

The Parma Polyhedra Library
User's Manual*
(version 1.2pre4)

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

September 27, 2014

*This work is based on previous work also by Elisa Ricci, Sara Bonini, Andrea Pescetti, Angela Stazzone, Tatiana Zolo. 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, and BUGSENG srl.

[‡]patricia.hill@bugseng.com, BUGSENG srl.

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

[¶]abramo.bagnara@bugseng.com, BUGSENG srl.

Copyright © 2001–2010 Roberto Bagnara (bagnara@cs.unipr.it)
Copyright © 2010–2014 BUGSENG srl (<http://bugseng.com>)

This document describes the Parma Polyhedra Library (PPL).

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**”.

The PPL 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. A copy of the license is included in the section entitled “**GNU GENERAL PUBLIC LICENSE**”.

The PPL 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.

If you have not received a copy of one or both the above mentioned licenses along with the PPL, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02111-1307, USA.

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

<http://bugseng.com/products/ppl/>



Contents

1	General Information on the PPL	1
1.1	The Main Features	1
1.2	Upward Approximation	5
1.3	Approximating Integers	6
1.4	Convex Polyhedra	7
1.5	Representations of Convex Polyhedra	8
1.6	Operations on Convex Polyhedra	11
1.7	Intervals and Boxes	18
1.8	Weakly-Relational Shapes	19
1.9	Rational Grids	20
1.10	Operations on Rational Grids	22
1.11	The Powerset Construction	24
1.12	Operations on the Powerset Construction	25
1.13	The Pointset Powerset Domain	26
1.14	Analysis of floating point computations	27
1.15	Using the Library	29
1.16	Bibliography	30
2	GNU General Public License	38
3	GNU Free Documentation License	46
4	Module Index	51
4.1	Modules	51
5	Namespace Index	51
5.1	Namespace List	51
6	Hierarchical Index	51
6.1	Class Hierarchy	51
7	Class Index	55
7.1	Class List	55
8	Module Documentation	61
8.1	C++ Language Interface	61
9	Namespace Documentation	78
9.1	Parma_Polyhedra_Library Namespace Reference	78
9.2	Parma_Polyhedra_Library::IO_Operators Namespace Reference	86
9.3	std Namespace Reference	87
10	Class Documentation	88
10.1	Parma_Polyhedra_Library::Approximable_Reference< Target > Class Template Reference	88
10.2	Parma_Polyhedra_Library::Approximable_Reference_Common< Target > Class Template Reference	88
10.3	Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter Class Reference	88
10.4	Parma_Polyhedra_Library::BD_Shape< T > Class Template Reference	90
10.5	Parma_Polyhedra_Library::BHRZ03_Certificate Class Reference	120
10.6	Parma_Polyhedra_Library::Binary_Operator< Target > Class Template Reference	121
10.7	Parma_Polyhedra_Library::Binary_Operator_Common< Target > Class Template Reference	121
10.8	Parma_Polyhedra_Library::Box< Interval > Class Template Reference	122



10.9	Parma_Polyhedra_Library::C_Polyhedron Class Reference	152
10.10	Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference	156
10.11	Parma_Polyhedra_Library::Cast_Operator< Target > Class Template Reference	159
10.12	Parma_Polyhedra_Library::Cast_Operator_Common< Target > Class Template Reference	159
10.13	Parma_Polyhedra_Library::Checked_Number< T, Policy > Class Template Reference	160
10.14	Parma_Polyhedra_Library::Variable::Compare Struct Reference	175
10.15	Parma_Polyhedra_Library::BHRZ03_Certificate::Compare Struct Reference	175
10.16	Parma_Polyhedra_Library::H79_Certificate::Compare Struct Reference	176
10.17	Parma_Polyhedra_Library::Grid_Certificate::Compare Struct Reference	176
10.18	Parma_Polyhedra_Library::Concrete_Expression< Target > Class Template Reference	176
10.19	Parma_Polyhedra_Library::Concrete_Expression_Common< Target > Class Template Ref- erence	182
10.20	Parma_Polyhedra_Library::Concrete_Expression_Type Class Reference	182
10.21	Parma_Polyhedra_Library::Congruence Class Reference	184
10.22	Parma_Polyhedra_Library::Congruence_System Class Reference	191
10.23	Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 > Class Template Reference	196
10.24	Parma_Polyhedra_Library::CO_Tree::const_iterator Class Reference	197
10.25	Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator Class Ref- erence	200
10.26	Parma_Polyhedra_Library::Linear_Expression::const_iterator Class Reference	201
10.27	Parma_Polyhedra_Library::Congruence_System::const_iterator Class Reference	203
10.28	Parma_Polyhedra_Library::MIP_Problem::const_iterator Class Reference	204
10.29	Parma_Polyhedra_Library::Grid_Generator_System::const_iterator Class Reference	205
10.30	Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface Class Ref- erence	206
10.31	Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, F- P_Format > Class Template Reference	208
10.32	Parma_Polyhedra_Library::Constraint Class Reference	210
10.33	Parma_Polyhedra_Library::Constraint_System Class Reference	219
10.34	Parma_Polyhedra_Library::Constraint_System_const_iterator Class Reference	222
10.35	Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 > Class Template Reference	223
10.36	Parma_Polyhedra_Library::Determinate< PSET > Class Template Reference	224
10.37	Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, F- P_Format > Class Template Reference	226
10.38	Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP- Format > Class Template Reference	229
10.39	Parma_Polyhedra_Library::Domain_Product< D1, D2 > Class Template Reference	232
10.40	Parma_Polyhedra_Library::Implementation::Doubly_Linked_Object Class Reference	232
10.41	Parma_Polyhedra_Library::Implementation::EList< T > Class Template Reference	233
10.42	Parma_Polyhedra_Library::Implementation::EList_Iterator< T > Class Template Refer- ence	235
10.43	Parma_Polyhedra_Library::Floating_Point_Constant< Target > Class Template Reference	236
10.44	Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target > Class Template Reference	236
10.45	Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference	236
10.46	Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type > Class Template Ref- erence	241
10.47	Parma_Polyhedra_Library::Generator Class Reference	242
10.48	Parma_Polyhedra_Library::Generator_System Class Reference	255
10.49	Parma_Polyhedra_Library::Generator_System_const_iterator Class Reference	259
10.50	Parma_Polyhedra_Library::GMP_Integer Class Reference	260
10.51	Parma_Polyhedra_Library::Grid Class Reference	262

10.52	Parma_Polyhedra_Library::Grid_Certificate Class Reference	291
10.53	Parma_Polyhedra_Library::Grid_Generator Class Reference	292
10.54	Parma_Polyhedra_Library::Grid_Generator_System Class Reference	300
10.55	Parma_Polyhedra_Library::H79_Certificate Class Reference	304
10.56	Parma_Polyhedra_Library::Implementation::Watchdog::Handler Class Reference	305
10.57	Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag > Class Template Reference	306
10.58	Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function Class Reference	307
10.59	Parma_Polyhedra_Library::Integer_Constant< Target > Class Template Reference	308
10.60	Parma_Polyhedra_Library::Integer_Constant_Common< Target > Class Template Ref- erence	308
10.61	Parma_Polyhedra_Library::Interval< Boundary, Info > Class Template Reference	309
10.62	Parma_Polyhedra_Library::CO_Tree::iterator Class Reference	312
10.63	Parma_Polyhedra_Library::Linear_Expression Class Reference	315
10.64	Parma_Polyhedra_Library::Linear_Form< C > Class Template Reference	326
10.65	Parma_Polyhedra_Library::MIP_Problem Class Reference	335
10.66	Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference	343
10.67	Parma_Polyhedra_Library::NNC_Polyhedron Class Reference	346
10.68	Parma_Polyhedra_Library::PIP_Solution_Node::No_Constraints Struct Reference	350
10.69	Parma_Polyhedra_Library::No_Reduction< D1, D2 > Class Template Reference	351
10.70	Parma_Polyhedra_Library::Octagonal_Shape< T > Class Template Reference	351
10.71	Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, F- P_Format > Class Template Reference	385
10.72	Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R > Class Template Reference	387
10.73	Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< Threshold > Class Template Reference	412
10.74	Parma_Polyhedra_Library::Implementation::Watchdog::Pending_List< Traits > Class Tem- plate Reference	413
10.75	Parma_Polyhedra_Library::PIP_Decision_Node Class Reference	414
10.76	Parma_Polyhedra_Library::PIP_Problem Class Reference	415
10.77	Parma_Polyhedra_Library::PIP_Solution_Node Class Reference	426
10.78	Parma_Polyhedra_Library::PIP_Tree_Node Class Reference	429
10.79	Parma_Polyhedra_Library::Pointset_Powerset< PSET > Class Template Reference	433
10.80	Parma_Polyhedra_Library::Poly_Con_Relation Class Reference	455
10.81	Parma_Polyhedra_Library::Poly_Gen_Relation Class Reference	456
10.82	Parma_Polyhedra_Library::Polyhedron Class Reference	458
10.83	Parma_Polyhedra_Library::Powerset< D > Class Template Reference	491
10.84	Parma_Polyhedra_Library::Recycle_Input Struct Reference	497
10.85	Parma_Polyhedra_Library::Select_Temp_Boundary_Type< Interval_Boundary_Type > Struct Template Reference	497
10.86	Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 > Class Template Ref- erence	498
10.87	Parma_Polyhedra_Library::Smash_Reduction< D1, D2 > Class Template Reference	499
10.88	Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference	499
10.89	Parma_Polyhedra_Library::Threshold_Watcher< Traits > Class Template Reference	502
10.90	Parma_Polyhedra_Library::Throwable Class Reference	502
10.91	Parma_Polyhedra_Library::Implementation::Watchdog::Time Class Reference	503
10.92	Parma_Polyhedra_Library::Unary_Operator< Target > Class Template Reference	503
10.93	Parma_Polyhedra_Library::Unary_Operator_Common< Target > Class Template Refer- ence	504
10.94	Parma_Polyhedra_Library::Variable Class Reference	504

10.95	Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP- _Format > Class Template Reference	506
10.96	Parma_Polyhedra_Library::Variables_Set Class Reference	509
10.97	Parma_Polyhedra_Library::Watchdog Class Reference	510

Index	511
--------------	------------

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 $x + 2*y + 5*z \leq 7$ 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).

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.

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.

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()`, `m_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

```
SemanticGD.add_constraint(c)
```

raises an exception, a fragment of the form

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

may give more precise results than a single

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

- `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-integer 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

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

For signed integers the wrapping function is, instead,

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

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.

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 $\{\text{wrap}_w^u(z) \mid z \in S\}$. 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 $\mathbf{v} = (v_0, \dots, v_{n-1})^T \in \mathbb{R}^n$. We denote by $\mathbf{0}$ the vector of \mathbb{R}^n , called *the origin*, having all components equal to zero. A vector $\mathbf{v} \in \mathbb{R}^n$ 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 \mathbf{v}^T .

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

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

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

1.4.2 Affine Hyperplanes and Half-spaces

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

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 $\mathcal{P} = \emptyset$. The set of all NNC polyhedra on the vector space \mathbb{R}^n is denoted \mathbb{P}_n .

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

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

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

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:

$$\mathcal{P} \subseteq \{ \mathbf{x} \in \mathbb{R}^n \mid -\lambda \leq x_j \leq \lambda \text{ for } j = 0, \dots, n-1 \}.$$

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

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

where, for all $i \in \{1, 2, 3\}$, $A_i \in \mathbb{R}^{m_i} \times \mathbb{R}^n$ and $\mathbf{b}_i \in \mathbb{R}^{m_i}$, 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 $S = \{x_1, \dots, x_k\} \subseteq \mathbb{R}^n$ be a finite set of vectors. For all scalars $\lambda_1, \dots, \lambda_k \in \mathbb{R}$, the vector $v = \sum_{j=1}^k \lambda_j x_j$ 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 $\forall j \in \{1, \dots, k\} : \lambda_j \in \mathbb{R}_+$;
- an *affine* combination, if $\sum_{j=1}^k \lambda_j = 1$;
- a *convex* combination, if it is both positive and affine.

We denote by $\text{linear.hull}(S)$ (resp., $\text{conic.hull}(S)$, $\text{affine.hull}(S)$, $\text{convex.hull}(S)$) 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 $\text{nnc.hull}(P, C)$ the set of all convex combinations of the vectors in S such that $\lambda_j > 0$ for some $x_j \in P$ (informally, we say that there exists a vector of P that plays an active role in the convex combination). Note that $\text{nnc.hull}(P, C) = \text{nnc.hull}(P, P \cup C)$ so that, if $C \subseteq P$,

$$\text{convex.hull}(P) = \text{nnc.hull}(P, \emptyset) = \text{nnc.hull}(P, P) = \text{nnc.hull}(P, C).$$

It can be observed that $\text{linear.hull}(S)$ is an affine space, $\text{conic.hull}(S)$ is a topologically closed convex cone, $\text{convex.hull}(S)$ is a topologically closed polytope, and $\text{nnc.hull}(P, C)$ 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 $p \in \mathcal{P}$ is called a *point* of \mathcal{P} ;
- a vector $c \in \mathbb{R}^n$ is called a *closure point* of \mathcal{P} if it is a point of the topological closure of \mathcal{P} ;
- a vector $r \in \mathbb{R}^n$, where $r \neq 0$, is called a *ray* (or *direction of infinity*) of \mathcal{P} if $\mathcal{P} \neq \emptyset$ and $p + \lambda r \in \mathcal{P}$, for all points $p \in \mathcal{P}$ and all $\lambda \in \mathbb{R}_+$;
- a vector $l \in \mathbb{R}^n$ is called a *line* of \mathcal{P} if both l and $-l$ 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 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 r_1 and r_2 of rays of \mathcal{P} , where $r \neq \lambda r_1$, $r \neq \lambda r_2$ and $r_1 \neq \lambda r_2$ 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

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

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

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

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 $\mathcal{P} = \emptyset$ if and only if $P = \emptyset$. 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 $L = \{(1, 0)^T\}$, $R = \{(0, 1)^T\}$, $P = \{(0, 0)^T\}$, and $C = \emptyset$. 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 $\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b$ where $\bowtie \in \{=, \geq, >\}$ and $\mathbf{a}, \mathbf{x} \in \mathbb{R}^m$, is dimension-compatible with a polyhedron having space dimension n if and only if $m \leq n$;

- the generator $\mathbf{x} \in \mathbb{R}^m$ 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 $\{\mathbf{x}_1, \dots, \mathbf{x}_k\} \subseteq \mathbb{R}^n$ is *affinely independent* if, for all $\lambda_1, \dots, \lambda_k \in \mathbb{R}$, the system of equations

$$\sum_{i=1}^k \lambda_i \mathbf{x}_i = \mathbf{0}, \quad \sum_{i=1}^k \lambda_i = 0$$

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

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

$$\mathcal{R} \stackrel{\text{def}}{=} \left\{ (x_0, \dots, x_{n-1}, y_0, \dots, y_{m-1})^T \in \mathbb{R}^{n+m} \mid (x_0, \dots, x_{n-1})^T \in \mathcal{P}, (y_0, \dots, y_{m-1})^T \in \mathcal{Q} \right\}.$$

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

$$\mathcal{Q} = \{ (x_0, x_1, x_2)^T \in \mathbb{R}^3 \mid (x_0, x_1)^T \in \mathcal{P} \}.$$

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

$$\mathcal{Q} = \{ (x_0, x_1, 0)^T \in \mathbb{R}^3 \mid (x_0, x_1)^T \in \mathcal{P} \}.$$

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 $\{(3, 1, 0, 2)^T\} \subseteq \mathbb{R}^4$, then after invoking this operator with the set of variables $\{x_1, x_2\}$ the resulting polyhedron is

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

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

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

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 $\rho: \{0, \dots, n-1\} \rightarrow \mathbb{N}$ such that $\rho(\{0, \dots, n-1\}) = \{0, \dots, m-1\}$ 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

$$\mathcal{Q} \stackrel{\text{def}}{=} \left\{ (v_{\rho^{-1}(0)}, \dots, v_{\rho^{-1}(m-1)})^T \mid (v_0, \dots, v_{n-1})^T \in \mathcal{P} \right\}.$$

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 $n, n+1, \dots, n+m-1$ of the result \mathcal{Q} are exact copies of the i -th space dimension of \mathcal{P} . More formally,

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

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

$$\mathcal{Q} \stackrel{\text{def}}{=} \bigoplus_{d=0}^m \mathcal{Q}_d$$

where

$$\mathcal{Q}_m \stackrel{\text{def}}{=} \left\{ \mathbf{u} \in \mathbb{R}^{n-m} \mid \begin{array}{l} \exists \mathbf{v} \in \mathcal{P} . u_{i'} = v_i \\ \wedge \forall k = 0, \dots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\}$$

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

$$\mathcal{Q}_d \stackrel{\text{def}}{=} \left\{ \mathbf{u} \in \mathbb{R}^{n-m} \mid \begin{array}{l} \exists \mathbf{v} \in \mathcal{P} . u_{i'} = v_{j_d} \\ \wedge \forall k = 0, \dots, n-1 : k \neq i \implies u_{k'} = v_k \end{array} \right\},$$

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

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

($\#S$ 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,

$$\phi(S) \stackrel{\text{def}}{=} \{ \mathbf{w} \in \mathbb{R}^m \mid \exists \mathbf{v} \in S . (\mathbf{v}, \mathbf{w}) \in \phi \}.$$

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

$$\phi^{-1}(S') \stackrel{\text{def}}{=} \{ \mathbf{v} \in \mathbb{R}^n \mid \exists \mathbf{w} \in S' . (\mathbf{v}, \mathbf{w}) \in \phi \}.$$

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

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

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

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 $\mathbf{b} \in \mathbb{R}^m$ such that,

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

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 $\mathbf{x} = (x_0, \dots, x_{n-1})^T$ of *unprimed* variables is used to represent the space dimensions of the domain of ϕ ;
- the vector $\mathbf{x}' = (x'_0, \dots, x'_{n-1})^T$ 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

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

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 $\langle \mathbf{a}, \mathbf{x} \rangle + b$, the *affine function* $\phi = (x'_k = \langle \mathbf{a}, \mathbf{x} \rangle + b) : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is defined by

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

where

$$A = \begin{pmatrix} 1 & & 0 & 0 & \dots & \dots & 0 \\ & \ddots & & \vdots & & & \vdots \\ 0 & & 1 & 0 & \dots & \dots & 0 \\ a_0 & \dots & a_{k-1} & a_k & a_{k+1} & \dots & a_{n-1} \\ 0 & \dots & \dots & 0 & 1 & \dots & 0 \\ \vdots & & & \vdots & & \ddots & \\ 0 & \dots & \dots & 0 & 0 & & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

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

$$\left(v_0, \dots, \left(\sum_{i=0}^{n-1} a_i v_i + b \right), \dots, v_{n-1} \right)^T.$$

The *affine image* operator computes the affine image of a polyhedron \mathcal{P} under $x'_k = \langle \mathbf{a}, \mathbf{x} \rangle + b$. For instance, suppose the polyhedron \mathcal{P} to be transformed is the square in \mathbb{R}^2 generated by the set of points $\{(0, 0)^T, (0, 3)^T, (3, 0)^T, (3, 3)^T\}$. 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 $\{(4, 0)^T, (10, 3)^T, (7, 0)^T, (13, 3)^T\}$ 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 $x'_k = \langle \mathbf{a}, \mathbf{x} \rangle + b$. 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 $(1, 0)^T$ 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 $\text{lb} = \langle \mathbf{a}, \mathbf{x} \rangle + b$ and $\text{ub} = \langle \mathbf{c}, \mathbf{x} \rangle + d$, the *bounded affine relation* $\phi = (\text{lb} \leq x'_k \leq \text{ub})$ is defined as

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

1.6.12 Affine Form Relations.

Let \mathbb{F}_f be the set of floating point numbers representables in a certain format f and let \mathbb{I}_f be the set of real intervals with bounds in \mathbb{F}_f . We can define a *floating-point interval linear form* $\langle \boldsymbol{\alpha}, \mathbf{x} \rangle + \beta$ as:

$$\langle \boldsymbol{\alpha}, \mathbf{x} \rangle + \beta = \alpha_0 x_0 + \dots + \alpha_{n-1} x_{n-1} + \beta,$$

where $\beta, \alpha_k \in \mathbb{I}_f$ for each $k = 0, \dots, n-1$.

Given a such linear form lf and a primed variable x'_k the *affine form image* operator computes the bounded affine image of a polyhedron \mathcal{P} under $\text{lb} \leq x'_k \leq \text{ub}$, where lb and ub are the upper and lower bound of lf respectively.

1.6.13 Generalized Affine Relations.

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

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

When $\text{lhs} = x_k$ and $\bowtie \in \{=\}$, then the above affine relation becomes equivalent to the single-update affine function $x'_k = \text{rhs}$ (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 $\text{lhs}' \leq \text{rhs}$ and $\text{rhs}' \geq \text{lhs}$ are not equivalent in general.

1.6.14 Cylindrification Operator

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

$$Q = \{ w \in \mathbb{R}^n \mid \exists v \in P . \forall j \in \{0, \dots, n-1\} : j \neq i \implies w_j = v_j \}.$$

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.

1.6.15 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 $P, Q \in \mathbb{P}_n$, the time-elapse between P and Q , denoted $P \nearrow Q$, is the smallest NNC polyhedron containing the set

$$\{ p + \lambda q \in \mathbb{R}^n \mid p \in P, q \in Q, \lambda \in \mathbb{R}_+ \}.$$

Note that the above set might not be an NNC polyhedron.

1.6.16 Positive Time-Elapse Operator

The *positive time-elapse* operator has been defined in [BFM11,BFM13]. The operator provided by the library works on NNC polyhedra. For any two NNC polyhedra $P, Q \in \mathbb{P}_n$, the positive time-elapse between P and Q , denoted $P \nearrow_{>0} Q$, is the NNC polyhedron containing exactly the set

$$\{ p + \lambda q \in \mathbb{R}^n \mid p \in P, q \in Q, \lambda \in \mathbb{R}^{>0} \},$$

where $\mathbb{R}^{>0}$ denotes the set of strictly positive reals. Notice that, differently from the case of the time-elapse operator, the set $P \nearrow_{>0} Q$ is always an NNC polyhedron, if P and Q are.

