  Output operator.

- bool less (Variable v, Variable w)

  Defines a total ordering on variables.

10.55.1 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;
```

10.55.2 Constructor & Destructor Documentation

10.55.2.1 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 `i+1` exceeds `Variable::max_space_dimension()`.

10.55.3 Member Function Documentation

10.55.3.1 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`.

10.55.4 Friends And Related Function Documentation

10.55.4.1 std::ostream & operator<< (std::ostream & *s*, const Variable & *v*) [related]

Output operator.

10.55.4.2 bool less (Variable *v*, Variable *w*) [related]

Defines a total ordering on variables.

The documentation for this class was generated from the following file:

- ppl.hh

10.56 Parma_Polyhedra_Library::Variables_Set Class Reference

An std::set of variables' indexes.

`#include <ppl.hh>`

Public Member Functions

- Variables_Set ()
  Builds the empty set of variable indexes.
- Variables_Set (const Variable &v)
  Builds the singleton set of indexes containing `v.id();`.
- Variables_Set (const Variable &v, const Variable &w)
  Builds the set of variables's indexes in the range from `v.id()` to `w.id()`.
- dimension_type space_dimension () const
  Returns the dimension of the smallest vector space enclosing all the variables whose indexes are in the set.
- void insert (Variable v)
  Inserts the index of variable `v` into the set.
- bool ascii_load (std::istream &s)
  Loads from `s` an ASCII representation (as produced by ascii_dump(std::ostream&) const) and sets `*this` accordingly. Returns `true` if successful, `false` otherwise.
- 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.

- void ascii_dump () const

  Writes to `std::cerr` *an ASCII representation of* `*this`.

- void ascii_dump (std::ostream &s) const

  Writes to `s` *an ASCII representation of* `*this`.

- void print () const

  Prints `*this` *to* `std::cerr` *using* `operator<<`.

Static Public Member Functions

- static dimension_type max_space_dimension ()

  Returns the maximum space dimension a Variables_Set can handle.

Related Functions

(Note that these are not member functions.)

- std::ostream & operator<< (std::ostream &s, const Variables_Set &v)

  Output operator.

10.56.1 Detailed Description

An std::set of variables' indexes.

10.56.2 Constructor & Destructor Documentation

10.56.2.1 Parma_Polyhedra_Library::Variables_Set::Variables_Set (const Variable & *v*, const Variable & *w*)

Builds the set of variables's indexes in the range from `v.id()` to `w.id()`.

If `v.id() <= w.id()`, this constructor builds the set of variables' indexes `v.id()`, `v.id()+1`, ..., `w.id()`. The empty set is built otherwise.

10.56.3 Friends And Related Function Documentation

10.56.3.1 std::ostream & operator<< (std::ostream & *s*, const Variables_Set & *v*) `[related]`

Output operator.

The documentation for this class was generated from the following file:

- ppl.hh

Index