The exact version of the time-elapse operator $P \nearrow Q$ defined in Section [Time-Elapse Operator](#), which may not be an NNC polyhedron, can be computed as the union of two NNC polyhedra, according to the following equation: $P \nearrow Q = P \cup (P \nearrow_{>0} Q)$.

1.6.17 Meet-Preserving Enlargement and Simplification

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

- R is *meet-preserving* with respect to P using context Q if $R \cap Q = P \cap Q$;
- R is an *enlargement* of P if $R \supseteq P$.
- R is a *simplification* with respect to P if $r \leq p$, where r and p are the cardinalities of minimized constraint representations for R and 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](#).

1.6.18 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 $c = (\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b)$ 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 $\mathcal{P} \cap \mathcal{Q} = \emptyset$; that is, adding c to \mathcal{C} gives us the empty polyhedron.
- \mathcal{P} *strictly intersects* c if $\mathcal{P} \cap \mathcal{Q} \neq \emptyset$ 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 $\langle \mathbf{a}, \mathbf{x} \rangle = b$; that is, adding the constraint $\langle \mathbf{a}, \mathbf{x} \rangle = b$ 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} .

1.6.19 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.

1.6.20 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* [B-HRZ03a]. 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.

1.6.21 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 $\mathbf{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, and for each relation symbol $\bowtie \in \{=, \geq, >\}$, the constraint $\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b$ is said to be a *interval constraint* if there exist an index $i \in \{0, \dots, n-1\}$ such that, for all $k \in \{0, \dots, i-1, i+1, \dots, n-1\}$, $a_k = 0$. 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 $\mathbf{e}_i = (0, \dots, 1, \dots, 0)^T$ 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 $\langle \mathbf{e}_i, \mathbf{x} \rangle \bowtie b$, 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 $\langle \mathbf{e}_i, \mathbf{x} \rangle \bowtie b$, 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 $\mathcal{P} = \emptyset$. 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 $\mathbf{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, and for each relation symbol $\bowtie \in \{=, \geq\}$, the linear constraint $\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b$ is said to be a *bounded difference* if there exist two indices $i, j \in \{0, \dots, n-1\}$ such that:

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

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

1.8.2 Octagonal Shapes

For each vector $\mathbf{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$, and for each relation symbol $\bowtie \in \{=, \geq\}$, the linear constraint $\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b$ is said to be an *octagonal* if there exist two indices $i, j \in \{0, \dots, n-1\}$ 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 $\mathcal{P} = \emptyset$.

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 \emptyset 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 $\mathbf{a} \in \mathbb{S}^n \setminus \{\mathbf{0}\}$ and scalars $b, f \in \mathbb{S}$, the notation $\langle \mathbf{a}, \mathbf{x} \rangle \equiv_f b$ stands for the *linear congruence relation in \mathbb{S}^n* defined by the set of vectors

$$\{ \mathbf{v} \in \mathbb{R}^n \mid \exists \mu \in \mathbb{Z} . \langle \mathbf{a}, \mathbf{v} \rangle = b + \mu f \};$$

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

$$\{ (\langle \mathbf{a}, \mathbf{x} \rangle = b + \mu f) \mid \mu \in \mathbb{Z} \};$$

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

1.9.2 Rational Grids

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

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

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

If the congruence system \mathcal{C} describes the \emptyset , the *empty grid*, then we say that \mathcal{C} is *inconsistent*. For example, the congruence systems $\{\langle \mathbf{0}, \mathbf{x} \rangle \equiv_0 1\}$ meaning that $0 = 1$ and $\{\langle \mathbf{a}, \mathbf{x} \rangle \equiv_2 0, \langle \mathbf{a}, \mathbf{x} \rangle \equiv_2 1\}$, for any $\mathbf{a} \in \mathbb{R}^n$, 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 \emptyset 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 $S = \{\mathbf{x}_1, \dots, \mathbf{x}_k\} \subseteq \mathbb{R}^n$ be a finite set of vectors. For all scalars $\mu_1, \dots, \mu_k \in \mathbb{Z}$, the vector $\mathbf{v} = \sum_{j=1}^k \mu_j \mathbf{x}_j$ is said to be a *integer combination* of the vectors in S .

We denote by $\text{int.hull}(S)$ (resp., $\text{int.affine.hull}(S)$) 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 $\mathbf{p} \in \mathcal{L}$ is called a *grid point* of \mathcal{L} ;
- a vector $\mathbf{q} \in \mathbb{R}^n$, where $\mathbf{q} \neq \mathbf{0}$, is called a *parameter* of \mathcal{L} if $\mathcal{L} \neq \emptyset$ and $\mathbf{p} + \mu\mathbf{q} \in \mathcal{L}$, for all points $\mathbf{p} \in \mathcal{L}$ and all $\mu \in \mathbb{Z}$;
- a vector $\mathbf{l} \in \mathbb{R}^n$ is called a *grid line* of \mathcal{L} if $\mathcal{L} \neq \emptyset$ and $\mathbf{p} + \lambda\mathbf{l} \in \mathcal{L}$, for all points $\mathbf{p} \in \mathcal{L}$ 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

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

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 $\mathcal{L} = \text{ggen}(L, Q, P)$.

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

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 $\text{expr} = \langle \mathbf{a}, \mathbf{x} \rangle + b$, these determine the affine transformation $\phi = (x'_k = \langle \mathbf{a}, \mathbf{x} \rangle + b) : \mathbb{R}^n \rightarrow \mathbb{R}^n$ that transforms any point $(v_0, \dots, v_{n-1})^T$ in a grid \mathcal{L} to

$$\left(v_0, \dots, \left(\sum_{i=0}^{n-1} a_i v_i + b \right), \dots, v_{n-1} \right)^T.$$

The *affine image* operator computes the affine image of a grid \mathcal{L} under $x'_k = \langle \mathbf{a}, \mathbf{x} \rangle + b$. 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 $\{(0, 0)^T, (0, 3)^T, (3, 0)^T\}$. 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 $\{(1, 0)^T, (7, 3)^T, (10, 0)^T\}$ 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 $\phi = (\text{lhs}', \text{rhs}, f) : \mathbb{R}^n \rightarrow \mathbb{R}^n$, where $\text{lhs} = \langle \mathbf{c}, \mathbf{x} \rangle + d$ and $\text{rhs} = \langle \mathbf{a}, \mathbf{x} \rangle + b$ are affine expressions and $f \in \mathbb{Q}$, is defined as

$$\forall \mathbf{v} \in \mathbb{R}^n, \mathbf{w} \in \mathbb{R}^n : (\mathbf{v}, \mathbf{w}) \in \phi \iff (\langle \mathbf{c}, \mathbf{w} \rangle + d \equiv_f \langle \mathbf{a}, \mathbf{v} \rangle + b) \wedge \left(\bigwedge_{0 \leq i < n, c_i = 0} w_i = v_i \right).$$

Note that, when $\text{lhs} = x_k$ 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 $\text{expr} = (\langle \mathbf{a}, \mathbf{x} \rangle + b)$ be a linear expression. Then if, for some $c, f \in \mathbb{R}$, all the points in \mathcal{L} satisfy the congruence $\text{cg} = (\text{expr} \equiv_f c)$, 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 $\text{val} = \langle \mathbf{a}, \mathbf{w} \rangle + b$ where $\mathbf{w} \in \mathcal{L}$ and

$$|\text{val}| = \min \left\{ |\langle \mathbf{a}, \mathbf{v} \rangle + b| \mid \mathbf{v} \in \mathcal{L} \right\}.$$

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 ($\text{expr} = c$); 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

$$\{ \mathbf{p} + \mu \mathbf{q} \in \mathbb{R}^n \mid \mathbf{p} \in \mathcal{L}_1, \mathbf{q} \in \mathcal{L}_2, \mu \in \mathbb{Z} \}.$$

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 $\text{cg} = (\langle \mathbf{a}, \mathbf{x} \rangle \equiv_f b)$ is a congruence relation with $\mathcal{L}_{\text{cg}} = \text{gcon}(\{\text{cg}\})$. The possible relations between \mathcal{L} and cg are as follows.

- \mathcal{L} is *disjoint* from cg if $\mathcal{L} \cap \mathcal{L}_{\text{cg}} = \emptyset$; that is, adding cg to \mathcal{C} gives us the empty grid.
- \mathcal{L} *strictly intersects* cg if $\mathcal{L} \cap \mathcal{L}_{\text{cg}} \neq \emptyset$ and $\mathcal{L} \cap \mathcal{L}_{\text{cg}} \subset \mathcal{L}$; 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 $\mathcal{L} \subseteq \mathcal{L}_{\text{cg}}$; 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 $c = (\langle \mathbf{a}, \mathbf{x} \rangle \bowtie b)$ 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 $\mathcal{L} \cap \mathcal{Q} = \emptyset$.
- \mathcal{L} *strictly intersects* c if $\mathcal{L} \cap \mathcal{Q} \neq \emptyset$ 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 $v_j = a \bmod 2^w$, for some $a \in \mathbb{R}$, then v_j is set equal to a' where $a' = a \bmod 2^w$ 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 $\wp_{\text{fn}}^+(D)$. The function $\Omega_D^+ : \wp_{\text{f}}(D) \rightarrow \wp_{\text{fn}}^+(D)$, called *Omega-reduction*, maps a finite set into its non-redundant counterpart; it is defined, for each $\mathcal{S} \in \wp_{\text{f}}(D)$, by

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

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

As the intended semantics of a powerset domain element $\mathcal{S} \in \wp_{\text{f}}(D)$ is that of disjunction of the semantics of D , the finite set \mathcal{S} is semantically equivalent to the non-redundant set $\Omega_D^+(\mathcal{S})$; 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 $\mathcal{S} \in \wp_{\text{f}}(D)$ such that $\mathcal{S} \neq \{\mathbf{0}\}$, $\Omega_D^+(\mathcal{S})$ 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_{P} ’ defined so that, for any \mathcal{S}_1 and $\mathcal{S}_2 \in D_{\text{P}}$, $\mathcal{S}_1 \vdash_{\text{P}} \mathcal{S}_2$ if and only if

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

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

Note

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

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

1.12.2 Adding a Disjunct

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

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 $\mathcal{S}_1 = \{d_1, \dots, d_m\}$, $\mathcal{S}_2 = \{c_1, \dots, c_n\}$ and $\mathcal{S} = \{s_1, \dots, s_q\}$ 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 [BG-P99 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 Analysis of floating point computations

This section describes the PPL abstract domains that are used for approximating floating point computations in software analysis. We follow the approach described in [Min04] and more detailedly in [Min05]. We will denote by \mathcal{V} the set of all floating point variables in the analyzed program. We will also denote by \mathbb{F}_a the set of floating point numbers in the format used by the analyzer (that is, the machine running the PPL) and by \mathbb{F}_t the set of floating point numbers in the format used by the machine that is expected to run the analyzed program. Recall that floating point numbers include the infinities $-\infty$ and $+\infty$.

1.14.1 Linear forms with interval coefficients

Generic concrete *floating point expressions* on \mathbb{F}_t are represented by the `Floating_Point_Expression` abstract class. Its concrete derivate classes are:

- `Cast_Floating_Point_Expression`,
- `Constant_Floating_Point_Expression`,
- `Variable_Floating_Point_Expression`,
- `Opposite_Floating_Point_Expression`, that is the negation (unary minus) of a floating point expression,
- `Sum_Floating_Point_Expression`, that is the sum of two floating point expressions,
- `Difference_Floating_Point_Expression`, that is the difference of two floating point expressions,

- `MultiplicationFloatingPointExpression`, that is the product of two floating point expressions, and
- `DivisionFloatingPointExpression`, that is the division of two floating point expressions.

The set of all the possible values in \mathbb{F}_t of a floating point expression at a given program point in a given abstract store can be overapproximated by a *linear form* with interval coefficients, that is a linear expression of this kind:

$$i + \sum_{v \in V} i_v v,$$

where all v are free floating point variables and i and all i_v are elements of \mathbb{I}_a , defined as the set of all intervals with boundaries in \mathbb{F}_a . This operation is called *linearization* and is performed by the method `linearize` of floating point expression classes.

Even though the intervals may be open, we will always use closed intervals in the documentation for the sake of simplicity, with the exception of unbounded intervals that have ∞ boundaries. We denote the set of all linear forms on \mathbb{F}_a by \mathbb{L}_a .

The `LinearForm` class provides common algebraic operations on linear forms: you can add or subtract two linear forms, and multiply or divide a linear form by a scalar. We are writing only about interval linear forms in this section, so our scalars will always be intervals with floating point boundaries. The operations on interval linear forms are intuitively defined as follows:

$$\begin{aligned} \left(i + \sum_{v \in V} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in V} i'_v v \right) &\stackrel{\text{def}}{=} (i \oplus^\# i') + \sum_{v \in V} (i_v \oplus^\# i'_v) v, \\ \left(i + \sum_{v \in V} i_v v \right) \boxminus^\# \left(i' + \sum_{v \in V} i'_v v \right) &\stackrel{\text{def}}{=} (i \ominus^\# i') + \sum_{v \in V} (i_v \ominus^\# i'_v) v, \\ i \boxtimes^\# \left(i' + \sum_{v \in V} i'_v v \right) &\stackrel{\text{def}}{=} (i \otimes^\# i') + \sum_{v \in V} (i \otimes^\# i'_v) v, \\ \left(i + \sum_{v \in V} i_v v \right) \boxdiv^\# i' &\stackrel{\text{def}}{=} (i \oslash^\# i') + \sum_{v \in V} (i_v \oslash^\# i') v. \end{aligned}$$

Where $\oplus^\#, \ominus^\#, \otimes^\#,$ and $\oslash^\#$ are the corresponding operations on intervals. Note that these operations always round the interval's lower bound towards $-\infty$ and the upper bound towards $+\infty$ in order to obtain a correct overapproximation.

A (*composite*) *floating point abstract store* is used to associate each floating point variable with its currently known approximation. The store is composed by two parts:

- an *interval abstract store* $\rho^\# : \mathcal{V} \rightarrow \mathbb{I}_a$ associating each variable with its current approximating interval, and
- a *linear form abstract store* $\rho_l^\# : \mathcal{V} \rightarrow \mathbb{L}_a$ associating each variable with its current approximating linear form.

An interval abstract store is represented by a `Box` with floating point boundaries, while a linear form abstract store is a map of the Standard Template Library. The `linearize` method requires both stores as its arguments. Please see the documentation of floating point expression classes for more information.

The linearization of a floating point expression e in the composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$ will be denoted by $\langle e \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$. There are two ways a linearization attempt can fail:

- whenever an interval boundary overflows to $+\infty$ or $-\infty$, and
- when we try to divide by an interval that contains 0.

1.14.2 Use of other abstract domains for floating point analysis

Three of the other abstract domains of the PPL (`BD_Shape`, `Octagonal_Shape`, and `Polyhedron`) provide a few optimized methods to be used in the analysis of floating point computations. They are recognized by the fact that they take interval linear forms and/or an interval abstract stores as their parameters.

Please see the methods' documentation for more information.

1.15 Using the Library

1.15.1 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.

1.15.2 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>`.

1.15.3 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.

1.15.4 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.

1.16 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.
- [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.
- [BFM11] M. Benerecetti, M. Faella, and S. Minopoli. Towards efficient exact synthesis for linear hybrid systems. In *Proceedings of 2nd International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2011)*, volume 54 of *Electronic Proceedings in Theoretical Computer Science*, pages 263-277, Minori, Amalfi Coast, Italy, 2011.
- [BFM13] M. Benerecetti, M. Faella, and S. Minopoli. Automatic synthesis of switching controllers for linear hybrid systems: Safety control. *Theoretical Computer Science*, 493:116-138, 2013.
- [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://bugseng.com/products/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://bugseng.com/products/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.
- [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. Superseded by [BMPZ12a].
- [BMPZ12a] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. The automatic synthesis of linear ranking functions: The complete unabridged version. Report arXiv:cs.PL/1004.0944v2, 2012. Available at <http://arxiv.org/> and <http://bugseng.com/products/ppl/>. Improved version of [BMPZ10].
- [BMPZ12b] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215:47-67, 2012.
- [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, San Antonio, TX, USA, 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 Programming*, 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. Jeannot. *Convex Polyhedra Library*, release 1.1.3c edition, March 2002. Documentation of the "New Polka" library.
- [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.
- [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 Beaulieu, 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

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

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

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

```
Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

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. Secondly, 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:

C++ Language Interface	61
-------------------------------	-----------

5 Namespace Index

5.1 Namespace List

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

Parma_Polyhedra_Library	
The entire library is confined to this namespace	78
Parma_Polyhedra_Library::IO_Operators	
All input/output operators are confined to this namespace	86
std	
The standard C++ namespace	87

6 Hierarchical Index

6.1 Class Hierarchy

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

Parma_Polyhedra_Library::Approximable_Reference< Target >	88
Parma_Polyhedra_Library::Approximable_Reference_Common< Target >	88
Parma_Polyhedra_Library::BD_Shape< T >	90
Parma_Polyhedra_Library::BHRZ03_Certificate	120
Parma_Polyhedra_Library::Binary_Operator< Target >	121
Parma_Polyhedra_Library::Binary_Operator_Common< Target >	121
Parma_Polyhedra_Library::Box< Interval >	122
Parma_Polyhedra_Library::Cast_Operator< Target >	159
Parma_Polyhedra_Library::Cast_Operator_Common< Target >	159
Parma_Polyhedra_Library::Checked_Number< T, Policy >	160
Parma_Polyhedra_Library::Variable::Compare	175
Parma_Polyhedra_Library::BHRZ03_Certificate::Compare	175
Parma_Polyhedra_Library::H79_Certificate::Compare	176
Parma_Polyhedra_Library::Grid_Certificate::Compare	176
Parma_Polyhedra_Library::Concrete_Expression< Target >	176
Parma_Polyhedra_Library::Concrete_Expression_Common< Target >	182
Parma_Polyhedra_Library::Concrete_Expression_Type	182
Parma_Polyhedra_Library::Congruence	184
Parma_Polyhedra_Library::Congruence_System	191
Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 >	196
Parma_Polyhedra_Library::CO_Tree::const_iterator	197
Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator	200
Parma_Polyhedra_Library::Linear_Expression::const_iterator	201
Parma_Polyhedra_Library::Congruence_System::const_iterator	203
Parma_Polyhedra_Library::MIP_Problem::const_iterator	204
Parma_Polyhedra_Library::Grid_Generator_System::const_iterator	205
Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface	206
Parma_Polyhedra_Library::Constraint	210
Parma_Polyhedra_Library::Constraint_System	219

Parma_Polyhedra_Library::Constraint_System_const_iterator	222
Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >	223
Parma_Polyhedra_Library::Determinate< PSET >	224
Parma_Polyhedra_Library::Domain_Product< D1, D2 >	232
Parma_Polyhedra_Library::Implementation::Doubly_Linked_Object	232
Parma_Polyhedra_Library::Implementation::EList< Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< typename Traits::Threshold > >	233
Parma_Polyhedra_Library::Implementation::EList< T >	233
Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< Threshold >	412
Parma_Polyhedra_Library::Implementation::EList_Iterator< T >	235
Parma_Polyhedra_Library::Floating_Point_Constant< Target >	236
Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target >	236
Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >	236
Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >	156
Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, F-P_Format >	208
Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, F-P_Format >	226
Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, F-P_Format >	229
Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >	343
Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, F-P_Format >	385
Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >	499
Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, F-P_Format >	506
Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >	241
Parma_Polyhedra_Library::Generator	242
Parma_Polyhedra_Library::Generator_System	255
Parma_Polyhedra_Library::Generator_System_const_iterator	259

Parma_Polyhedra_Library::GMP_Integer	260
Parma_Polyhedra_Library::Grid	262
Parma_Polyhedra_Library::Grid_Certificate	291
Parma_Polyhedra_Library::Grid_Generator	292
Parma_Polyhedra_Library::Grid_Generator_System	300
Parma_Polyhedra_Library::H79_Certificate	304
Parma_Polyhedra_Library::Implementation::Watchdog::Handler	305
Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag >	306
Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function	307
Parma_Polyhedra_Library::Integer_Constant< Target >	308
Parma_Polyhedra_Library::Integer_Constant_Common< Target >	308
Parma_Polyhedra_Library::Interval< Boundary, Info >	309
Parma_Polyhedra_Library::CO_Tree::iterator	312
Parma_Polyhedra_Library::Linear_Expression	315
Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter	88
Parma_Polyhedra_Library::Linear_Form< C >	326
Parma_Polyhedra_Library::MIP_Problem	335
Parma_Polyhedra_Library::PIP_Solution_Node::No_Constraints	350
Parma_Polyhedra_Library::No_Reduction< D1, D2 >	351
Parma_Polyhedra_Library::Octagonal_Shape< T >	351
Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >	387
Parma_Polyhedra_Library::Implementation::Watchdog::Pending_List< Traits >	413
Parma_Polyhedra_Library::PIP_Problem	415
Parma_Polyhedra_Library::PIP_Tree_Node	429
Parma_Polyhedra_Library::PIP_Decision_Node	414
Parma_Polyhedra_Library::PIP_Solution_Node	426
Parma_Polyhedra_Library::Poly_Con_Relation	455
Parma_Polyhedra_Library::Poly_Gen_Relation	456
Parma_Polyhedra_Library::Polyhedron	458

Parma_Polyhedra_Library::C_Polyhedron	152
Parma_Polyhedra_Library::NNC_Polyhedron	346
Parma_Polyhedra_Library::Powerset< D >	491
Parma_Polyhedra_Library::Powerset< Parma_Polyhedra_Library::Determinate< PSET > >	491
Parma_Polyhedra_Library::Pointset_Powerset< PSET >	433
Parma_Polyhedra_Library::Recycle_Input	497
Parma_Polyhedra_Library::Select_Temp_Boundary_Type< Interval_Boundary_Type >	497
Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 >	498
Parma_Polyhedra_Library::Smash_Reduction< D1, D2 >	499
Parma_Polyhedra_Library::Threshold_Watcher< Traits >	502
Parma_Polyhedra_Library::Throwable	502
Parma_Polyhedra_Library::Implementation::Watchdog::Time	503
Parma_Polyhedra_Library::Unary_Operator< Target >	503
Parma_Polyhedra_Library::Unary_Operator_Common< Target >	504
Parma_Polyhedra_Library::Variable	504
Parma_Polyhedra_Library::Variables_Set	509
Parma_Polyhedra_Library::Watchdog	510

7 Class Index

7.1 Class List

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

Parma_Polyhedra_Library::Approximable_Reference< Target > A concrete expression representing a reference to some approximable	88
Parma_Polyhedra_Library::Approximable_Reference_Common< Target > Base class for references to some approximable	88
Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter Artificial parameters in PIP solution trees	88
Parma_Polyhedra_Library::BD_Shape< T > A bounded difference shape	90
Parma_Polyhedra_Library::BHRZ03_Certificate The convergence certificate for the BHRZ03 widening operator	120

Parma_Polyhedra_Library::Binary_Operator< Target >	
A binary operator applied to two concrete expressions	121
Parma_Polyhedra_Library::Binary_Operator_Common< Target >	
Base class for binary operator applied to two concrete expressions	121
Parma_Polyhedra_Library::Box< Interval >	
A not necessarily closed, iso-oriented hyperrectangle	122
Parma_Polyhedra_Library::C_Polyhedron	
A closed convex polyhedron	152
Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >	
A generic Cast Floating Point Expression	156
Parma_Polyhedra_Library::Cast_Operator< Target >	
A cast operator converting one concrete expression to some type	159
Parma_Polyhedra_Library::Cast_Operator_Common< Target >	
Base class for cast operator concrete expressions	159
Parma_Polyhedra_Library::Checked_Number< T, Policy >	
A wrapper for numeric types implementing a given policy	160
Parma_Polyhedra_Library::Variable::Compare	
Binary predicate defining the total ordering on variables	175
Parma_Polyhedra_Library::BHRZ03_Certificate::Compare	
A total ordering on BHRZ03 certificates	175
Parma_Polyhedra_Library::H79_Certificate::Compare	
A total ordering on H79 certificates	176
Parma_Polyhedra_Library::Grid_Certificate::Compare	
A total ordering on Grid certificates	176
Parma_Polyhedra_Library::Concrete_Expression< Target >	
The base class of all concrete expressions	176
Parma_Polyhedra_Library::Concrete_Expression_Common< Target >	
Base class for all concrete expressions	182
Parma_Polyhedra_Library::Concrete_Expression_Type	
The type of a concrete expression	182
Parma_Polyhedra_Library::Congruence	
A linear congruence	184
Parma_Polyhedra_Library::Congruence_System	
A system of congruences	191
Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 >	
This class provides the reduction method for the Congruences_Product domain	196

Parma_Polyhedra_Library::CO_Tree::const_iterator	
A const iterator on the tree elements, ordered by key	197
Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator	200
Parma_Polyhedra_Library::Linear_Expression::const_iterator	201
Parma_Polyhedra_Library::Congruence_System::const_iterator	
An iterator over a system of congruences	203
Parma_Polyhedra_Library::MIP_Problem::const_iterator	
A read-only iterator on the constraints defining the feasible region	204
Parma_Polyhedra_Library::Grid_Generator_System::const_iterator	
An iterator over a system of grid generators	205
Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface	206
Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format >	
A generic Constant Floating Point Expression	208
Parma_Polyhedra_Library::Constraint	
A linear equality or inequality	210
Parma_Polyhedra_Library::Constraint_System	
A system of constraints	219
Parma_Polyhedra_Library::Constraint_System_const_iterator	
An iterator over a system of constraints	222
Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >	
This class provides the reduction method for the Constraints_Product domain	223
Parma_Polyhedra_Library::Determinate< PSET >	
A wrapper for PPL pointsets, providing them with a <i>determinate constraint system</i> interface, as defined in [Bag98]	224
Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >	
A generic Difference Floating Point Expression	226
Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >	
A generic Division Floating Point Expression	229
Parma_Polyhedra_Library::Domain_Product< D1, D2 >	
This class is temporary and will be removed when template typedefs will be supported in C++	232
Parma_Polyhedra_Library::Implementation::Doubly_Linked_Object	
A (base) class for doubly linked objects	232
Parma_Polyhedra_Library::Implementation::EList< T >	
A simple kind of embedded list (i.e., a doubly linked objects where the links are embedded in the objects themselves)	233

Parma_Polyhedra_Library::Implementation::EList_Iterator< T >	235
A class providing iterators for embedded lists	
Parma_Polyhedra_Library::Floating_Point_Constant< Target >	236
A floating-point constant concrete expression	
Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target >	236
Base class for floating-point constant concrete expression	
Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >	236
Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >	241
An abstract class to be implemented by an external analyzer such as ECLAIR in order to provide to the PPL the necessary information for performing the analysis of floating point computations	
Parma_Polyhedra_Library::Generator	242
A line, ray, point or closure point	
Parma_Polyhedra_Library::Generator_System	255
A system of generators	
Parma_Polyhedra_Library::Generator_System.const_iterator	259
An iterator over a system of generators	
Parma_Polyhedra_Library::GMP_Integer	260
Unbounded integers as provided by the GMP library	
Parma_Polyhedra_Library::Grid	262
A grid	
Parma_Polyhedra_Library::Grid_Certificate	291
The convergence certificate for the Grid widening operator	
Parma_Polyhedra_Library::Grid_Generator	292
A grid line, parameter or grid point	
Parma_Polyhedra_Library::Grid_Generator_System	300
A system of grid generators	
Parma_Polyhedra_Library::H79_Certificate	304
A convergence certificate for the H79 widening operator	
Parma_Polyhedra_Library::Implementation::Watchdog::Handler	305
Abstract base class for handlers of the watchdog events	
Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag >	306
A kind of Handler that installs a flag onto a flag-holder	
Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function	307
A kind of Handler calling a given function	
Parma_Polyhedra_Library::Integer_Constant< Target >	308
An integer constant concrete expression	

Parma_Polyhedra_Library::Integer_Constant_Common< Target >	308
Base class for integer constant concrete expressions	
Parma_Polyhedra_Library::Interval< Boundary, Info >	309
A generic, not necessarily closed, possibly restricted interval	
Parma_Polyhedra_Library::CO_Tree::iterator	312
An iterator on the tree elements, ordered by key	
Parma_Polyhedra_Library::Linear_Expression	315
A linear expression	
Parma_Polyhedra_Library::Linear_Form< C >	326
A linear form with interval coefficients	
Parma_Polyhedra_Library::MIP_Problem	335
A Mixed Integer (linear) Programming problem	
Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >	343
A generic Multiplication Floating Point Expression	
Parma_Polyhedra_Library::NNC_Polyhedron	346
A not necessarily closed convex polyhedron	
Parma_Polyhedra_Library::PIP_Solution_Node::No_Constraints	350
A tag type to select the alternative copy constructor	
Parma_Polyhedra_Library::No_Reduction< D1, D2 >	351
This class provides the reduction method for the Direct_Product domain	
Parma_Polyhedra_Library::Octagonal_Shape< T >	351
An octagonal shape	
Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format >	385
A generic Opposite Floating Point Expression	
Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >	387
The partially reduced product of two abstractions	
Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< Threshold >	412
A class for pending watchdog events with embedded links	
Parma_Polyhedra_Library::Implementation::Watchdog::Pending_List< Traits >	413
An ordered list for recording pending watchdog events	
Parma_Polyhedra_Library::PIP_Decision_Node	414
A tree node representing a decision in the space of solutions	
Parma_Polyhedra_Library::PIP_Problem	415
A Parametric Integer (linear) Programming problem	
Parma_Polyhedra_Library::PIP_Solution_Node	426
A tree node representing part of the space of solutions	

Parma_Polyhedra_Library::PIP_Tree_Node	429
A node of the PIP solution tree	
Parma_Polyhedra_Library::Pointset_Powerset< PSET >	433
The powerset construction instantiated on PPL pointset domains	
Parma_Polyhedra_Library::Poly_Con_Relation	455
The relation between a polyhedron and a constraint	
Parma_Polyhedra_Library::Poly_Gen_Relation	456
The relation between a polyhedron and a generator	
Parma_Polyhedra_Library::Polyhedron	458
The base class for convex polyhedra	
Parma_Polyhedra_Library::Powerset< D >	491
The powerset construction on a base-level domain	
Parma_Polyhedra_Library::Recycle_Input	497
A tag class	
Parma_Polyhedra_Library::Select_Temp_Boundary_Type< Interval_Boundary_Type >	497
Helper class to select the appropriate numerical type to perform boundary computations so as to reduce the chances of overflow without incurring too much overhead	
Parma_Polyhedra_Library::Shape_Preserving_Reduction< D1, D2 >	498
This class provides the reduction method for the Shape_Preserving_Product domain	
Parma_Polyhedra_Library::Smash_Reduction< D1, D2 >	499
This class provides the reduction method for the Smash_Product domain	
Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >	499
A generic Sum Floating Point Expression	
Parma_Polyhedra_Library::Threshold_Watcher< Traits >	502
A class of watchdogs controlling the exceeding of a threshold	
Parma_Polyhedra_Library::Throwable	502
User objects the PPL can throw	
Parma_Polyhedra_Library::Implementation::Watchdog::Time	503
A class for representing and manipulating positive time intervals	
Parma_Polyhedra_Library::Unary_Operator< Target >	503
A unary operator applied to one concrete expression	
Parma_Polyhedra_Library::Unary_Operator_Common< Target >	504
Base class for unary operator applied to one concrete expression	
Parma_Polyhedra_Library::Variable	504
A dimension of the vector space	
Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >	506
A generic Variable Floating Point Expression	

Parma_Polyhedra_Library::Variables_Set An <code>std::set</code> of variables' indexes	509
Parma_Polyhedra_Library::Watchdog A watchdog timer	510

8 Module Documentation

8.1 C++ Language Interface

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

Namespaces

- [Parma_Polyhedra_Library::IO_Operators](#)
All input/output operators are confined to this namespace.
- [std](#)
The standard C++ namespace.

Classes

- 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::Throwable](#)
User objects the PPL can throw.
- struct [Parma_Polyhedra_Library::Recycle_Input](#)
A tag class.
- class [Parma_Polyhedra_Library::Linear_Form< C >](#)
A linear form with interval coefficients.
- class [Parma_Polyhedra_Library::Checked_Number< T, Policy >](#)
A wrapper for numeric types implementing a given policy.
- class [Parma_Polyhedra_Library::Interval< Boundary, Info >](#)
A generic, not necessarily closed, possibly restricted interval.
- class [Parma_Polyhedra_Library::Linear_Expression](#)
A linear expression.
- class [Parma_Polyhedra_Library::Constraint](#)
A linear equality or inequality.
- class [Parma_Polyhedra_Library::Generator](#)
A line, ray, point or closure point.
- class [Parma_Polyhedra_Library::Grid_Generator](#)
A grid line, parameter or grid point.
- class [Parma_Polyhedra_Library::Congruence](#)
A linear congruence.
- class [Parma_Polyhedra_Library::Box< Interval >](#)
A not necessarily closed, iso-oriented hyperrectangle.
- 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::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::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::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::Grid_Generator_System](#)
A system of grid generators.
- class [Parma_Polyhedra_Library::Grid](#)
A grid.
- class [Parma_Polyhedra_Library::BD_Shape< T >](#)
A bounded difference shape.
- class [Parma_Polyhedra_Library::C_Polyhedron](#)
A closed convex polyhedron.
- class [Parma_Polyhedra_Library::Octagonal_Shape< T >](#)
An octagonal shape.
- 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.
- class [Parma_Polyhedra_Library::H79_Certificate](#)
A convergence certificate for the H79 widening operator.
- class [Parma_Polyhedra_Library::Grid_Certificate](#)
*The convergence certificate for the *Grid* widening operator.*
- class [Parma_Polyhedra_Library::NNC_Polyhedron](#)
A not necessarily closed convex polyhedron.
- 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::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Cast Floating Point Expression.
- class `Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Constant Floating Point Expression.
- class `Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Variable Floating Point Expression.
- class `Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Sum Floating Point Expression.
- class `Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Difference Floating Point Expression.
- class `Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Multiplication Floating Point Expression.
- class `Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Division Floating Point Expression.
- class `Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format >`
A generic Opposite Floating Point Expression.
- class `Parma_Polyhedra_Library::GMP_Integer`
Unbounded integers as provided by the GMP library.

Macros

- `#define PPL_VERSION_MAJOR 1`
The major number of the PPL version.
- `#define PPL_VERSION_MINOR 2`
The minor number of the PPL version.
- `#define PPL_VERSION_REVISION 0`
The revision number of the PPL version.
- `#define PPL_VERSION_BETA 4`
The beta number of the PPL version. This is zero for official releases and nonzero for development snapshots.
- `#define PPL_VERSION "1.2pre4"`
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::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::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](#) }

- enum Parma_Polyhedra_Library::Bounded_Integer_Type_Representation { Parma_Polyhedra_Library::UNSIGNED, Parma_Polyhedra_Library::SIGNED_2_COMPLEMENT }
- enum Parma_Polyhedra_Library::Bounded_Integer_Type_Overflow { Parma_Polyhedra_Library::OVERFLOW_WRAPS, Parma_Polyhedra_Library::OVERFLOW_UNDEFINED, Parma_Polyhedra_Library::OVERFLOW_IMPOSSIBLE }
- enum Parma_Polyhedra_Library::Representation { Parma_Polyhedra_Library::DENSE, Parma_Polyhedra_Library::SPARSE }
- enum Parma_Polyhedra_Library::Floating_Point_Format { Parma_Polyhedra_Library::IEEE754_HALF, Parma_Polyhedra_Library::IEEE754_SINGLE, Parma_Polyhedra_Library::IEEE754_DOUBLE, Parma_Polyhedra_Library::IEEE754_QUAD, Parma_Polyhedra_Library::INTEL_DOUBLE_EXTENDED, Parma_Polyhedra_Library::IBM_SINGLE, Parma_Polyhedra_Library::IBM_DOUBLE }
- 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.

Functions Inspecting and/or Combining Result Values

- Result Parma_Polyhedra_Library::operator& (Result x, Result y)
- Result Parma_Polyhedra_Library::operator| (Result x, Result y)
- Result Parma_Polyhedra_Library::operator- (Result x, Result y)
- Result_Class Parma_Polyhedra_Library::result_class (Result r)
- Result_Relation Parma_Polyhedra_Library::result_relation (Result r)
- Result Parma_Polyhedra_Library::result_relation_class (Result r)

Functions Inspecting and/or Combining Rounding_Dir Values

- Rounding_Dir Parma_Polyhedra_Library::operator& (Rounding_Dir x, Rounding_Dir y)
- Rounding_Dir Parma_Polyhedra_Library::operator| (Rounding_Dir x, Rounding_Dir y)
- Rounding_Dir Parma_Polyhedra_Library::inverse (Rounding_Dir dir)
- Rounding_Dir Parma_Polyhedra_Library::round_dir (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_down (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_up (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_ignore (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_not_needed (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_not_requested (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_direct (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_inverse (Rounding_Dir dir)
- bool Parma_Polyhedra_Library::round_strict_relation (Rounding_Dir dir)
- fpu_rounding_direction_type Parma_Polyhedra_Library::round_fpu_dir (Rounding_Dir dir)

Functions for the Synthesis of Linear Rankings

- `template<typename PSET >`
`bool Parma_Polyhedra_Library::termination_test_MS (const PSET &pset)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::termination_test_MS_2 (const PSET &pset_before, const PSET &pset_after)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::one_affine_ranking_function_MS (const PSET &pset, Generator &mu)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::one_affine_ranking_function_MS_2 (const PSET &pset_before, const PSET &pset_after, Generator &mu)`
- `template<typename PSET >`
`void Parma_Polyhedra_Library::all_affine_ranking_functions_MS (const PSET &pset, C_Polyhedron &mu_space)`
- `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)`
- `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)`
- `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)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::termination_test_PR (const PSET &pset)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::termination_test_PR_2 (const PSET &pset_before, const PSET &pset_after)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::one_affine_ranking_function_PR (const PSET &pset, Generator &mu)`
- `template<typename PSET >`
`bool Parma_Polyhedra_Library::one_affine_ranking_function_PR_2 (const PSET &pset_before, const PSET &pset_after, Generator &mu)`
- `template<typename PSET >`
`void Parma_Polyhedra_Library::all_affine_ranking_functions_PR (const PSET &pset, NNC_Polyhedron &mu_space)`
- `template<typename PSET >`
`void Parma_Polyhedra_Library::all_affine_ranking_functions_PR_2 (const PSET &pset_before, const PSET &pset_after, NNC_Polyhedron &mu_space)`

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 Macro Definition Documentation

#define PPL_VERSION_MAJOR 1 The major number of the PPL version.

#define PPL_VERSION_MINOR 2 The minor number of the PPL version.

#define PPL_VERSION_REVISION 0 The revision number of the PPL version.

#define PPL_VERSION "1.2pre4" 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

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.

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

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.

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 between the exact result and the computed one.

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.

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.

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.

enum Parma_Polyhedra_Library::Optimization_Mode Possible optimization modes.

Enumerator

MINIMIZATION Minimization is requested.

MAXIMIZATION Maximization is requested.

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.

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.

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.

enum Parma_Polyhedra_Library::Representation \ Possible representations of coefficient sequences (i.e. linear expressions and more complex objects containing linear expressions, e.g. Constraints, Generators, etc.).

Enumerator

DENSE Dense representation: the coefficient sequence is represented as a vector of coefficients, including the zero coefficients. If there are only a few nonzero coefficients, this representation is faster and also uses a bit less memory.

SPARSE Sparse representation: only the nonzero coefficient are stored. If there are many nonzero coefficients, this improves memory consumption and run time (both because there is less data to process in $O(n)$ operations and because finding zeroes/nonzeroes is much faster since zeroes are not stored at all, so any stored coefficient is nonzero).

enum Parma_Polyhedra_Library::Floating_Point_Format \ Floating point formats known to the library.

The parameters of each format are defined by a specific struct in file Float_defs.hh. See the section on [Analysis of floating point computations](#) for more information.

Enumerator

IEEE754_HALF IEEE 754 half precision, 16 bits (5 exponent, 10 mantissa).

IEEE754_SINGLE IEEE 754 single precision, 32 bits (8 exponent, 23 mantissa).

IEEE754_DOUBLE IEEE 754 double precision, 64 bits (11 exponent, 52 mantissa).

IEEE754_QUAD IEEE 754 quad precision, 128 bits (15 exponent, 112 mantissa).

INTEL_DOUBLE_EXTENDED Intel double extended precision, 80 bits (15 exponent, 64 mantissa)

IBM_SINGLE IBM single precision, 32 bits (7 exponent, 24 mantissa).

IBM_DOUBLE IBM double precision, 64 bits (7 exponent, 56 mantissa).

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.

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 Function Documentation

Result Parma_Polyhedra_Library::operator& (Result x, Result y) [inline]

Result Parma_Polyhedra_Library::operator| (Result x, Result y) [inline]

Result Parma_Polyhedra_Library::operator- (Result x, Result y) [inline]

Result_Class Parma_Polyhedra_Library::result_class (Result *r*) [inline] \ Extracts the value class part of *r* (representable number, unrepresentable minus/plus infinity or nan).

Result_Relation Parma_Polyhedra_Library::result_relation (Result *r*) [inline] \ Extracts the relation part of *r*.

Result Parma_Polyhedra_Library::result_relation_class (Result *r*) [inline]

Rounding_Dir Parma_Polyhedra_Library::operator& (Rounding_Dir *x*, Rounding_Dir *y*) [inline]

Rounding_Dir Parma_Polyhedra_Library::operator| (Rounding_Dir *x*, Rounding_Dir *y*) [inline]

Rounding_Dir Parma_Polyhedra_Library::inverse (Rounding_Dir *dir*) [inline] \ Returns the inverse rounding mode of *dir*, ROUND_IGNORE being the inverse of itself.

Rounding_Dir Parma_Polyhedra_Library::round_dir (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_down (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_up (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_ignore (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_not_needed (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_not_requested (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_direct (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_inverse (Rounding_Dir *dir*) [inline]

bool Parma_Polyhedra_Library::round_strict_relation (Rounding_Dir *dir*) [inline]

fpu_rounding_direction_type Parma_Polyhedra_Library::round_fpu_dir (Rounding_Dir *dir*) [inline]

template<typename PSET> bool Parma_Polyhedra_Library::termination_test_MS (const PSET & *pset*) \ Termination test using an improvement of the method by Mesnard and Serebrenik [\[BMPZ10\]](#).
Template Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset</i>	<p>A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:</p> <ul style="list-style-type: none"> • x'_1, \dots, x'_n go onto space dimensions $0, \dots, n - 1$, • x_1, \dots, x_n go onto space dimensions $n, \dots, 2n - 1$, <p>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.</p>
-------------	--

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.

template<typename PSET > bool Parma_Polyhedra_Library::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 Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset_before</i>	A pointset approximating the values of loop-relevant variables <i>before</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • x_1, \dots, x_n go onto space dimensions $0, \dots, n - 1$.
<i>pset_after</i>	A pointset approximating the values of loop-relevant variables <i>after</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • 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.

template<typename PSET > bool Parma_Polyhedra_Library::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 Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset</i>	<p>A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:</p> <ul style="list-style-type: none"> • x'_1, \dots, x'_n go onto space dimensions $0, \dots, n - 1$, • x_1, \dots, x_n go onto space dimensions $n, \dots, 2n - 1$, <p>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.</p>
<i>mu</i>	<p>When <code>true</code> 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 $\mu_0, \mu_1, \dots, \mu_n$ are the coefficients of <code>mu</code> corresponding to the space dimensions $n, 0, \dots, n - 1$, respectively.</p>

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.

template<typename PSET > bool Parma_Polyhedra_Library::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 Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset_before</i>	<p>A pointset approximating the values of loop-relevant variables <i>before</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:</p> <ul style="list-style-type: none"> • x_1, \dots, x_n go onto space dimensions $0, \dots, n - 1$.
<i>pset_after</i>	<p>A pointset approximating the values of loop-relevant variables <i>after</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows:</p> <ul style="list-style-type: none"> • 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`.

Parameters

<i>mu</i>	When <code>true</code> 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 $\mu_0, \mu_1, \dots, \mu_n$ are the coefficients of <code>mu</code> corresponding to the space dimensions $n, 0, \dots, n-1$, 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.

template<typename PSET > void Parma_Polyhedra_Library::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 Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset</i>	<p>A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows:</p> <ul style="list-style-type: none"> x'_1, \dots, x'_n go onto space dimensions $0, \dots, n-1$, x_1, \dots, x_n go onto space dimensions $n, \dots, 2n-1$, <p>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.</p>
<i>mu_space</i>	<p>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 <code>pset</code>. These ranking functions are of the form $\mu_0 + \sum_{i=1}^n \mu_i x_i$ where $\mu_0, \mu_1, \dots, \mu_n$ identify any point of the <code>mu_space</code> polyhedron. The variables $\mu_0, \mu_1, \dots, \mu_n$ correspond to the space dimensions of <code>mu_space</code> $n, 0, \dots, n-1$, respectively. When <code>mu_space</code> is empty, it means that the test is inconclusive. However, if <code>pset</code> <i>precisely</i> characterizes the effect of the loop body onto the loop-relevant program variables, then <code>mu_space</code> is empty <i>if and only if</i> the loop does <i>not</i> terminate.</p>

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

Template Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset_before</i>	A pointset approximating the values of loop-relevant variables <i>before</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • x_1, \dots, x_n go onto space dimensions $0, \dots, n - 1$.
<i>pset_after</i>	A pointset approximating the values of loop-relevant variables <i>after</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • 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*.

Parameters

<i>mu_space</i>	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 <i>pset</i> . These ranking functions are of the form $\mu_0 + \sum_{i=1}^n \mu_i x_i$ where $\mu_0, \mu_1, \dots, \mu_n$ identify any point of the <i>mu_space</i> polyhedron. The variables $\mu_0, \mu_1, \dots, \mu_n$ correspond to the space dimensions of <i>mu_space</i> $n, 0, \dots, n - 1$, respectively. When <i>mu_space</i> is empty, it means that the test is inconclusive. However, if <i>pset_before</i> and <i>pset_after</i> <i>precisely</i> characterize the effect of the loop body onto the loop-relevant program variables, then <i>mu_space</i> is empty <i>if and only if</i> the loop does <i>not</i> terminate.
-----------------	--

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

Template Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset</i>	A pointset approximating the behavior of a loop whose termination is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • x'_1, \dots, x'_n go onto space dimensions $0, \dots, n - 1$, • x_1, \dots, x_n go onto space dimensions $n, \dots, 2n - 1$, <p>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.</p>
-------------	---

<i>decreasing_mu_space</i>	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 <i>pset</i> .
<i>bounded_mu_space</i>	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 <i>pset</i> .

These quasi-ranking functions are of the form $\mu_0 + \sum_{i=1}^n \mu_i x_i$ where $\mu_0, \mu_1, \dots, \mu_n$ identify any point of the *decreasing_mu_space* and *bounded_mu_space* polyhedrons. The variables $\mu_0, \mu_1, \dots, \mu_n$ correspond to the space dimensions $n, 0, \dots, n - 1$, 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.

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

Template Parameters

<i>PSET</i>	Any pointset supported by the PPL that provides the <code>minimized_constraints()</code> method.
-------------	--

Parameters

<i>pset_before</i>	A pointset approximating the values of loop-relevant variables <i>before</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • x_1, \dots, x_n go onto space dimensions $0, \dots, n - 1$.
<i>pset_after</i>	A pointset approximating the values of loop-relevant variables <i>after</i> the update performed in the loop body that is being analyzed. The variables indices are allocated as follows: <ul style="list-style-type: none"> • 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*.

Parameters

<i>decreasing_mu_space</i>	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 <i>pset</i> .
<i>bounded_mu_space</i>	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 <i>pset</i> .

These ranking functions are of the form $\mu_0 + \sum_{i=1}^n \mu_i x_i$ where $\mu_0, \mu_1, \dots, \mu_n$ identify any point of the *decreasing_mu_space* and *bounded_mu_space* polyhedrons. The variables $\mu_0, \mu_1, \dots, \mu_n$ correspond to the space dimensions $n, 0, \dots, n - 1$, 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.

template<typename PSET > bool Parma_Polyhedra_Library::termination_test_PR (const PSET & pset) \ Like [termination_test_MS\(\)](#) but using the method by Podelski and Rybalchenko [BMPZ10].

template<typename PSET > bool Parma_Polyhedra_Library::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 Parma_Polyhedra_Library::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 Parma_Polyhedra_Library::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 Parma_Polyhedra_Library::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 Parma_Polyhedra_Library::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].

8.1.6 Variable Documentation

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

- [IO_Operators](#)

All input/output operators are confined to this namespace.

Classes

- class [Checked_Number](#)

A wrapper for numeric types implementing a given policy.

- class [Concrete_Expression](#)

The base class of all concrete expressions.

- class [Binary_Operator](#)

A binary operator applied to two concrete expressions.

- class [Unary_Operator](#)

A unary operator applied to one concrete expression.

- class [Cast_Operator](#)

A cast operator converting one concrete expression to some type.

- class [Integer_Constant](#)

An integer constant concrete expression.

- class [Floating_Point_Constant](#)

A floating-point constant concrete expression.

- class [Approximable_Reference](#)

A concrete expression representing a reference to some approximable.

- class [Linear_Form](#)

A linear form with interval coefficients.

- class [FP_Oracle](#)

An abstract class to be implemented by an external analyzer such as ECLAIR in order to provide to the PPL the necessary information for performing the analysis of floating point computations.

- class [Variable](#)

A dimension of the vector space.

- class [Throwable](#)

User objects the PPL can throw.

- struct [Recycle_Input](#)

A tag class.

- class [Box](#)

A not necessarily closed, iso-oriented hyperrectangle.

- class [Interval](#)

A generic, not necessarily closed, possibly restricted interval.

- struct [Select_Temp_Boundary_Type](#)

Helper class to select the appropriate numerical type to perform boundary computations so as to reduce the chances of overflow without incurring too much overhead.

- class [Variables_Set](#)

An `std::set` of variables' indexes.

- class [Octagonal_Shape](#)

An octagonal shape.

- class [BD_Shape](#)

A bounded difference shape.

- class [Linear_Expression](#)

- A linear expression.*

 - class [Constraint](#)
- A linear equality or inequality.*

 - class [Generator](#)
- A line, ray, point or closure point.*

 - class [Grid_Generator](#)
- A grid line, parameter or grid point.*

 - class [Congruence](#)
- A linear congruence.*

 - 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 [Constraint_System](#)
- A system of constraints.*

 - class [Constraint_System_const_iterator](#)
- An iterator over a system of constraints.*

 - class [Congruence_System](#)
- A system of congruences.*

 - class [Poly_Con_Relation](#)
- The relation between a polyhedron and a constraint.*

 - class [Generator_System](#)
- A system of generators.*

 - class [Generator_System_const_iterator](#)
- An iterator over a system of generators.*

 - 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 [Floating_Point_Expression](#)
- A system of grid generators.*

 - class [Grid_Generator_System](#)
- A grid.*

 - class [Grid](#)
- A closed convex polyhedron.*

 - class [C_Polyhedron](#)
- A concrete expression type.*

 - class [Concrete_Expression_Type](#)

- The type of a concrete expression.*

 - class [Concrete_Expression_Common](#)

Base class for all concrete expressions.
 - class [Binary_Operator_Common](#)

Base class for binary operator applied to two concrete expressions.
 - class [Unary_Operator_Common](#)

Base class for unary operator applied to one concrete expression.
 - class [Cast_Operator_Common](#)

Base class for cast operator concrete expressions.
 - class [Integer_Constant_Common](#)

Base class for integer constant concrete expressions.
 - class [Floating_Point_Constant_Common](#)

Base class for floating-point constant concrete expression.
 - class [Approximable_Reference_Common](#)

Base class for references to some approximable.
 - 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 [Grid_Certificate](#)

The convergence certificate for the [Grid](#) widening operator.
 - class [NNC_Polyhedron](#)

A not necessarily closed convex polyhedron.
 - class [Pointset_Powerset](#)

The powerset construction instantiated on PPL pointset domains.
 - 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 [Cast_Floating_Point_Expression](#)

A generic Cast Floating Point Expression.
 - class [Constant_Floating_Point_Expression](#)

A generic Constant Floating Point Expression.
 - class [Variable_Floating_Point_Expression](#)

*A generic *Variable* Floating Point Expression.*
 - class [Sum_Floating_Point_Expression](#)

- *A generic Sum Floating Point Expression.*
- class [Difference_Floating_Point_Expression](#)
A generic Difference Floating Point Expression.
- class [Multiplication_Floating_Point_Expression](#)
A generic Multiplication Floating Point Expression.
- class [Division_Floating_Point_Expression](#)
A generic Division Floating Point Expression.
- class [Opposite_Floating_Point_Expression](#)
A generic Opposite Floating Point Expression.
- class [Watchdog](#)
A watchdog timer.
- class [Threshold_Watcher](#)
A class of watchdogs controlling the exceeding of a threshold.
- 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 int [Concrete_Expression_Kind](#)
Encodes the kind of concrete expression.
- typedef int [Concrete_Expression_BOP](#)
Encodes a binary operator of concrete expressions.
- typedef int [Concrete_Expression_UOP](#)
Encodes a unary operator of concrete expressions.
- 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 [Rounding_Dir](#) { [ROUND_DOWN](#), [ROUND_UP](#), [ROUND_IGNORE](#), [ROUND_NOT_NEEDED](#), [ROUND_STRICT_RELATION](#) }

- enum `Degenerate_Element` { `UNIVERSE`, `EMPTY` }
Rounding directions for arithmetic computations.
- enum `Relation_Symbol` { `EQUAL`, `LESS_THAN`, `LESS_OR_EQUAL`, `GREATER_THAN`, `GREATER_OR_EQUAL`, `NOT_EQUAL` }
Kinds of degenerate abstract elements.
- enum `Complexity_Class` { `POLYNOMIAL_COMPLEXITY`, `SIMPLEX_COMPLEXITY`, `ANY_C_COMPLEXITY` }
Relation symbols.
- enum `Optimization_Mode` { `MINIMIZATION`, `MAXIMIZATION` }
Complexity pseudo-classes.
- enum `Bounded_Integer_Type_Width` { `BITS_8`, `BITS_16`, `BITS_32`, `BITS_64`, `BITS_128` }
Possible optimization modes.
- enum `Bounded_Integer_Type_Representation` { `UNSIGNED`, `SIGNED_2_COMPLEMENT` }
- enum `Bounded_Integer_Type_Overflow` { `OVERFLOW_WRAPS`, `OVERFLOW_UNDEFINED`, `OVERFLOW_IMPOSSIBLE` }
- enum `Representation` { `DENSE`, `SPARSE` }
- enum `Floating_Point_Format` { `IEEE754_HALF`, `IEEE754_SINGLE`, `IEEE754_DOUBLE`, `IEEE754_QUAD`, `INTEL_DOUBLE_EXTENDED`, `IBM_SINGLE`, `IBM_DOUBLE` }
- 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.

Functions

- `dimension_type not_a_dimension ()`
Returns a value that does not designate a valid dimension.
- `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.
- `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 initialize ()`
Initializes the library.
- `void finalize ()`
Finalizes the library.
- `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.
- `dimension_type max_space_dimension ()`

Returns the maximum space dimension this library can handle.

Library Version Control 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.

Functions Inspecting and/or Combining Result Values

- [Result operator&](#) (Result x, Result y)
- [Result operator|](#) (Result x, Result y)
- [Result operator-](#) (Result x, Result y)
- [Result_Class result_class](#) (Result r)
- [Result_Relation result_relation](#) (Result r)
- [Result result_relation_class](#) (Result r)

Functions Controlling Floating Point Unit

- 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.

Functions Inspecting and/or Combining Rounding_Dir Values

- [Rounding_Dir operator&](#) (Rounding_Dir x, Rounding_Dir y)
- [Rounding_Dir operator|](#) (Rounding_Dir x, Rounding_Dir y)
- [Rounding_Dir inverse](#) (Rounding_Dir dir)
- [Rounding_Dir round_dir](#) (Rounding_Dir dir)
- bool [round_down](#) (Rounding_Dir dir)
- bool [round_up](#) (Rounding_Dir dir)

- bool `round_ignore` (`Rounding_Dir` dir)
- bool `round_not_needed` (`Rounding_Dir` dir)
- bool `round_not_requested` (`Rounding_Dir` dir)
- bool `round_direct` (`Rounding_Dir` dir)
- bool `round_inverse` (`Rounding_Dir` dir)
- bool `round_strict_relation` (`Rounding_Dir` dir)
- `fpu_rounding_direction_type` `round_fpu_dir` (`Rounding_Dir` dir)

Functions for the Synthesis of Linear Rankings

- `template<typename PSET >`
bool `termination_test_MS` (const PSET &pset)
- `template<typename PSET >`
bool `termination_test_MS_2` (const PSET &pset_before, const PSET &pset_after)
- `template<typename PSET >`
bool `one_affine_ranking_function_MS` (const PSET &pset, `Generator` &mu)
- `template<typename PSET >`
bool `one_affine_ranking_function_MS_2` (const PSET &pset_before, const PSET &pset_after, `Generator` &mu)
- `template<typename PSET >`
void `all_affine_ranking_functions_MS` (const PSET &pset, `C_Polyhedron` &mu_space)
- `template<typename PSET >`
void `all_affine_ranking_functions_MS_2` (const PSET &pset_before, const PSET &pset_after, `C_Polyhedron` &mu_space)
- `template<typename PSET >`
void `all_affine_quasi_ranking_functions_MS` (const PSET &pset, `C_Polyhedron` &decreasing_mu_space, `C_Polyhedron` &bounded_mu_space)
- `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)
- `template<typename PSET >`
bool `termination_test_PR` (const PSET &pset)
- `template<typename PSET >`
bool `termination_test_PR_2` (const PSET &pset_before, const PSET &pset_after)
- `template<typename PSET >`
bool `one_affine_ranking_function_PR` (const PSET &pset, `Generator` &mu)
- `template<typename PSET >`
bool `one_affine_ranking_function_PR_2` (const PSET &pset_before, const PSET &pset_after, `Generator` &mu)
- `template<typename PSET >`
void `all_affine_ranking_functions_PR` (const PSET &pset, `NNC_Polyhedron` &mu_space)
- `template<typename PSET >`
void `all_affine_ranking_functions_PR_2` (const PSET &pset_before, const PSET &pset_after, `NNC_Polyhedron` &mu_space)

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 Typedef Documentation

typedef int Parma_Polyhedra_Library::Concrete_Expression_Kind Encodes the kind of concrete expression.

The values should be defined by the particular instance and uniquely identify one of: [Binary_Operator](#), [Unary_Operator](#), [Cast_Operator](#), [Integer_Constant](#), [Floating_Point_Constant](#), or [Approximable_Reference](#). For example, the [Binary_Operator](#) kind integer constant should be defined by an instance as the member [Binary_Operator](#)<T>::KIND.

typedef int Parma_Polyhedra_Library::Concrete_Expression_BOP Encodes a binary operator of concrete expressions.

The values should be uniquely defined by the particular instance and named: ADD, SUB, MUL, DIV, REM, BAND, BOR, BXOR, LSHIFT, RSHIFT.

typedef int Parma_Polyhedra_Library::Concrete_Expression_UOP Encodes a unary operator of concrete expressions.

The values should be uniquely defined by the particular instance and named: PLUS, MINUS, BNOT.

9.1.3 Enumeration Type Documentation

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.

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.

9.1.4 Function Documentation

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.

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.

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

<i>std::invalid_argument</i>	Thrown if p is greater than <code>INT_MAX</code> .
------------------------------	--

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.

void Parma_Polyhedra_Library::restore_pre_PPL_rounding () [inline] Sets the FPU rounding mode as it was before initialization of the PPL.

This is important if the application uses floating-point computations outside the PPL. It is crucial when the application uses functions from a mathematical library that are not guaranteed to work correctly under all rounding modes.

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.

9.2 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

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

<i>src_string</i>	The source string holding the lines to wrap.
<i>indent_depth</i>	The indentation depth.
<i>preferred_first_line_length</i>	The preferred length for the first line of text.
<i>preferred_line_length</i>	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 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::Approximable_Reference< Target > Class Template Reference

A concrete expression representing a reference to some approximable.

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

10.1.1 Detailed Description

```
template<typename Target>class Parma_Polyhedra_Library::Approximable_Reference< Target >
```

A concrete expression representing a reference to some approximable.

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

- `ppl.hh`

10.2 Parma_Polyhedra_Library::Approximable_Reference_Common< Target > Class Template Reference

Base class for references to some approximable.

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

10.2.1 Detailed Description

`template<typename Target>class Parma_Polyhedra_Library::Approximable_Reference_Common<Target>`

Base class for references to some approximable.

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

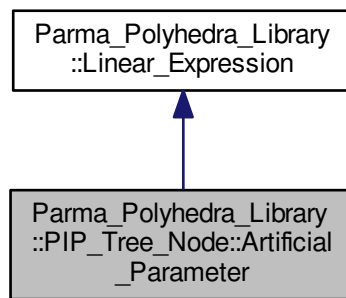
- ppl.hh

10.3 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial Parameter Class Reference

Artificial parameters in PIP solution trees.

`#include <ppl.hh>`

Inheritance diagram for Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter:



Public Member Functions

- [Artificial_Parameter](#) ()
Default constructor: builds a zero artificial parameter.
- [Artificial_Parameter](#) (const [Linear_Expression](#) &expr, Coefficient_traits::const_reference d)
Constructor.
- [Artificial_Parameter](#) (const [Artificial_Parameter](#) &y)
Copy constructor.
- Coefficient_traits::const_reference [denominator](#) () const
Returns the normalized (i.e., positive) denominator.
- void [m_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.)

- void `swap` (PIP_Tree_Node::Artificial_Parameter &x, PIP_Tree_Node::Artificial_Parameter &y)
Swaps x with y.
- std::ostream & `operator<<` (std::ostream &os, const PIP_Tree_Node::Artificial_Parameter &x)
Output operator.
- void `swap` (PIP_Tree_Node::Artificial_Parameter &x, PIP_Tree_Node::Artificial_Parameter &y)

Additional Inherited Members

10.3.1 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.

10.3.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter::Artificial_Parameter (const Linear_Expression &expr, Coefficient_traits::const_reference d) Constructor.

Builds artificial parameter $\frac{\text{expr}}{d}$.

Parameters

<i>expr</i>	The expression that, after normalization, will form the numerator of the artificial parameter.
<i>d</i>	The integer constant that, after normalization, will form the denominator of the artificial parameter.

Exceptions

<i>std::invalid_argument</i>	Thrown if d is zero.
------------------------------	----------------------

Normalization will ensure that the denominator is positive.

10.3.3 Member Function Documentation

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.

10.3.4 Friends And Related Function Documentation

void swap (PIP_Tree_Node::Artificial_Parameter & x, PIP_Tree_Node::Artificial_Parameter & y)
[related] Swaps x with y.

std::ostream & operator<< (std::ostream & os, const PIP_Tree_Node::Artificial_Parameter & x)
[related] Output operator.

void swap (PIP_Tree_Node::Artificial_Parameter & x, PIP_Tree_Node::Artificial_Parameter & y)
[related] The documentation for this class was generated from the following file:

- ppl.hh

10.4 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 **m_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 $*this$.*

 - template<typename Interval_Info >
void `refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< T, Interval_Info > > &left, const `Linear_Form`< `Interval`< T, Interval_Info > > &right)
*Refines the system of `BD_Shape` constraints defining $*this$ using the constraint expressed by $left \leq right$.*
 - template<typename Interval_Info >
void `generalized_refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< T, Interval_Info > > &left, const `Linear_Form`< `Interval`< T, Interval_Info > > &right, `Relation_Symbol` relsym)
*Refines the system of `BD_Shape` constraints defining $*this$ using the constraint expressed by $left \bowtie right$, where \bowtie is the relation symbol specified by `relsym`.*
 - template<typename U >
void `export_interval_constraints` (U &dest) const
*Applies to `dest` the interval constraints embedded in $*this$.*
 - 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`.*
 - template<typename Interval_Info >
void `affine_form_image` (`Variable` var, const `Linear_Form`< `Interval`< T, Interval_Info > > &lf)
*Assigns to $*this$ the `affine form image` of $*this$ under the function mapping variable `var` into the affine expression(s) specified by `lf`.*
 - 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 $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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 $\text{lhs}' \bowtie \text{rhs}$, 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 $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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 $\text{lhs}' \bowtie \text{rhs}$, 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 $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.*
- 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 $\text{var}' \leq \frac{\text{lb_expr}}{\text{denominator}} \leq \frac{\text{ub_expr}}{\text{denominator}}$.*
- 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 *cs, p=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*.
- 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 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.

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 >
void [swap](#) ([BD_Shape](#)< T > &x, [BD_Shape](#)< T > &y)
Swaps *x* with *y*.

- `template<typename T >`
`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.
- `template<typename T >`
`bool operator!= (const BD_Shape< T > &x, const BD_Shape< T > &y)`
Returns true if and only if x and y are not 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)`
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)`
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)`
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 >`
`bool operator== (const BD_Shape< T > &x, const BD_Shape< T > &y)`
- `template<typename T >`
`bool operator!= (const BD_Shape< T > &x, const BD_Shape< T > &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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1,
Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1,
Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1,
Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `template<typename T >`
`void swap (BD_Shape< T > &x, BD_Shape< T > &y)`
- `template<typename T >`
`std::ostream & operator<< (std::ostream &s, const BD_Shape< T > &bds)`

10.4.1 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 lose 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);
```

10.4.2 Constructor & Destructor Documentation

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

<i>num_-dimensions</i>	The number of dimensions of the vector space enclosing the BDS;
<i>kind</i>	Specifies whether the universe or the empty BDS has to be built.

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.

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.

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

<i>cs</i>	A system of BD constraints.
-----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>cs</i> contains a constraint which is not optimally supported by the BD shape domain.
------------------------------	--

template<typename T> Parma_Polyhedra_Library::BD_Shape< T>::BD_Shape (const Congruence_System & cgs) [explicit] Builds a BDS from a system of congruences.

The BDS inherits the space dimension of *cgs*

Parameters

<i>cgs</i>	A system of congruences.
------------	--------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>cgs</i> contains congruences which are not optimally supported by the BD shape domain.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

template<typename T> Parma_Polyhedra_Library::BD_Shape< T>::BD_Shape (const Polyhedron & ph, Complexity_Class *complexity* = ANY_COMPLEXITY) [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`.

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

<i>box</i>	The box representing the BDS to be built.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>box</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>grid</i>	The grid used to build the BDS.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>grid</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>os</i>	The octagonal shape used to build the BDS.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>os</code> exceeds the maximum allowed space dimension.
--------------------------	---

10.4.3 Member Function Documentation

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value;
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <code>expr</code> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if and only if the infimum is also the minimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if and only if the infimum is also the minimum value;
<i>g</i>	When minimization succeeds, will be assigned a point or closure point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

Parameters

<i>expr</i>	The linear expression for which the frequency is needed;
<i>freq_n</i>	If <code>true</code> is returned, the value is set to 0; Present for interface compatibility with class Grid , where the frequency can have a non-zero value;
<i>freq_d</i>	If <code>true</code> is returned, the value is set to 1;
<i>val_n</i>	The numerator of <i>val</i> ;
<i>val_d</i>	The denominator of <i>val</i> ;

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::contains (const BD_Shape< T > & *y*) const Returns `true` if and only if **this* contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::is_disjoint_from (const BD_Shape< T > & y) const Returns `true` if and only if **this* and *y* are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

**template<typename T> Poly_Con_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation-
_with (const Constraint & c) const** Returns the relations holding between **this* and the constraint *c*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

**template<typename T> Poly_Con_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation-
_with (const Congruence & cg) const** Returns the relations holding between **this* and the congruence *cg*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

**template<typename T> Poly_Gen_Relation Parma_Polyhedra_Library::BD_Shape< T >::relation-
_with (const Generator & g) const** Returns the relations holding between **this* and the generator *g*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible.
------------------------------	---

template<typename T> bool Parma_Polyhedra_Library::BD_Shape< T >::constrains (Variable *var*) const Returns `true` if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_constraint (const Constraint & c) Adds a copy of constraint *c* to the system of bounded differences defining **this*.

Parameters

<i>c</i>	The constraint to be added.
----------	-----------------------------

Exceptions

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

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::add_congruence (const Congruence & cg) Adds a copy of congruence *cg* to the system of congruences of **this*.

Parameters

<i>cg</i>	The congruence to be added.
-----------	-----------------------------

Exceptions

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

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

<i>cs</i>	The constraints that will be added.
-----------	-------------------------------------

Exceptions

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

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

<i>cs</i>	The constraint system to be added to <i>*this</i> . The constraints in <i>cs</i> may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible, or <i>cs</i> 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.

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

<i>cgs</i>	Contains the congruences that will be added to the system of constraints of <i>*this</i> .
------------	--

Exceptions

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

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

<i>cgs</i>	Contains the congruences that will be added to the system of constraints of <i>*this</i> . Its elements may be recycled.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or <i>cgs</i> 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.

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

<i>c</i>	The constraint. If it is not a bounded difference, it will be ignored.
----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

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

<i>cg</i>	The congruence. If it is not a bounded difference equality, it will be ignored.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

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

<i>cs</i>	The constraint system to be used. Constraints that are not bounded differences are ignored.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::refine_with_congruences (const Congruence_System & *cgs*) Uses a copy of the congruences in *cgs* to refine the system of bounded differences defining **this*.

Parameters

<i>cgs</i>	The congruence system to be used. Congruences that are not bounded difference equalities are ignored.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::BD_Shape< T >::refine_with_linear_form_inequality (const Linear_Form< Interval< T, Interval_Info >> & *left*, const Linear_Form< Interval< T, Interval_Info >> & *right*) Refines the system of [BD_Shape](#) constraints defining **this* using the constraint expressed by $\text{left} \leq \text{right}$.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is at the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is at the right of the comparison operator. All of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>left</i> (or <i>right</i>) is dimension-incompatible with <i>*this</i> .
------------------------------	--

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by *left* and *right* respectively.

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::BD_Shape< T >::generalized_refine_with_linear_form_inequality (const Linear_Form< Interval< T, Interval_Info >> & *left*, const Linear_Form< Interval< T, Interval_Info >> & *right*, Relation_Symbol *relsym*) [inline] Refines the system of [BD_Shape](#) constraints defining **this* using the constraint expressed by $\text{left} \bowtie \text{right}$, where \bowtie is the relation symbol specified by *relsym*.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is at the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is at the right of the comparison operator. All of its coefficients MUST be bounded.
<i>relsym</i>	The relation symbol.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>left</code> (or <code>right</code>) is dimension-incompatible with <code>*this</code> .
<i>std::runtime_error</i>	Thrown if <code>relsym</code> is not a valid relation symbol.

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by `left` and `right` respectively.

template<typename T > template<typename U > void Parma_Polyhedra_Library::BD_Shape< T >::export_interval_constraints (U & *dest*) const Applies to `dest` the interval constraints embedded in `*this`.

Parameters

<i>dest</i>	The object to which the constraints will be added.
-------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> .
------------------------------	---

The template type parameter `U` must provide the following methods.

```
dimension_type space_dimension() const
```

returns the space dimension of the object.

```
void set_empty()
```

sets the object to an empty object.

```
bool restrict_lower(dimension_type dim, const T& lb)
```

restricts the object by applying the lower bound `lb` to the space dimension `dim` and returns `false` if and only if the object becomes empty.

```
bool restrict_upper(dimension_type dim, const T& ub)
```

restricts the object by applying the upper bound `ub` to the space dimension `dim` and returns `false` if and only if the object becomes empty.

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::unconstrain (Variable *var*) Computes the [cylindrification](#) of `*this` with respect to space dimension `var`, assigning the result to `*this`.

Parameters

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::intersection_assign (const BD_Shape< T > & y) Assigns to **this* the intersection of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::upper_bound_assign (const BD_Shape< T > & y) Assigns to **this* the smallest BDS containing the union of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> 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.

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::difference_assign (const BD_Shape< T > & y) Assigns to **this* the smallest BD shape containing the set difference of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T > bool Parma_Polyhedra_Library::BD_Shape< T >::simplify_using_context_assign (const BD_Shape< T > & y) Assigns to **this* a [meet-preserving simplification](#) of **this* with respect to *y*. If *false* is returned, then the intersection is empty.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T>::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one ()) Assigns to **this* the [affine image](#) of **this* under the function mapping variable *var* into the affine expression specified by *expr* and *denominator*.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>expr</i>	The numerator of the affine expression.
<i>denominator</i>	The denominator of the affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> .
------------------------------	--

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::BD_Shape< T>::affine_form_image (Variable *var*, const Linear_Form< Interval< T, Interval_Info> > & *lf*) Assigns to **this* the [affine form image](#) of **this* under the function mapping variable *var* into the affine expression(s) specified by *lf*.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>lf</i>	The linear form on intervals with floating point coefficients that defines the affine expression. ALL of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>lf</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> .
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T>::affine_preimage (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one ()) Assigns to **this* the [affine preimage](#) of **this* under the function mapping variable *var* into the affine expression specified by *expr* and *denominator*.

Parameters

<i>var</i>	The variable to which the affine expression is substituted.
<i>expr</i>	The numerator of the affine expression.
<i>denominator</i>	The denominator of the affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> .
------------------------------	--

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T>::generalized_affine_image (Variable *var*, Relation_Symbol *relsym*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_one ()) Assigns to **this* the image of **this* with respect to the [affine relation](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine transfer function.
<i>relsym</i>	The relation symbol.
<i>expr</i>	The numerator of the right hand side affine expression.
<i>denominator</i>	The denominator of the right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::BD.Shape< T >::generalized_affine_image (const Linear_Expression & lhs, Relation_Symbol relsym, const Linear_Expression & rhs) Assigns to **this* the image of **this* with respect to the [affine relation](#) $lhs' \bowtie rhs$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression.
<i>relsym</i>	The relation symbol.
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::BD.Shape< T >::generalized_affine_preimage (Variable var, Relation_Symbol relsym, const Linear_Expression & expr, Coefficient_traits::const_reference denominator = Coefficient_one ()) Assigns to **this* the preimage of **this* with respect to the [affine relation](#) $var' \bowtie \frac{expr}{denominator}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine transfer function.
<i>relsym</i>	The relation symbol.
<i>expr</i>	The numerator of the right hand side affine expression.
<i>denominator</i>	The denominator of the right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::BD.Shape< T >::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](#) $lhs' \bowtie rhs$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression.
<i>relysym</i>	The relation symbol.
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>relysym</i> is a strict relation symbol.
------------------------------	--

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 ()) Assigns to **this* the image of **this* with respect to the **bounded affine relation** $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the preimage of **this* with respect to the **bounded affine relation** $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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 * cs_p = 0, unsigned complexity_threshold = 16, bool wrap_individually = true) Wraps the specified dimensions of the vector space.
Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	Possibly null pointer to a constraint system whose variables are contained in <i>vars</i> . If * <i>cs_p</i> depends on variables not in <i>vars</i> , 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 * <i>cs_p</i> .
<i>complexity_threshold</i>	A precision parameter of the wrapping operator : higher values result in possibly improved precision.
<i>wrap_individually</i>	true if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

<i>std::invalid_argument</i>	Thrown if * <i>cs_p</i> is dimension-incompatible with <i>vars</i> , or if * <i>this</i> is dimension-incompatible <i>vars</i> or with * <i>cs_p</i> .
------------------------------	--

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::drop_some_non_integer_points (Complexity_Class complexity = ANY_COMPLEXITY) Possibly tightens **this* by dropping some points with non-integer coordinates.
Parameters

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

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

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::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*.
Parameters

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

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 <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

template<typename T > template<typename Iterator > void Parma_Polyhedra_Library::BD_Shape< T >::CC76_extrapolation_assign (const BD_Shape< T > & y, Iterator first, Iterator last, unsigned * tp = 0) Assigns to *this the result of computing the [CC76-extrapolation](#) between *this and y.

Parameters

y	A BDS that <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::BHMZ05_widening.-assign (const BD_Shape< T > & y, unsigned * tp = 0) Assigns to *this the result of computing the [BHMZ05-widening](#) of *this and y.

Parameters

y	A BDS that <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::limited_BHMZ05_extrapolation.-assign (const BD_Shape< T > & y, const Constraint_System & cs, unsigned * tp = 0) Improves the result of the [BHMZ05-widening](#) computation by also enforcing those constraints in cs that are satisfied by all the points of *this.

Parameters

y	A BDS that <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a strict inequality.
------------------------------	--

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::CC76_narrowing_assign (const BD_Shape< T > & y) Assigns to **this* the result of restoring in *y* the constraints of **this* that were lost by [CC76-extrapolation](#) applications.

Parameters

<i>y</i>	A BDS that <i>must</i> contain <i>*this</i> .
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> 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 $y\Delta x$.

template<typename T> void Parma_Polyhedra_Library::BD_Shape< T >::limited_CC76_extrapolation_assign (const BD_Shape< T > & y, const Constraint_System & cs, unsigned *tp = 0) Improves the result of the [CC76-extrapolation](#) computation by also enforcing those constraints in *cs* that are satisfied by all the points of **this*.

Parameters

<i>y</i>	A BDS that <i>must</i> be contained in <i>*this</i> .
<i>cs</i>	The system of constraints used to improve the widened BDS.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a strict inequality.
------------------------------	--

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

<i>y</i>	A BDS that <i>must</i> be contained in <i>*this</i> .
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>y</i>	A BDS that <i>must</i> be contained in <i>*this</i> .
<i>cs</i>	The system of constraints used to improve the widened BDS.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_embed (dimension_type m) Adds *m* new dimensions and embeds the old BDS into the new space.

Parameters

<i>m</i>	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

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

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::add_space_dimensions_and_project (dimension_type m) Adds *m* new dimensions to the BDS and does not embed it in the new vector space.

Parameters

<i>m</i>	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

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

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::concatenate_assign (const BD_Shape< T > & y) Assigns to **this* the [concatenation](#) of **this* and *y*, taken in this order.

Exceptions

<i>std::length_error</i>	Thrown if the concatenation would cause the vector space to exceed dimension max_space_dimension() .
--------------------------	--

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::remove_space_dimensions (const Variables_Set & vars) Removes all the specified dimensions.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the dimensions to be removed.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <i>new_dimension</i> is greater than the space dimension of <i>*this</i> .
------------------------------	--

template<typename T > template<typename Partial_Function > void Parma_Polyhedra_Library::BD_Shape< T >::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a [partial function](#).

Parameters

<i>pfunc</i>	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](#).

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::expand_space_dimension (Variable var, dimension_type m) Creates m copies of the space dimension corresponding to *var*.

Parameters

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding m new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

template<typename T > void Parma_Polyhedra_Library::BD_Shape< T >::fold_space_dimensions (const Variables_Set & vars, Variable dest) Folds the space dimensions in *vars* into *dest*.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>dest</i> or with one of the Variable objects contained in <i>vars</i> . Also thrown if <i>dest</i> is contained in <i>vars</i> .
------------------------------	---

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.

template<typename T > template<typename Interval Info > void Parma_Polyhedra_Library::BD_Shape< T >::refine_fp_interval_abstract_store (Box< Interval< T, Interval_Info > > & store) const [inline] Refines *store* with the constraints defining **this*.

Parameters

<i>store</i>	The interval floating point abstract store to refine.
--------------	---

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 $x.hash_code() == y.hash_code()$.

10.4.4 Friends And Related Function Documentation

template<typename T > std::ostream & operator<< (std::ostream & s, const BD_Shape< T > & bds) [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 “,”.

template<typename T > void swap (BD_Shape< T > & x, BD_Shape< T > & y) [related] Swaps x with y .

template<typename T > bool operator==(const BD_Shape< T > & x, const BD_Shape< T > & y) [related] Returns *true* if and only if x and y are the same BDS.

Note that x and y may be dimension-incompatible shapes: in this case, the value *false* is returned.

template<typename T > bool operator!=(const BD_Shape< T > & x, const BD_Shape< T > & y) [related] Returns *true* if and only if x and y are not the same BDS.

Note that x and y may be dimension-incompatible shapes: in this case, the value *true* is returned.

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>.

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) [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<Temp, Extended_Number_Policy>`.

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`.

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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

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 `CheckedNumber<To, Extended-Number_Policy>`.

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) [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 `CheckedNumber<Temp, Extended-Number_Policy>`.

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`.

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

template<typename T > bool operator!=(const BD_Shape< T > & x, const BD_Shape< T > & y) [related]

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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2) [related]

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, const Rounding_Dir dir) [related]

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, const Rounding_Dir dir) [related]

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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2) [related]

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, const Rounding_Dir dir) [related]

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, const Rounding_Dir dir) [related]


```
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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
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, const Rounding_Dir dir ) [related]
```

```
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, const
Rounding_Dir dir ) [related]
```

```
template<typename T > void swap ( BD_Shape< T > & x, BD_Shape< T > & y ) [related]
```

```
template<typename T > std::ostream & operator<< ( std::ostream & s, const BD_Shape< T > &
bds ) [related] The documentation for this class was generated from the following file:
```

- ppl.hh

10.5 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.*

10.5.1 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.

10.5.2 Member Function Documentation

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

10.6 Parma_Polyhedra_Library::Binary_Operator< Target > Class Template Reference

A binary operator applied to two concrete expressions.

#include <ppl.hh>

10.6.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Binary_Operator< Target >

A binary operator applied to two concrete expressions.

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

- ppl.hh

10.7 Parma_Polyhedra_Library::Binary_Operator_Common< Target > Class Template Reference

Base class for binary operator applied to two concrete expressions.

#include <ppl.hh>

Public Member Functions

- [Concrete_Expression_BOP binary_operator](#) () const
*Returns a constant identifying the operator of *this.*
- const [Concrete_Expression](#)
< Target > * [left_hand_side](#) () const
*Returns the left-hand side of *this.*
- const [Concrete_Expression](#)
< Target > * [right_hand_side](#) () const
*Returns the right-hand side of *this.*

10.7.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Binary_Operator_Common< Target >

Base class for binary operator applied to two concrete expressions.

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

- ppl.hh

10.8 Parma Polyhedra Library::Box< Interval > 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 [has_lower_bound](#) (Variable var, Coefficient &n, Coefficient &d, bool &closed) const
If the space dimension of var is unbounded below, return false. Otherwise return true and set n, d and closed accordingly.
- bool [has_upper_bound](#) (Variable var, Coefficient &n, Coefficient &d, bool &closed) const
If the space dimension of var is unbounded above, return false. Otherwise return true and set n, d and closed 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.*
- int32_t [hash_code](#) () const
*Returns a 32-bit hash code 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<<.*
- 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 &gr, 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 m_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 propagation 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_form_image` (`Variable` var, const `Linear_Form`< `ITV` > &lf)
- Assigns to $*this$ the **affine form image** of $*this$ under the function mapping variable var into the affine expression(s) specified by lf .*

 - 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** $var' \bowtie \frac{expr}{denominator}$, 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** $var' \bowtie \frac{expr}{denominator}$, 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** $lhs' \bowtie rhs$, 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* $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relysym*.
- 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* $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - 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* $\text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - 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 *cs_p=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.

Related Functions

(Note that these are not member functions.)

- template<typename ITV >
void `swap` (`Box`< ITV > &x, `Box`< ITV > &y)
Swaps x with y.
- template<typename ITV >
bool `operator==` (const `Box`< ITV > &x, const `Box`< ITV > &y)
Returns true if and only if x and y are the same box.
- template<typename ITV >
bool `operator!=` (const `Box`< ITV > &x, const `Box`< ITV > &y)
Returns true if and only if x and y are not 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)
Computes the rectilinear (or Manhattan) distance between x and y.

- [illegible]

- `template<typename Temp , typename To , typename ITV >`
`bool linfinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, const Rounding_Dir dir)`
- `template<typename To , typename ITV >`
`bool linfinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Box< ITV > &x, const Box< ITV > &y, const Rounding_Dir dir)`
- `template<typename ITV >`
`void swap (Box< ITV > &x, Box< ITV > &y)`
- `template<typename ITV >`
`std::ostream & operator<< (std::ostream &s, const Box< ITV > &box)`

10.8.1 Detailed Description

`template<typename Interval>class Parma_Polyhedra_Library::Box< Interval >`

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](#).

10.8.2 Constructor & Destructor Documentation

`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

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the box;
<i>kind</i>	Specifies whether the universe or the empty box has to be built.

`template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Box< Interval > &y, Complexity_Class complexity = ANY_COMPLEXITY) [inline]` Ordinary copy constructor.

The complexity argument is ignored.

`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.

**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

<i>cs</i>	A system of constraints: constraints that are not interval constraints are ignored (even though they may have contributed to the space dimension).
-----------	--

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

<i>cs</i>	A system of constraints: constraints that are not interval constraints are ignored (even though they may have contributed to the space dimension).
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Generator_System & gs) [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

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

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

<i>gs</i>	The generator system describing the polyhedron to be approximated.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

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

<i>cgs</i>	A system of congruences: congruences that are not non-relational equality constraints are ignored (though they may have contributed to the space dimension).
------------	--

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

<i>cgs</i>	A system of congruences: congruences that are not non-relational equality constraints are ignored (though they will contribute to the space dimension).
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

template<typename ITV > template<typename T > Parma_Polyhedra_Library::Box< ITV >::Box (const BD_Shape< T > & bds, Complexity_Class complexity = POLYNOMIAL_COMPLEXITY) [explicit] Builds a box containing the BDS *bds*.

Builds the smallest box containing *bds* using a polynomial algorithm. The *complexity* argument is ignored.

template<typename ITV > template<typename T > Parma_Polyhedra_Library::Box< ITV >::Box (const Octagonal_Shape< T > & oct, Complexity_Class complexity = POLYNOMIAL_COMPLEXITY) [explicit] Builds a box containing the octagonal shape *oct*.

Builds the smallest box containing *oct* using a polynomial algorithm. The *complexity* argument is ignored.

template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Polyhedron & ph, Complexity_Class complexity = ANY_COMPLEXITY) [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*.

template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::Box (const Grid & gr, Complexity_Class complexity = POLYNOMIAL_COMPLEXITY) [explicit] Builds a box containing the grid *gr*.

Builds the smallest box containing *gr* using a polynomial algorithm. The *complexity* argument is ignored.

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) [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*.

10.8.3 Member Function Documentation

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::constrains (Variable var) const Returns *true* if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename ITV > Poly_Con_Relation Parma_Polyhedra_Library::Box< ITV >::relation.-with (const Constraint & c) const Returns the relations holding between **this* and the constraint *c*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::relation_-with (const Congruence & *cg*) const Returns the relations holding between **this* and the congruence *cg*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>cg</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > Parma_Polyhedra_Library::Box< ITV >::relation_-with (const Generator & *g*) const Returns the relations holding between **this* and the generator *g*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	true if and only if the supremum is also the maximum value;
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <i>expr</i> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value;
<i>g</i>	When minimization succeeds, will be assigned a point or closure point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

Parameters

<i>expr</i>	The linear expression for which the frequency is needed;
<i>freq_n</i>	If <code>true</code> is returned, the value is set to 0; Present for interface compatibility with class Grid , where the frequency can have a non-zero value;
<i>freq_d</i>	If <code>true</code> is returned, the value is set to 1;
<i>val_n</i>	The numerator of <i>val</i> ;
<i>val_d</i>	The denominator of <i>val</i> ;

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::contains (const Box< Interval > & y) const Returns `true` if and only if **this* contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::strictly_contains (const Box< Interval > & y) const [inline] Returns `true` if and only if **this* strictly contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::is_disjoint_from (const Box< Interval > & y) const Returns `true` if and only if **this* and *y* are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>c</i>	The constraint to be added.
----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible, or <i>c</i> is not optimally supported by the Box domain.
------------------------------	--

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

<i>cs</i>	The constraints to be added.
-----------	------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible, or <i>cs</i> contains a constraint which is not optimally supported by the box domain.
------------------------------	---

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

<i>cs</i>	The constraints to be added. They may be recycled.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible, or <i>cs</i> 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.

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

<i>cg</i>	The congruence to be added.
-----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible, or <i>cg</i> is not optimally supported by the box domain.
------------------------------	--

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

<i>cgs</i>	The congruences to be added.
------------	------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or <i>cgs</i> contains a congruence which is not optimally supported by the box domain.
------------------------------	---

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

<i>cgs</i>	The congruence system to be added to <i>*this</i> . The congruences in <i>cgs</i> may be recycled.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or <i>cgs</i> 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.

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_constraint (const Constraint & c) [inline] Use the constraint *c* to refine **this*.

Parameters

<i>c</i>	The constraint to be used for refinement.
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible.
------------------------------	---

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

<i>cs</i>	The constraints to be used for refinement. To avoid termination problems, each constraint in <i>cs</i> will be used for a single refinement step.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> 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.

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::refine_with_congruence (const Congruence & cg) [inline] Use the congruence *cg* to refine **this*.

Parameters

<i>cg</i>	The congruence to be used for refinement.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cg</i> are dimension-incompatible.
------------------------------	--

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

<i>cgs</i>	The congruences to be used for refinement.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

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

<i>c</i>	The constraint to be used for constraint propagation.
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible.
------------------------------	---

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 propagation on **this*.

Parameters

<i>cs</i>	The constraints to be used for constraint propagation.
<i>max_iterations</i>	The maximum number of propagation steps for each constraint in <i>cs</i> . If zero (the default), the number of propagation steps will be unbounded, possibly resulting in an infinite loop.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

Warning

This method may lead to non-termination if *max_iterations* is 0.

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

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::intersection_assign (const Box< Interval > & y) Assigns to **this* the intersection of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::upper_bound_assign (const Box< Interval > & y) Assigns to **this* the smallest box containing the union of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::upper_bound_assign_if_exact (const Box< Interval > & 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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::difference_assign (const Box< Interval > & y) Assigns to **this* the difference of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::simplify_using_context_assign (const Box< Interval > & y) Assigns to **this* a [meet-preserving simplification](#) of **this* with respect to *y*. If *false* is returned, then the intersection is empty.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::affine_image (Variable *var*, const Linear_Expression & *expr*, Coefficient_traits::const_reference *denominator* = Coefficient_traits::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

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::affine_form_image (Variable `var`, const Linear_Form< ITV > & `lf`) Assigns to `*this` the [affine form image](#) of `*this` under the function mapping variable `var` into the affine expression(s) specified by `lf`.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>lf</i>	The linear form on intervals with floating point boundaries that defines the affine expression(s). ALL of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>lf</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a dimension of <code>*this</code> .
------------------------------	---

This function is used in abstract interpretation to model an assignment of a value that is correctly overapproximated by `lf` to the floating point variable represented by `var`.

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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

<i>var</i>	The variable to which the affine expression is substituted;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

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()) Assigns to `*this` the image of `*this` with respect to the [generalized affine relation](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by `relsym`.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

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()) Assigns to **this* the preimage of **this* with respect to the [generalized affine relation](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relsym*.
Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relsym*.
Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> .
------------------------------	--

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relsym*.
Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> .
------------------------------	--

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()) Assigns to **this* the image of **this* with respect to the [bounded affine relation](#) $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the preimage of **this* with respect to the **bounded affine relation** $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::time_elapse_assign (const Box< Interval > & *y*) Assigns to **this* the result of computing the **time-elapse** between **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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 * *cs* *p* = 0, unsigned complexity_threshold = 16, bool wrap_individually = true) Wraps the specified dimensions of the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.

<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	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 <i>*cs_p</i> .
<i>complexity-threshold</i>	A precision parameter which is ignored for the Box domain.
<i>wrap-individually</i>	A precision parameter which is ignored for the Box domain.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with one of the Variable objects contained in <i>vars</i> or with <i>*cs_p</i> .
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::drop_some_non_integer_points (Complexity_Class *complexity* = ANY_COMPLEXITY) Possibly tightens **this* by dropping some points with non-integer coordinates.

Parameters

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

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

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::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*.

Parameters

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

template<typename Interval > template<typename T > Enable_If<Is_Same<T, Box>::value && Is_Same_Or_Derived<Interval_Base, ITV>::value, void>::type Parma_Polyhedra_Library::Box< Interval >::CC76_widening_assign (const T & *y*, unsigned * *tp* = 0) Assigns to **this* the result of computing the [CC76-widening](#) between **this* and *y*.

Parameters

<i>y</i>	A box that <i>must</i> be contained in <i>*this</i> .
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename Interval > template<typename T, typename Iterator > Enable_If<Is_Same<T, Box>::value && Is_Same_Or_Derived<Interval_Base, ITV>::value, void>::type Parma_Polyhedra_Library::Box< Interval >::CC76_widening_assign (const T & y, Iterator *first*, Iterator *last*)
Assigns to **this* the result of computing the [CC76-widening](#) between **this* and *y*.

Parameters

<i>y</i>	A box that <i>must</i> be contained in <i>*this</i> .
<i>first</i>	An iterator that points to the first stop-point.
<i>last</i>	An iterator that points one past the last stop-point.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::limited_CC76_extrapolation_assign (const Box< Interval > & y, const Constraint_System & cs, unsigned * *tp* = 0) Improves the result of the [CC76-extrapolation](#) computation by also enforcing those constraints in *cs* that are satisfied by all the points of **this*.

Parameters

<i>y</i>	A box that <i>must</i> be contained in <i>*this</i> .
<i>cs</i>	The system of constraints used to improve the widened box.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a strict inequality.
------------------------------	--

template<typename Interval > template<typename T > Enable_If<Is_Same<T, Box>::value && Is_Same_Or_Derived<Interval_Base, ITV>::value, void>::type Parma_Polyhedra_Library::Box< Interval >::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.

Parameters

<i>y</i>	A Box that <i>must</i> contain <i>*this</i> .
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> 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 $y\Delta x$.

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

<i>m</i>	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

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

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

<i>m</i>	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

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

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::concatenate_assign (const Box< Interval > & *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*.

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

$$R \stackrel{\text{def}}{=} \left\{ (x_1, \dots, x_n, y_1, \dots, y_m)^T \mid (x_1, \dots, x_n)^T \in B, (y_1, \dots, y_m)^T \in D \right\}.$$

Another way of seeing it is as follows: first increases the space dimension of **this* by adding *y.space_dimension()* new dimensions; then adds to the system of constraints of **this* a renamed-apart version of the constraints of *y*.

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::remove_space_dimensions (const Variables_Set & *vars*) [inline] Removes all the specified dimensions.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the dimensions to be removed.
-------------	--

Exceptions

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

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::remove_higher_space_dimensions (dimension_type *new_dimension*) Removes the higher dimensions so that the resulting space will have dimension *new_dimension*.

Exceptions

<code>std::invalid_argument</code>	Thrown if <code>new_dimension</code> is greater than the space dimension of <code>*this</code> .
------------------------------------	--

template<typename ITV > template<typename Partial_Function > void Parma_Polyhedra_Library::Box< ITV >::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a [partial function](#).

Parameters

<code>pfunc</code>	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](#).

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

<code>var</code>	The variable corresponding to the space dimension to be replicated;
<code>m</code>	The number of replicas to be created.

Exceptions

<code>std::invalid_argument</code>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<code>std::length_error</code>	Thrown if adding <code>m</code> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

template<typename ITV > void Parma_Polyhedra_Library::Box< ITV >::fold_space_dimensions (const Variables_Set & vars, Variable dest) Folds the space dimensions in `vars` into `dest`.

Parameters

<code>vars</code>	The set of Variable objects corresponding to the space dimensions to be folded;
<code>dest</code>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>dest</i> or with one of the Variable objects contained in <i>vars</i> . Also thrown if <i>dest</i> is contained in <i>vars</i> .
------------------------------	---

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.

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

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::has_lower_bound (Variable var, Coefficient & n, Coefficient & d, bool & closed) const [inline] If the space dimension of *var* is unbounded below, return *false*. Otherwise return *true* and set *n*, *d* and *closed* accordingly.

Note

It is assumed that **this* is a non-empty box having space dimension greater than or equal to that of *var*. An undefined behavior is obtained if this assumption is not met.

Let I be the interval corresponding to variable *var* in the non-empty box **this*. If I is not bounded from below, simply return *false* (leaving all other parameters unchanged). Otherwise, set *n*, *d* and *closed* as follows:

- *n* and *d* are assigned the integers n and d such that the fraction n/d corresponds to the greatest lower bound of I . The fraction n/d is in canonical form, meaning that n and d have no common factors, d is positive, and if n is zero then d is one;
- *closed* is set to *true* if and only if the lower boundary of I is closed (i.e., it is included in the interval).

template<typename ITV > bool Parma_Polyhedra_Library::Box< ITV >::has_upper_bound (Variable var, Coefficient & n, Coefficient & d, bool & closed) const [inline] If the space dimension of *var* is unbounded above, return *false*. Otherwise return *true* and set *n*, *d* and *closed* accordingly.

Note

It is assumed that **this* is a non-empty box having space dimension greater than or equal to that of *var*. An undefined behavior is obtained if this assumption is not met.

Let I be the interval corresponding to variable *var* in the non-empty box **this*. If I is not bounded from above, simply return *false* (leaving all other parameters unchanged). Otherwise, set *n*, *d* and *closed* as follows:

- n and d are assigned the integers n and d such that the fraction n/d corresponds to the least upper bound of I . The fraction n/d is in canonical form, meaning that n and d have no common factors, d is positive, and if n is zero then d is one;
- `closed` is set to `true` if and only if the upper boundary of I is closed (i.e., it is included in the interval).

template<typename ITV > int32_t Parma_Polyhedra_Library::Box< ITV >::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()`.

10.8.4 Friends And Related Function Documentation

template<typename ITV > void swap (Box< ITV > &x, Box< ITV > &y) [related] Swaps x with y .

template<typename ITV > bool operator== (const Box< ITV > &x, const Box< ITV > &y)
[related] 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.

template<typename ITV > bool operator!= (const Box< ITV > &x, const Box< ITV > &y)
[related] Returns `true` if and only if x and y are not the same box.

Note that x and y may be dimension-incompatible boxes: in this case, the value `true` is returned.

template<typename ITV > std::ostream & operator<< (std::ostream &s, const Box< ITV > &box) [related] 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) [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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

template<typename To , typename ITV > bool Linfinity_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>`.

template<typename Temp , typename To , typename ITV > bool Linfinity_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<Temp, Extended_Number_Policy>`.

template<typename Temp , typename To , typename ITV > bool Linfinity_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`.


```
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,
const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
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,
const Rounding_Dir dir ) [related]
```

```
template<typename To, typename ITV> bool rectilinear_distance_assign ( Checked_Number< To,
Extended_Number_Policy> & r, const Box< ITV> & x, const Box< ITV> & y, const Rounding-
_Dir dir ) [related]
```

```
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,
const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
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,
const Rounding_Dir dir ) [related]
```

```
template<typename To, typename ITV> bool euclidean_distance_assign ( Checked_Number< To,
Extended_Number_Policy> & r, const Box< ITV> & x, const Box< ITV> & y, const Rounding-
_Dir dir ) [related]
```

```
template<typename Temp, typename To, typename ITV> bool Linfinity_distance_assign ( Checked-
_Number< To, Extended_Number_Policy> & r, const Box< ITV> & x, const Box< ITV> & y,
const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
template<typename Temp, typename To, typename ITV> bool Linfinity_distance_assign ( Checked-
_Number< To, Extended_Number_Policy> & r, const Box< ITV> & x, const Box< ITV> & y,
const Rounding_Dir dir ) [related]
```

```
template<typename To, typename ITV> bool Linfinity_distance_assign ( Checked_Number< To,
Extended_Number_Policy> & r, const Box< ITV> & x, const Box< ITV> & y, const Rounding-
_Dir dir ) [related]
```

```
template<typename ITV> void swap ( Box< ITV> & x, Box< ITV> & y ) [related]
```

```
template<typename ITV> std::ostream & operator<< ( std::ostream & s, const Box< ITV> &
box ) [related] The documentation for this class was generated from the following file:
```

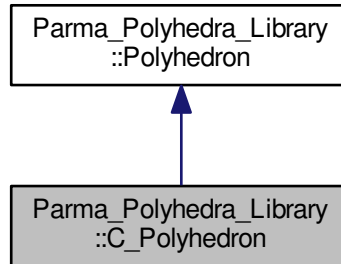
- ppl.hh

10.9 Parma_Polyhedra_Library::C_Polyhedron Class Reference

A closed convex polyhedron.

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

Inheritance diagram for Parma_Polyhedra_Library::C_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)`.
 - `void positive_time_elapse_assign (const Polyhedron &y)`
Assigns to **this* the smallest *C* polyhedron containing the result of computing the *positive time-elapse* between **this* and *y*.

Additional Inherited Members

10.9.1 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.

10.9.2 Constructor & Destructor Documentation

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

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the C polyhedron;
<i>kind</i>	Specifies whether a universe or an empty C polyhedron should be built.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

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

<i>cs</i>	The system of constraints defining the polyhedron.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of constraints contains strict inequalities.
------------------------------	---

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

<i>cs</i>	The system of constraints defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of constraints contains strict inequalities.
------------------------------	---

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

<i>gs</i>	The system of generators defining the polyhedron.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points, or if it contains closure points.
------------------------------	--

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

<i>gs</i>	The system of generators defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points, or if it contains closure points.
------------------------------	--

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

<i>cgs</i>	The system of congruences defining the polyhedron.
------------	--

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

<i>cgs</i>	The system of congruences defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

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

<i>y</i>	The NNC polyhedron to be used;
<i>complexity</i>	This argument is ignored.

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

<i>box</i>	The box representing the polyhedron to be approximated;
<i>complexity</i>	This argument is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>box</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>bd</i>	The BDS used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

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

<i>os</i>	The octagonal shape used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

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

<i>grid</i>	The grid used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

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.

10.9.3 Member Function Documentation

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::C_Polyhedron::positive_time_elapse_assign (const Polyhedron & *y*) Assigns to **this* the smallest C polyhedron containing the result of computing the [positive time-elapse](#) between **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

- ppl.hh

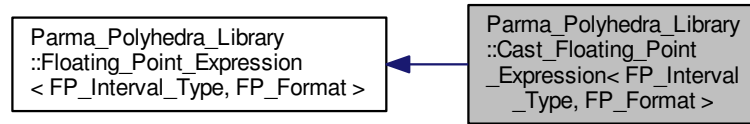
10.10 Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic Cast Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type,

FP.Format >:



Public Types

- typedef
[Floating_Point_Expression](#)
 < FP.Interval_Type, FP.Format >
[::FP_Linear_Form](#) [FP_Linear_Form](#)
Alias for the Linear_Form<FP.Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP.Interval_Type, FP.Format >
[::FP_Interval_Abstract_Store](#) [FP_Interval_Abstract_Store](#)
Alias for the Box<FP.Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP.Interval_Type, FP.Format >
[::FP_Linear_Form_Abstract_Store](#) [FP_Linear_Form_Abstract_Store](#)
Alias for the std::map<dimension_type, FP.Linear_Form> from [Floating_Point_Expression](#).

Public Member Functions

- bool [linearize](#) (const [FP_Interval_Abstract_Store](#) &int_store, const [FP_Linear_Form_Abstract_Store](#) &lf_store, [FP_Linear_Form](#) &result) const
Linearizes the expression in a given astract store.
- void [m_swap](#) ([Cast_Floating_Point_Expression](#) &y)
*Swaps *this with y.*

Constructors and Destructor

- [Cast_Floating_Point_Expression](#) ([Floating_Point_Expression](#)< FP.Interval_Type, FP.Format > *const expr)
Builds a cast floating point expression with the value expressed by expr.
- [~Cast_Floating_Point_Expression](#) ()
Destructor.

Related Functions

(Note that these are not member functions.)

- template<typename FP.Interval_Type, typename FP.Format >
 void [swap](#) ([Cast_Floating_Point_Expression](#)< FP.Interval_Type, FP.Format > &x, [Cast_Floating_Point_Expression](#)< FP.Interval_Type, FP.Format > &y)

Swaps x with y .

- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap (Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`

Additional Inherited Members

10.10.1 Detailed Description

template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >

A generic Cast Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of floating-point cast expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms and $\boxplus^\#$ a sound abstract operator on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \oplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \oplus^\# i'_v) v.$$

Given a floating point expression e and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket \text{cast}(e) \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket \text{cast}(e) \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket e \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \varepsilon_f \left(\llbracket e \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# m_{f_f}[-1, 1]$$

where $\varepsilon_f(l)$ is the linear form computed by calling method `Floating_Point_Expression::relative_error` on l and m_{f_f} is a rounding error defined in `Floating_Point_Expression::absolute_error`.

10.10.2 Member Function Documentation

template<typename FP_Interval_Type, typename FP_Format > bool Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & int_store, const FP_Linear_Form_Abstract_Store & lf_store, FP_Linear_Form & result) const [virtual] Linearizes the expression in a given astract store.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

See the class description for an explanation of how `result` is computed.

Implements `Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >`.

10.10.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] Swaps x with y .

template<typename FP_Interval_Type , typename FP_Format > void swap (Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] The documentation for this class was generated from the following file:

- ppl.hh

10.11 Parma_Polyhedra_Library::Cast_Operator< Target > Class Template Reference

A cast operator converting one concrete expression to some type.

#include <ppl.hh>

10.11.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Cast_Operator< Target >

A cast operator converting one concrete expression to some type.

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

- ppl.hh

10.12 Parma_Polyhedra_Library::Cast_Operator_Common< Target > Class Template Reference

Base class for cast operator concrete expressions.

#include <ppl.hh>

10.12.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Cast_Operator_Common< Target >

Base class for cast operator concrete expressions.

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

- ppl.hh

10.13 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 infinity_sign (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)`
- `template<typename T , typename Policy >`
`memory_size_type total_memory_in_bytes (const Checked_Number< T, Policy > &x)`
- `template<typename T , typename Policy >`
`memory_size_type external_memory_in_bytes (const Checked_Number< T, Policy > &x)`
- `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 T , typename Policy >`
`Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > &x)`
- `template<typename T , typename Policy >`
`Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > &x)`
- `template<typename From >`
`Enable_If`
`< Is_Native_Or_Checked< From >`
`::value, int >::type sgn (const From &x)`
- `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)`
- `template<typename T >`
`Enable_If`
`< Is_Native_Or_Checked< T >`
`::value, Result >::type output (std::ostream &os, const T &x, const Numeric_Format &format, Rounding_Dir dir)`
- `template<typename T , typename Policy >`
`std::ostream & operator<< (std::ostream &os, const Checked_Number< T, Policy > &x)`
- `template<typename T >`
`Enable_If`
`< Is_Native_Or_Checked< T >`
`::value, Result >::type input (T &x, std::istream &is, Rounding_Dir dir)`
- `template<typename T , typename Policy >`
`std::istream & operator>> (std::istream &is, Checked_Number< T, Policy > &x)`
- `template<typename T , typename Policy >`
`void swap (Checked_Number< T, Policy > &x, Checked_Number< T, Policy > &y)`

Memory Size Inspection Functions

- `template<typename T, typename Policy >`
`memory_size_type 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 $s*y + t*z = x = \text{gcd}(y, z)$.*
- `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 $y \cdot 2^{\text{exp}}$.
- `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 $y/2^{\text{exp}}$.
- `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, 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 &&(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, 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 &&(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, 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 &&(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, 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 &&(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, 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 &&(Is_Checked`
`< T1 >::value||Is_Checked< T2 >`
`::value), bool >::type operator< (const T1 &x, const T2 &y)`
Less than operator.

- `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)
- `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 &format, 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, 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, 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.
- `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.

10.13.1 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.

10.13.2 Member Function Documentation

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*.

10.13.3 Friends And Related Function Documentation

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_not_a_-number (const T & *x*) **[related]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_minus_-infinity (const T & *x*) **[related]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_plus_-infinity (const T & *x*) **[related]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, int >::type infinity_sign (const T & *x*) **[related]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type is_integer (const T & *x*) **[related]**

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]**

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]**

template<typename To > Enable_If< Is_Native_Or_Checked< To >::value, Result >::type assign_r (To & *to*, const char * *x*, Rounding_Dir *dir*) **[related]**

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]**

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*.

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*.

template<typename T, typename Policy > Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > & x) [related] Unary plus operator.

template<typename T, typename Policy > Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > & x) [related] Unary minus operator.

template<typename T, typename Policy > void floor_assign (Checked_Number< T, Policy > & x) [related] 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) [related] Assigns to x largest integral value not greater than y .

template<typename T, typename Policy > void ceil_assign (Checked_Number< T, Policy > & x) [related] 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) [related] Assigns to x smallest integral value not less than y .

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.

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.

template<typename T, typename Policy > void neg_assign (Checked_Number< T, Policy > & x) [related] Assigns to x its negation.

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 .

template<typename T, typename Policy > void abs_assign (Checked_Number< T, Policy > & x) [related] 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) [related] 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) [related] 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) [related] 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) [related] 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) [related] Assigns to x the greatest common divisor of y and z , setting s and t such that $s*y + t*z = x = \gcd(y, z)$.

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 .

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 $y \cdot 2^{\text{exp}}$.

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 $y/2^{\text{exp}}$.

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) [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 .

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 .

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.

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) [related]

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.

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) [related]

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.

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) [related]

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.

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) **[related]**

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.

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) **[related]**

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.

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) **[related]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, Result >::type output (std::ostream & os, const T & x, const Numeric_Format & format, Rounding_Dir dir) **[related]**

template<typename T , typename Policy > std::ostream & operator<< (std::ostream & os, const Checked_Number< T, Policy > & x) **[related]** Output operator.

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, void >::type ascii_dump (std::ostream & s, const T & t) **[related]** Ascii dump for native or checked.

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

<i>is</i>	Input stream to read from;
<i>x</i>	Number (possibly extended) to assign to in case of successful reading;
<i>dir</i>	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*^+11111111111111111111;
- 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 $*^$).

<code>number</code>	: <code>NAN</code> <code>SIGN INF</code> <code>INF</code> <code>num</code> <code>num DIV num</code> ;	<code>INF</code>	: <code>'inf'</code> ;
<code>num</code>	: <code>u_num</code> <code>SIGN u_num</code> ;	<code>NAN</code>	: <code>'nan'</code> ;
<code>u_num</code>	: <code>u_num1</code> <code>HEX u_num1</code> <code>base BASE u_num1</code> ;	<code>SIGN</code>	: <code>'-'</code> <code>'+'</code> ;
<code>u_num1</code>	: <code>mantissa</code> <code>mantissa EXP exponent</code> ;	<code>EXP</code>	: <code>'e'</code> <code>'p'</code> <code>'*^'</code> ;
<code>mantissa</code>	: <code>bdigits</code> <code>POINT bdigits</code> <code>bdigits POINT</code> <code>bdigits POINT bdigits</code> ;	<code>POINT</code>	: <code>'.'</code> ;
<code>exponent</code>	: <code>SIGN digits</code> <code>digits</code> ;	<code>DIV</code>	: <code>'/'</code> ;
<code>bdigits</code>	: <code>BDIGIT</code>	<code>MINUS</code>	: <code>'-'</code> ;
		<code>PLUS</code>	: <code>'+'</code> ;
		<code>HEX</code>	: <code>'0x'</code> ;
		<code>BASE</code>	: <code>'^^'</code>

```

        | bdigits BDIGIT
    ;

    DIGIT : '0' .. '9'
    ;

    BDIGIT : '0' .. '9'
            | 'a' .. 'z'
    ;

```

template<typename T, typename Policy > std::istream & operator>> (std::istream & *is*, Checked_Number< T, Policy > & *x*) **[related]** Input operator.

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, bool >::type **ascii_load (std::ostream & *s*, T & *t*)** **[related]** Ascii load for native or checked.

template<typename T, typename Policy > void **swap (Checked_Number< T, Policy > & *x*, Checked_Number< T, Policy > & *y*)** **[related]** Swaps *x* with *y*.

template<typename T, typename Policy > const T & raw_value (const Checked_Number< T, Policy > & *x*) **[related]**

template<typename T, typename Policy > T & raw_value (Checked_Number< T, Policy > & *x*) **[related]**

template<typename T, typename Policy > memory_size_type **total_memory_in_bytes (const Checked_Number< T, Policy > & *x*)** **[related]**

template<typename T, typename Policy > memory_size_type **external_memory_in_bytes (const Checked_Number< T, Policy > & *x*)** **[related]**

template<typename To > Enable_If< Is_Native_Or_Checked< To >::value, Result >::type **assign_r (To & *to*, const char * *x*, Rounding_Dir *dir*)** **[related]**

template<typename T, typename Policy > Checked_Number< T, Policy > operator+ (const Checked_Number< T, Policy > & *x*) **[related]**

template<typename T, typename Policy > Checked_Number< T, Policy > operator- (const Checked_Number< T, Policy > & *x*) **[related]**

template<typename From > Enable_If< Is_Native_Or_Checked< From >::value, int >::type **sgn (const From & *x*)** **[related]**

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]**

template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, Result >::type **output (std::ostream & *os*, const T & *x*, const Numeric_Format & *format*, Rounding_Dir *dir*)** **[related]**

template<typename T, typename Policy > std::ostream & operator<< (std::ostream & *os*, const Checked_Number< T, Policy > & *x*) **[related]**


```
template<typename T > Enable_If< Is_Native_Or_Checked< T >::value, Result >::type input ( T
& x, std::istream & is, Rounding_Dir dir ) [related]
```

```
template<typename T, typename Policy > std::istream & operator>> ( std::istream & is, Checked-
_Number< T, Policy > & x ) [related]
```

```
template<typename T, typename Policy > void swap ( Checked_Number< T, Policy > & x, Checked-
_Number< T, Policy > & y ) [related] The documentation for this class was generated from the
following file:
```

- ppl.hh

10.14 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.

10.14.1 Detailed Description

Binary predicate defining the total ordering on variables.

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

- ppl.hh

10.15 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.

10.15.1 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

10.16 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.

10.16.1 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

10.17 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.

10.17.1 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

10.18 Parma Polyhedra Library::Concrete_Expression< Target > Class Template Reference

The base class of all concrete expressions.

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

Related Functions

(Note that these are not member functions.)

- template<typename Target , typename FP.Interval.Type >
static bool [add_linearize](#) (const [Binary.Operator](#)< Target > &bop_expr, const [FP.Oracle](#)< Target, FP.Interval.Type > &oracle, const std::map< [dimension.type](#), [Linear.Form](#)< FP.Interval.Type > > &lf_store, [Linear.Form](#)< FP.Interval.Type > &result)
- template<typename Target , typename FP.Interval.Type >
static bool [sub_linearize](#) (const [Binary.Operator](#)< Target > &bop_expr, const [FP.Oracle](#)< Target, FP.Interval.Type > &oracle, const std::map< [dimension.type](#), [Linear.Form](#)< FP.Interval.Type > > &lf_store, [Linear.Form](#)< FP.Interval.Type > &result)
- template<typename Target , typename FP.Interval.Type >
static bool [mul_linearize](#) (const [Binary.Operator](#)< Target > &bop_expr, const [FP.Oracle](#)< Target, FP.Interval.Type > &oracle, const std::map< [dimension.type](#), [Linear.Form](#)< FP.Interval.Type > > &lf_store, [Linear.Form](#)< FP.Interval.Type > &result)

- `template<typename Target , typename FP.Interval.Type >`
`static bool div_linearize (const Binary.Operator< Target > &bop_expr, const FP_Oracle< Target, FP.Interval.Type > &oracle, const std::map< dimension.type, Linear.Form< FP.Interval.Type > > &lf_store, Linear.Form< FP.Interval.Type > &result)`
- `template<typename Target , typename FP.Interval.Type >`
`static bool cast_linearize (const Cast.Operator< Target > &cast_expr, const FP_Oracle< Target, FP.Interval.Type > &oracle, const std::map< dimension.type, Linear.Form< FP.Interval.Type > > &lf_store, Linear.Form< FP.Interval.Type > &result)`
- `template<typename Target , typename FP.Interval.Type >`
`bool linearize (const Concrete.Expression< Target > &expr, const FP_Oracle< Target, FP.Interval.Type > &oracle, const std::map< dimension.type, Linear.Form< FP.Interval.Type > > &lf_store, Linear.Form< FP.Interval.Type > &result)`

Linearizes a floating point expression.

10.18.1 Detailed Description

`template<typename Target>class Parma_Polyhedra_Library::Concrete_Expression< Target >`

The base class of all concrete expressions.

10.18.2 Friends And Related Function Documentation

`template<typename Target , typename FP.Interval.Type > static bool add_linearize (const Binary.Operator< Target > & bop_expr, const FP_Oracle< Target, FP.Interval.Type > & oracle, const std::map< dimension.type, Linear.Form< FP.Interval.Type > > & lf_store, Linear.Form< FP.Interval.Type > & result)` [**related**] Helper function used by `linearize` to linearize a sum of floating point expressions.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete.Expression to be used.
<i>FP.Interval.Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>bop_expr</i>	The binary operator concrete expression to linearize. Its binary operator type must be <code>ADD</code> .
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Linearization of sum floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms and $\boxplus^\#$ a sound abstract operator on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxplus^\# i'_v) v.$$

Given an expression $e_1 \oplus e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \oplus e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle e_1 \oplus e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \varepsilon_f \left(\langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \varepsilon_f \left(\langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# mf_f[-1, 1]$$

where $\varepsilon_f(l)$ is the relative error associated to l (see method `relative_error` of class [Linear_Form](#)) and mf_f is a rounding error computed by function `compute_absolute_error`.

template<typename Target , typename FP_Interval_Type > static bool sub_linearize (const Binary_Operator< Target > & bop_expr, const FP_Oracle< Target, FP_Interval_Type > & oracle, const std::map< dimension_type, Linear_Form< FP_Interval_Type > > & lf_store, Linear_Form< FP_Interval_Type > & result) [related] Helper function used by `linearize` to linearize a difference of floating point expressions.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete_Expression to be used.
<i>FP_Interval_Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>bop_expr</i>	The binary operator concrete expression to linearize. Its binary operator type must be SUB.
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Linearization of difference floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxminus^\#$ two sound abstract operators on linear form such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxplus^\# i'_v) v,$$

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxminus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxminus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxminus^\# i'_v) v.$$

Given an expression $e_1 \ominus e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \ominus e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ on \mathcal{V} as follows:

$$\langle e_1 \ominus e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \boxminus^\# \langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \varepsilon_f \left(\langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \varepsilon_f \left(\langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# mf_f[-1, 1]$$

where $\varepsilon_f(l)$ is the relative error associated to l (see method `relative_error` of class [Linear_Form](#)) and mf_f is a rounding error computed by function `compute_absolute_error`.

template<typename **Target** , typename **FP_Interval_Type** > **static bool mul_linearize** (**const** **Binary_**
_Operator< **Target** > & *bop_expr*, **const** **FP_Oracle**< **Target**, **FP_Interval_Type** > & *oracle*, **const**
std::map< **dimension_type**, **Linear_Form**< **FP_Interval_Type** > > & *lf_store*, **Linear_Form**< **FP_**
Interval_Type > & *result*) **[related]** Helper function used by `linearize` to linearize a product
of floating point expressions.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete_Expression to be used.
<i>FP_Interval_Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>bop_expr</i>	The binary operator concrete expression to linearize. Its binary operator type must be <code>MUL</code> .
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Linearization of multiplication floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxtimes^\#$ two sound abstract operators on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxplus^\# i'_v) v,$$

$$i \boxtimes^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxtimes^\# i') + \sum_{v \in \mathcal{V}} (i \boxtimes^\# i'_v) v.$$

Given an expression $[a, b] \otimes e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket [a, b] \otimes e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket [a, b] \otimes e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \left([a, b] \boxtimes^\# \llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \left([a, b] \boxtimes^\# \varepsilon_f \left(\llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \right) \boxplus^\# m_{\mathbf{f}}[-1, 1].$$

Given an expression $e_1 \otimes [a, b]$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \otimes [a, b] \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket e_1 \otimes [a, b] \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket [a, b] \otimes e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket.$$

Given an expression $e_1 \otimes e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \otimes e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket e_1 \otimes e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \rho^\# \otimes e_2 \llbracket \rho^\#, \rho_l^\# \rrbracket,$$

where $\varepsilon_f(l)$ is the relative error associated to l (see method `relative_error` of class [Linear_Form](#)), $\iota(l)\rho^\#$ is the intervalization of l (see method `intervalize` of class [Linear_Form](#)), and m_{f_f} is a rounding error computed by function `compute_absolute_error`.

Even though we intervalize the first operand in the above example, the actual implementation utilizes an heuristics for choosing which of the two operands must be intervalized in order to obtain the most precise result.

template<typename Target , typename FP_Interval_Type > static bool div_linearize (const Binary_Operator< Target > & bop_expr, const FP_Oracle< Target, FP_Interval_Type > & oracle, const std::map< dimension_type, Linear_Form< FP_Interval_Type > > & lf_store, Linear_Form< FP_Interval_Type > & result) [**related**] Helper function used by `linearize` to linearize a division of floating point expressions.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete_Expression to be used.
<i>FP_Interval_Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>bop_expr</i>	The binary operator concrete expression to linearize. Its binary operator type must be <code>DIV</code> .
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Linearization of division floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxminus^\#$ two sound abstract operator on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxplus^\# i'_v) v,$$

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxminus^\# i' = (i \boxminus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxminus^\# i'_v) v.$$

Given an expression $e_1 \odot [a, b]$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \odot [a, b] \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle e_1 \odot [a, b] \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \left(\langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \boxminus^\# [a, b] \right) \boxplus^\# \left(\varepsilon_f \left(\langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxminus^\# [a, b] \right) \boxplus^\# m_{f_f}[-1, 1],$$

given an expression $e_1 \odot e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \odot e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle e_1 \odot e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \langle e_1 \odot \iota \left(\langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \rho^\# \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket,$$

where $\varepsilon_f(l)$ is the relative error associated to l (see method `relative_error` of class [Linear_Form](#)), $\iota(l)\rho^\#$ is the intervalization of l (see method `intervalize` of class [Linear_Form](#)), and m_{f_f} is a rounding error computed by function `compute_absolute_error`.

template<typename Target , typename FP_Interval_Type > static bool cast_linearize (const Cast_Operator< Target > & cast_expr, const FP_Oracle< Target, FP_Interval_Type > & oracle, const std::map< dimension_type, Linear_Form< FP_Interval_Type > > & lf_store, Linear_Form< FP_Interval_Type > & result) **[related]** Helper function used by `linearize` to linearize a cast floating point expression.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete_Expression to be used.
<i>FP_Interval_Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>cast_expr</i>	The cast operator concrete expression to linearize.
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

template<typename Target , typename FP_Interval_Type > bool linearize (const Concrete_Expression< Target > & expr, const FP_Oracle< Target, FP_Interval_Type > & oracle, const std::map< dimension_type, Linear_Form< FP_Interval_Type > > & lf_store, Linear_Form< FP_Interval_Type > & result) **[related]** Linearizes a floating point expression.

Makes `result` become a linear form that correctly approximates the value of `expr` in the given composite abstract store.

Template Parameters

<i>Target</i>	A type template parameter specifying the instantiation of Concrete_Expression to be used.
<i>FP_Interval_Type</i>	A type template parameter for the intervals used in the abstract domain. The interval bounds should have a floating point type.

Returns

`true` if the linearization succeeded, `false` otherwise.

Parameters

<i>expr</i>	The concrete expression to linearize.
<i>oracle</i>	The FP_Oracle to be queried.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	Becomes the linearized expression.

Formally, if `expr` represents the expression e and `lf_store` represents the linear form abstract store $\rho_l^\#$, then `result` will become $\langle e \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ if the linearization succeeds.

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

- `ppl.hh`

10.19 Parma Polyhedra Library::Concrete_Expression_Common< Target > Class Template Reference

Base class for all concrete expressions.

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

Public Member Functions

- [Concrete_Expression_Type](#) type () const
*Returns the type of `*this`.*
- [Concrete_Expression_Kind](#) kind () const
*Returns the kind of `*this`.*
- template<template< typename T > class Derived>
bool [is](#) () const
*Tests if `*this` has the same kind as `Derived<Target>`.*
- template<template< typename T > class Derived>
Derived< Target > * [as](#) ()
*Returns a pointer to `*this` converted to type `Derived<Target>*`.*
- template<template< typename T > class Derived>
const Derived< Target > * [as](#) () const
*Returns a pointer to `*this` converted to type `const Derived<Target>*`.*

10.19.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Concrete_Expression_Common< Target >

Base class for all concrete expressions.

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

- ppl.hh

10.20 Parma_Polyhedra_Library::Concrete_Expression_Type Class Reference

The type of a concrete expression.

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

Public Member Functions

- bool [is_bounded_integer](#) () const
*Returns `true` if and only if `*this` is a bounded integer type.*
- bool [is_floating_point](#) () const
*Returns `true` if and only if `*this` is a floating point type.*
- [Bounded_Integer_Type_Width](#) [bounded_integer_type_width](#) () const
*Returns the width in bits of the bounded integer type encoded by `*this`.*
- [Bounded_Integer_Type_Representation](#) [bounded_integer_type_representation](#) () const
*Returns the representation of the bounded integer type encoded by `*this`.*
- [Bounded_Integer_Type_Overflow](#) [bounded_integer_type_overflow](#) () const
*Returns the overflow behavior of the bounded integer type encoded by `*this`.*
- [Floating_Point_Format](#) [floating_point_format](#) () const
*Returns the format of the floating point type encoded by `*this`.*
- bool [OK](#) () const
Checks if all the invariants are satisfied.

Static Public Member Functions

- static `Concrete_Expression_Type` `bounded_integer` (`Bounded_Integer_Type_Width` width, `Bounded_Integer_Type_Representation` representation, `Bounded_Integer_Type_Overflow` overflow)
Returns the bounded integer type corresponding to width, representation and overflow.
- static `Concrete_Expression_Type` `floating_point` (`Floating_Point_Format` format)
Returns the floating point type corresponding to format.

10.20.1 Detailed Description

The type of a concrete expression.

10.20.2 Member Function Documentation

Bounded_Integer_Type_Width `Parma_Polyhedra_Library::Concrete_Expression_Type::bounded_integer_type_width () const [inline]` Returns the width in bits of the bounded integer type encoded by `*this`.

The behavior is undefined if `*this` does not encode a bounded integer type.

Bounded_Integer_Type_Representation `Parma_Polyhedra_Library::Concrete_Expression_Type::bounded_integer_type_representation () const [inline]` Returns the representation of the bounded integer type encoded by `*this`.

The behavior is undefined if `*this` does not encode a bounded integer type.

Bounded_Integer_Type_Overflow `Parma_Polyhedra_Library::Concrete_Expression_Type::bounded_integer_type_overflow () const [inline]` Returns the overflow behavior of the bounded integer type encoded by `*this`.

The behavior is undefined if `*this` does not encode a bounded integer type.

Floating_Point_Format `Parma_Polyhedra_Library::Concrete_Expression_Type::floating_point_format () const [inline]` Returns the format of the floating point type encoded by `*this`.

The behavior is undefined if `*this` does not encode a floating point type.

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

- `ppl.hh`

10.21 Parma_Polyhedra_Library::Congruence Class Reference

A linear congruence.

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

Public Types

- typedef
`Expression_Adapter_Transparent`
< `Linear_Expression` > `expr_type`

The type of the (adapted) internal expression.

Public Member Functions

- [Congruence](#) ([Representation](#) r=[default_representation](#))
Constructs the $0 = 0$ congruence with space dimension 0.
- [Congruence](#) (const [Congruence](#) &cg)
Ordinary copy constructor.
- [Congruence](#) (const [Congruence](#) &c, [Representation](#) r)
Copy constructor with specified representation.
- [Congruence](#) (const [Constraint](#) &c, [Representation](#) r=[default_representation](#))
Copy-constructs (modulo 0) from equality constraint c.
- [~Congruence](#) ()
Destructor.
- [Congruence](#) & [operator=](#) (const [Congruence](#) &y)
Assignment operator.
- [Representation](#) [representation](#) () const
*Returns the current representation of *this.*
- void [set_representation](#) ([Representation](#) r)
*Converts *this to the specified representation.*
- [dimension_type](#) [space_dimension](#) () const
*Returns the dimension of the vector space enclosing *this.*
- [expr_type](#) [expression](#) () const
Partial read access to the (adapted) internal expression.
- [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.*
- void [set_modulus](#) ([Coefficient_traits::const_reference](#) m)
- void [scale](#) ([Coefficient_traits::const_reference](#) factor)
Multiplies all the coefficients, including the modulus, by factor.
- [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.*
- [memory_size_type](#) [total_memory_in_bytes](#) () const
*Returns a lower bound to the total size in bytes of the memory occupied by *this.*
- [memory_size_type](#) [external_memory_in_bytes](#) () const
*Returns the size in bytes of the memory managed by *this.*
- bool [OK](#) () const
Checks if all the invariants are satisfied.

- void `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`.*
- void `m.swap (Congruence &y)`
*Swaps `*this` with `y`.*
- `Congruence` (const `Congruence` &cg, `dimension_type` new_space_dimension)
Copy-constructs with the specified space dimension.
- `Congruence` (const `Congruence` &cg, `dimension_type` new_space_dimension, `Representation` r)
Copy-constructs with the specified space dimension and representation.
- `Congruence` (const `Constraint` &cg, `dimension_type` new_space_dimension, `Representation` r=default_representation)
Copy-constructs with the specified space dimension and representation.
- `Congruence` (`Linear_Expression` &le, `Coefficient_traits::const_reference` m, `Recycle_Input`)
Constructs from `Linear_Expression` `le`, using modulus `m`.
- void `swap_space_dimensions (Variable v1, Variable v2)`
Swaps the coefficients of the variables `v1` and `v2`.
- void `set_space_dimension (dimension_type n)`
Set the space dimension to `n`.
- void `shift_space_dimensions (Variable v, dimension_type n)`
Shift the space dimension of the variable `v` by `n`.
- void `sign_normalize ()`
Normalizes the signs.
- void `normalize ()`
Normalizes signs and the inhomogeneous term.
- void `strong_normalize ()`
Calls `normalize`, then divides out common factors.

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, `Representation` r=default_representation)
Returns the congruence $e1 = e2 \pmod{1}$.
- static `Congruence create` (const `Linear_Expression` &e, `Coefficient_traits::const_reference` n, `Representation` r=default_representation)
Returns the congruence $e = n \pmod{1}$.

- static `Congruence create` (Coefficient_traits::const_reference n, const `Linear_Expression` &e, `Representation r=default_representation`)

Returns the congruence $n = e \pmod{1}$.

Static Public Attributes

- static const `Representation default_representation = SPARSE`

The representation used for new Congruences.

Related Functions

(Note that these are not member functions.)

- 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.
- 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 $e = n \pmod{1}$.
- `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.
- void `swap` (`Congruence` &x, `Congruence` &y)
- `Congruence operator%=` (const `Linear_Expression` &e1, const `Linear_Expression` &e2)
- `Congruence operator%=` (const `Linear_Expression` &e, Coefficient_traits::const_reference n)
- `Congruence operator/` (const `Congruence` &cg, Coefficient_traits::const_reference k)
- `Congruence operator/` (const `Constraint` &c, Coefficient_traits::const_reference m)
- bool `operator==` (const `Congruence` &x, const `Congruence` &y)
- bool `operator!=` (const `Congruence` &x, const `Congruence` &y)
- void `swap` (`Congruence` &x, `Congruence` &y)

10.21.1 Detailed Description

A linear congruence.

An object of the class `Congruence` is a congruence:

$$\text{cg} = \sum_{i=0}^{n-1} a_i x_i + b = 0 \pmod{m}$$

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.

10.21.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Congruence::Congruence (const Congruence & cg) [inline] Ordinary copy constructor.

Note

The new [Congruence](#) will have the same representation as ‘cg’, not default_representation, so that they are indistinguishable.

Parma_Polyhedra_Library::Congruence::Congruence (const Constraint & c, Representation r = default_representation) [explicit] Copy-constructs (modulo 0) from equality constraint c.

Exceptions

<i>std::invalid_argument</i>	Thrown if c is an inequality.
------------------------------	-------------------------------

Parma_Polyhedra_Library::Congruence::Congruence (const Congruence & cg, dimension_type new_space_dimension) [inline] Copy-constructs with the specified space dimension.

Note

The new [Congruence](#) will have the same representation as ‘cg’, not default_representation, for consistency with the copy constructor.

Parma_Polyhedra_Library::Congruence::Congruence (const Constraint & cg, dimension_type new_space_dimension, Representation r = default_representation) Copy-constructs from a constraint, with the specified space dimension and (optional) representation.

Parma_Polyhedra_Library::Congruence::Congruence (Linear_Expression & le, Coefficient_traits::const_reference m, Recycle_Input) [inline] Constructs from [Linear_Expression](#) le, using modulus m.

Builds a congruence with modulus m, stealing the coefficients from le.

Note

The new [Congruence](#) will have the same representation as ‘le’.

Parameters

<i>le</i>	The Linear_Expression holding the coefficients.
<i>m</i>	The modulus for the congruence, which must be zero or greater.

10.21.3 Member Function Documentation

Coefficient_traits::const_reference Parma_Polyhedra_Library::Congruence::coefficient (Variable v) const [inline] Returns the coefficient of v in *this.

Exceptions

<i>std::invalid_argument</i>	thrown if the index of v is greater than or equal to the space dimension of *this.
------------------------------	--

void Parma_Polyhedra_Library::Congruence::set_modulus (Coefficient_traits::const_reference m) [inline] Sets the modulus of *this to m. If m is 0, the congruence becomes an equality.

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 $e_1 = e_2 \pmod{mk}$.

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 $b = 0 \pmod{m}$.

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 $b \neq 0 \pmod{m}$.

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.

bool Parma_Polyhedra_Library::Congruence::is_equality () const [inline] Returns `true` if $*this$ is an equality.

A modulus of zero denotes a linear equality.

void Parma_Polyhedra_Library::Congruence::set_space_dimension (dimension_type n) [inline] Sets the space dimension by n , adding or removing coefficients as needed.

void Parma_Polyhedra_Library::Congruence::shift_space_dimensions (Variable v, dimension_type n) [inline] Shift by n positions the coefficients of variables, starting from the coefficient of v . This increases the space dimension by n .

void Parma_Polyhedra_Library::Congruence::sign_normalize () Normalizes the signs.

The signs of the coefficients and the inhomogeneous term are normalized, leaving the first non-zero homogeneous coefficient positive.

void Parma_Polyhedra_Library::Congruence::normalize () Normalizes signs and the inhomogeneous term.

Applies `sign_normalize`, then reduces the inhomogeneous term to the smallest possible positive number.

void Parma_Polyhedra_Library::Congruence::strong_normalize () Calls `normalize`, then divides out common factors.

Strongly normalized Congruences have equivalent semantics if and only if they have the same syntax (as output by `operator<<`).

10.21.4 Friends And Related Function Documentation

bool operator==(const Congruence & x, const Congruence & y) [related] Returns `true` if and only if `x` and `y` are equivalent.

bool operator!=(const Congruence & x, const Congruence & y) [related] Returns `false` if and only if `x` and `y` are equivalent.

std::ostream & operator<< (std::ostream & s, const Congruence & c) [related] Output operators.

Congruence operator%=(const Linear_Expression & e1, const Linear_Expression & e2) [related]
Returns the congruence $e_1 = e_2 \pmod{1}$.

Congruence operator%=(const Linear_Expression & e, Coefficient_traits::const_reference n) [related]
Returns the congruence $e = n \pmod{1}$.

Congruence operator/(const Congruence & cg, Coefficient_traits::const_reference k) [related]
Returns a copy of `cg`, multiplying `k` into the copy's modulus.

If `cg` represents the congruence $e_1 = e_2 \pmod{m}$, then the result represents the congruence $e_1 = e_2 \pmod{mk}$.

Congruence operator/(const Constraint & c, Coefficient_traits::const_reference m) [related]
Creates a congruence from `c`, with `m` as the modulus.

void swap (Congruence & x, Congruence & y) [related]

Congruence operator%=(const Linear_Expression & e1, const Linear_Expression & e2) [related]

Congruence operator%=(const Linear_Expression & e, Coefficient_traits::const_reference n) [related]

Congruence operator/(const Congruence & cg, Coefficient_traits::const_reference k) [related]

Congruence operator/(const Constraint & c, Coefficient_traits::const_reference m) [related]

bool operator==(const Congruence & x, const Congruence & y) [related]

bool operator!=(const Congruence & x, const Congruence & y) [related]

void swap (Congruence & x, Congruence & y) [related]

10.21.5 Member Data Documentation

const Representation Parma_Polyhedra_Library::Congruence::default_representation = SPARSE [static]
The representation used for new Congruences.

Note

The copy constructor and the copy constructor with specified size use the representation of the original object, so that it is indistinguishable from the original object.

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

- `ppl.hh`

10.22 Parma Polyhedra Library::Congruence_System Class Reference

A system of congruences.

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

Classes

- class [const_iterator](#)

An iterator over a system of congruences.

Public Member Functions

- [Congruence_System](#) ([Representation](#) r=default_representation)
Default constructor: builds an empty system of congruences.
- [Congruence_System](#) ([dimension_type](#) d, [Representation](#) r=default_representation)
Builds an empty (i.e. zero rows) system of dimension d.
- [Congruence_System](#) (const [Congruence](#) &cg, [Representation](#) r=default_representation)
Builds the singleton system containing only congruence cg.
- [Congruence_System](#) (const [Constraint](#) &c, [Representation](#) r=default_representation)
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, [Representation](#) r=default_representation)
Builds a system containing copies of any equalities in cs.
- [Congruence_System](#) (const [Congruence_System](#) &cgs)
Ordinary copy constructor.
- [Congruence_System](#) (const [Congruence_System](#) &cgs, [Representation](#) r)
Copy constructor with specified representation.
- [~Congruence_System](#) ()
Destructor.
- [Congruence_System](#) & operator= (const [Congruence_System](#) &y)
Assignment operator.
- [Representation](#) representation () const
*Returns the current representation of *this.*
- void [set_representation](#) ([Representation](#) r)
*Converts *this to the specified representation.*
- [dimension_type](#) space_dimension () const
*Returns the dimension of the vector space enclosing *this.*
- bool [is_equal_to](#) (const [Congruence_System](#) &y) const
*Returns true if and only if *this is exactly equal to y.*
- 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](#) ([Congruence](#) &cg, [Recycle_Input](#))
*Inserts in *this the congruence cg, stealing its contents and 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` &y)

*Inserts in `*this` a copy of the congruences in `y`, increasing the number of space dimensions if needed.*
 - void `insert` (`Congruence_System` &cgs, `Recycle_Input`)

*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 `m_swap` (`Congruence_System` &y)

*Swaps `*this` with `y`.*
 - void `add_unit_rows_and_space_dimensions` (`dimension_type` dims)

Adds `dims` rows and `dims` space dimensions to the matrix, initializing the added rows as in the unit congruence system.
 - void `permute_space_dimensions` (const std::vector< `Variable` > &cycle)

Permutes the space dimensions of the system.
 - void `swap_space_dimensions` (`Variable` v1, `Variable` v2)

Swaps the columns having indexes `i` and `j`.
 - bool `set_space_dimension` (`dimension_type` new_space_dim)

Sets the number of space dimensions to `new_space_dim`.

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.)

- bool `operator== (const Congruence_System &x, const Congruence_System &y)`
- `std::ostream & operator<< (std::ostream &s, const Congruence_System &cgs)`
Output operator.
- void `swap (Congruence_System &x, Congruence_System &y)`
- void `swap (Congruence_System &x, Congruence_System &y)`

10.22.1 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.

10.22.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Congruence_System::Congruence_System (const Constraint & c, Representation r = default_representation) [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

<code>std::invalid_argument</code>	Thrown if c is not an equality constraint.
------------------------------------	--

Parma_Polyhedra_Library::Congruence_System::Congruence_System (const Congruence_System & cgs) [inline] Ordinary copy constructor.

Note

The new `Congruence_System` will have the same Representation as ‘cgs’ so that it’s indistinguishable from ‘cgs’.

10.22.3 Member Function Documentation

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.

void Parma_Polyhedra_Library::Congruence_System::insert (Congruence & cg, Recycle_Input) [inline]

Inserts in `*this` the congruence `cg`, stealing its contents and increasing the number of space dimensions if needed.

`cg` will be strongly normalized.

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

<code>std::invalid_argument</code>	Thrown if c is a relational constraint.
------------------------------------	---

void Parma_Polyhedra_Library::Congruence_System::insert (const Congruence_System & y) Inserts in `*this` a copy of the congruences in `y`, increasing the number of space dimensions if needed.

The inserted copies will be strongly normalized.

void Parma_Polyhedra_Library::Congruence_System::add_unit_rows_and_space_dimensions (dimension_type dims) Adds *dims* rows and *dims* space dimensions to the matrix, initializing the added rows as in the unit congruence system.

Parameters

<i>dims</i>	The number of rows and space dimensions to be added: must be strictly positive.
-------------	---

Turns the $r \times c$ matrix A into the $(r + \text{dims}) \times (c + \text{dims})$ matrix $\begin{pmatrix} 0 & B \\ A & A \end{pmatrix}$ where B is the $\text{dims} \times \text{dims}$ unit matrix of the form $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The matrix is expanded avoiding reallocation whenever possible.

void Parma_Polyhedra_Library::Congruence_System::permute_space_dimensions (const std::vector< Variable > & cycle) Permutes the space dimensions of the system.

Parameters

<i>cycle</i>	A vector representing a cycle of the permutation according to which the columns must be rearranged.
--------------	---

The *cycle* vector represents a cycle of a permutation of space dimensions. For example, the permutation $\{x_1 \mapsto x_2, x_2 \mapsto x_3, x_3 \mapsto x_1\}$ can be represented by the vector containing x_1, x_2, x_3 .

bool Parma_Polyhedra_Library::Congruence_System::set_space_dimension (dimension_type new_space_dim) Sets the number of space dimensions to *new_space_dim*.

If *new_space_dim* is lower than the current space dimension, the coefficients referring to the removed space dimensions are lost.

10.22.4 Friends And Related Function Documentation

bool operator== (const Congruence_System & x, const Congruence_System & y) [**related**]

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 `”, ”`.

void swap (Congruence_System & x, Congruence_System & y) [**related**]

void swap (Congruence_System & x, Congruence_System & y) [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.23 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.

10.23.1 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.

10.23.2 Member Function Documentation

template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Congruences_Reduction<D1, D2 >::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.

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

<i>d1</i>	A pointset domain element;
<i>d2</i>	A pointset domain element;

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

- ppl.hh

10.24 Parma_Polyhedra_Library::CO_Tree::const_iterator Class Reference

A const iterator on the tree elements, ordered by key.

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

Public Member Functions

- [const_iterator \(\)](#)
Constructs an invalid [const_iterator](#).
- [const_iterator \(const CO_Tree &tree\)](#)
Constructs an iterator pointing to the first element of the tree.
- [const_iterator \(const CO_Tree &tree, \[dimension_type\]\(#\) i\)](#)
Constructs a [const_iterator](#) pointing to the i-th node of the tree.
- [const_iterator \(const \[const_iterator\]\(#\) &itr\)](#)

- The copy constructor.*

 - `const_iterator` (const `iterator` &itr)

Converts an iterator into a `const_iterator`.
- void `m_swap` (const `iterator` &itr)

*Swaps itr with *this.*
- `const_iterator` & `operator=` (const `const_iterator` &itr)

*Assigns itr to *this.*
- `const_iterator` & `operator=` (const `iterator` &itr)

*Assigns itr to *this.*
- `const_iterator` & `operator++` ()

Navigates to the next element.
- `const_iterator` & `operator--` ()

Navigates to the previous element.
- `const_iterator` `operator++` (int)

Navigates to the next element.
- `const_iterator` `operator--` (int)

Navigates to the previous element.
- data_type_const_reference `operator*` () const

Returns the current element.
- dimension_type `index` () const

*Returns the index of the element pointed to by *this.*
- bool `operator==` (const `const_iterator` &x) const

*Compares *this with x.*
- bool `operator!=` (const `const_iterator` &x) const

*Compares *this with x.*

10.24.1 Detailed Description

A const iterator on the tree elements, ordered by key.

Iterator increment and decrement operations are $O(1)$ time. These iterators are invalidated by operations that add or remove elements from the tree.

10.24.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::CO_Tree::const_iterator::const_iterator () [inline], [explicit]

Constructs an invalid `const_iterator`.

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::const_iterator::const_iterator (const CO_Tree &tree) [inline], [explicit] Constructs an iterator pointing to the first element of the tree.

Parameters

<i>tree</i>	The tree that the new iterator will point to.
-------------	---

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::const_iterator::const_iterator (const CO_Tree &tree, dimension_type i) [inline] Constructs a `const_iterator` pointing to the i-th node of the tree.

Parameters

<i>tree</i>	The tree that the new iterator will point to.
<i>i</i>	The index of the element in <i>tree</i> to which the iterator will point to.

The *i*-th node must be a node with a value or end().

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::const_iterator::const_iterator (const const_iterator & itr)
[inline] The copy constructor.

Parameters

<i>itr</i>	The iterator that will be copied.
------------	-----------------------------------

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::const_iterator::const_iterator (const iterator & itr) **[inline]**
Converts an iterator into a [const_iterator](#).

Parameters

<i>itr</i>	The iterator that will be converted into a const_iterator .
------------	---

This constructor takes $O(1)$ time.

10.24.3 Member Function Documentation

void Parma_Polyhedra_Library::CO_Tree::const_iterator::m_swap (const_iterator & itr) **[inline]**
Swaps *itr* with **this*.

Parameters

<i>itr</i>	The iterator that will be swapped with <i>*this</i> .
------------	---

This method takes $O(1)$ time.

CO_Tree::const_iterator & Parma_Polyhedra_Library::CO_Tree::const_iterator::operator= (const const_iterator & itr) **[inline]** Assigns *itr* to **this*.

Parameters

<i>itr</i>	The iterator that will be assigned into <i>*this</i> .
------------	--

This method takes $O(1)$ time.

CO_Tree::const_iterator & Parma_Polyhedra_Library::CO_Tree::const_iterator::operator= (const iterator & itr) **[inline]** Assigns *itr* to **this*.

Parameters

<i>itr</i>	The iterator that will be assigned into <i>*this</i> .
------------	--

This method takes $O(1)$ time.

CO_Tree::const_iterator & Parma_Polyhedra_Library::CO_Tree::const_iterator::operator++ ()
[inline] Navigates to the next element.

This method takes $O(1)$ time.

CO_Tree::const_iterator & Parma_Polyhedra_Library::CO_Tree::const_iterator::operator-- () **[inline]**
Navigates to the previous element.

This method takes $O(1)$ time.

CO_Tree::const_iterator Parma_Polyhedra_Library::CO_Tree::const_iterator::operator++ (int)
[inline] Navigates to the next element.

This method takes $O(1)$ time.

CO_Tree::const_iterator Parma_Polyhedra_Library::CO_Tree::const_iterator::operator-- (int)
[inline] Navigates to the previous element.

This method takes $O(1)$ time.

dimension_type Parma_Polyhedra_Library::CO_Tree::const_iterator::index () const [inline]
Returns the index of the element pointed to by **this*.

Returns

the index of the element pointed to by **this*.

bool Parma_Polyhedra_Library::CO_Tree::const_iterator::operator== (const const_iterator & x)
const [inline] Compares **this* with *x*.

Parameters

<i>x</i>	The iterator that will be compared with <i>*this</i> .
----------	--

bool Parma_Polyhedra_Library::CO_Tree::const_iterator::operator!= (const const_iterator & x)
const [inline] Compares **this* with *x*.

Parameters

<i>x</i>	The iterator that will be compared with <i>*this</i> .
----------	--

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

- ppl.hh

10.25 Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator Class Reference

#include <ppl.hh>

Inherits const_iterator_interface.

Public Member Functions

- virtual const_iterator_interface * **clone** () const
- virtual void **operator++** ()
- virtual void **operator--** ()
- virtual reference **operator*** () const

Returns the current element.

- virtual **Variable variable** () const

*Returns the variable of the coefficient pointed to by *this.*

- virtual bool **operator==** (const const_iterator_interface &x) const

*Compares *this with x.*

10.25.1 Detailed Description

template<typename Row>class Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator

An interface for const iterators on the expression (homogeneous) coefficients that are nonzero.

These iterators are invalidated by operations that modify the expression.

10.25.2 Member Function Documentation

template<typename Row > Linear_Expression_Interface::const_iterator_interface * Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator::clone () const [virtual] Returns a copy of **this*. This returns a pointer to dynamic-allocated memory. The caller has the duty to free the memory when it's not needed anymore.

template<typename Row > void Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator::operator++ () [virtual] Navigates to the next nonzero coefficient. Note that this method does **not** return a reference, to increase efficiency since it's virtual.

template<typename Row > void Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator::operator-- () [virtual] Navigates to the previous nonzero coefficient. Note that this method does **not** return a reference, to increase efficiency since it's virtual.

template<typename Row > Variable Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator::variable () const [virtual] Returns the variable of the coefficient pointed to by **this*.

Returns

the variable of the coefficient pointed to by **this*.

template<typename Row > bool Parma_Polyhedra_Library::Linear_Expression_Impl< Row >::const_iterator::operator== (const const_iterator_interface & x) const [virtual] Compares **this* with *x*.
Parameters

<i>x</i>	The iterator that will be compared with <i>*this</i> .
----------	--

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

- ppl.hh

10.26 Parma_Polyhedra_Library::Linear_Expression::const_iterator Class Reference

#include <ppl.hh>

Public Member Functions

- [const_iterator](#) ()
Constructs an invalid [const_iterator](#).
- [const_iterator](#) (const [const_iterator](#) &itr)
The copy constructor.
- void [m_swap](#) (const [const_iterator](#) &itr)
*Swaps itr with **this*.*

- `const_iterator & operator= (const const_iterator &itr)`
*Assigns `itr` to `*this`.*
- `const_iterator & operator++ ()`
Navigates to the next nonzero coefficient.
- `const_iterator & operator-- ()`
Navigates to the previous nonzero coefficient.
- `reference operator* () const`
Returns the current element.
- `Variable variable () const`
*Returns the variable of the coefficient pointed to by `*this`.*
- `bool operator== (const const_iterator &x) const`
*Compares `*this` with `x`.*
- `bool operator!= (const const_iterator &x) const`
*Compares `*this` with `x`.*

Related Functions

(Note that these are not member functions.)

- `void swap (Linear_Expression::const_iterator &x, Linear_Expression::const_iterator &y)`
Swaps `x` with `y`.
- `void swap (Linear_Expression::const_iterator &x, Linear_Expression::const_iterator &y)`

10.26.1 Detailed Description

A const iterator on the expression (homogeneous) coefficient that are nonzero.
These iterators are invalidated by operations that modify the expression.

10.26.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Linear_Expression::const_iterator::const_iterator () [inline], [explicit]

Constructs an invalid `const_iterator`.

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::Linear_Expression::const_iterator::const_iterator (const const_iterator &itr) [inline] The copy constructor.

Parameters

<i>itr</i>	The iterator that will be copied.
------------	-----------------------------------

This constructor takes $O(1)$ time.

10.26.3 Member Function Documentation

void Parma_Polyhedra_Library::Linear_Expression::const_iterator::m_swap (const_iterator &itr) [inline] Swaps `itr` with `*this`.

Parameters

<i>itr</i>	The iterator that will be swapped with <code>*this</code> .
------------	---

This method takes $O(1)$ time.

Linear_Expression::const_iterator & Parma_Polyhedra_Library::Linear_Expression::const_iterator::operator= (const const_iterator &itr) [inline] Assigns `itr` to `*this`.

Parameters

<i>itr</i>	The iterator that will be assigned into *this.
------------	--

This method takes $O(1)$ time.

Linear_Expression::const_iterator & Parma_Polyhedra_Library::Linear_Expression::const_iterator::operator++ () [inline] Navigates to the next nonzero coefficient.

This method takes $O(n)$ time for dense expressions, and $O(1)$ time for sparse expressions.

Linear_Expression::const_iterator & Parma_Polyhedra_Library::Linear_Expression::const_iterator::operator-- () [inline] Navigates to the previous nonzero coefficient.

This method takes $O(n)$ time for dense expressions, and $O(1)$ time for sparse expressions.

Variable Parma_Polyhedra_Library::Linear_Expression::const_iterator::variable () const [inline]

Returns the variable of the coefficient pointed to by *this.

Returns

the variable of the coefficient pointed to by *this.

bool Parma_Polyhedra_Library::Linear_Expression::const_iterator::operator== (const const_iterator & x) const [inline] Compares *this with x .

Parameters

<i>x</i>	The iterator that will be compared with *this.
----------	--

bool Parma_Polyhedra_Library::Linear_Expression::const_iterator::operator!= (const const_iterator & x) const [inline] Compares *this with x .

Parameters

<i>x</i>	The iterator that will be compared with *this.
----------	--

10.26.4 Friends And Related Function Documentation

void swap (Linear_Expression::const_iterator & x, Linear_Expression::const_iterator & y) [related]

Swaps x with y.

void swap (Linear_Expression::const_iterator & x, Linear_Expression::const_iterator & y) [related]

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

- ppl.hh

10.27 Parma_Polyhedra_Library::Congruence_System::const_iterator Class Reference

An iterator over a system of congruences.

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

Inherits iterator< std::forward_iterator_tag, Congruence, std::ptrdiff_t, const Congruence *, const Congruence & >.

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.*

10.27.1 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`

10.28 Parma_Polyhedra_Library::MIP_Problem::const_iterator Class Reference

A read-only iterator on the constraints defining the feasible region.

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

Public Member Functions

- `difference_type operator- (const const_iterator &y) const`
Iterator difference: computes distances.
- `const_iterator & operator++ ()`
Prefix increment.
- `const_iterator & operator-- ()`

- Prefix decrement.*
- `const_iterator operator++` (int)
- Postfix increment.*
- `const_iterator operator--` (int)
- Postfix decrement.*
- `const_iterator & operator+=` (difference_type n)
- Moves iterator forward of n positions.*
- `const_iterator & operator-=` (difference_type n)
- Moves iterator backward of n positions.*
- `const_iterator operator+` (difference_type n) const
- Returns an iterator n positions forward.*
- `const_iterator operator-` (difference_type n) const
- Returns an iterator n positions backward.*
- `reference operator*` () const
- Returns a reference to the "pointed" object.*
- `pointer operator->` () const
- Returns the address of the "pointed" object.*
- `bool operator==` (const `const_iterator` &y) const
- Compares *this with y.*
- `bool operator!=` (const `const_iterator` &y) const
- Compares *this with y.*

10.28.1 Detailed Description

A read-only iterator on the constraints defining the feasible region.

10.28.2 Member Function Documentation

bool Parma_Polyhedra_Library::MIP_Problem::const_iterator::operator== (const `const_iterator` &y) const **[inline]** Compares *this with y.
Parameters

y	The iterator that will be compared with *this.
---	--

bool Parma_Polyhedra_Library::MIP_Problem::const_iterator::operator!= (const `const_iterator` &y) const **[inline]** Compares *this with y.
Parameters

y	The iterator that will be compared with *this.
---	--

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

- ppl.hh

10.29 Parma_Polyhedra_Library::Grid_Generator_System::const_iterator Class Reference

An iterator over a system of grid generators.

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

Inherits `iterator< std::forward_iterator_tag, Grid_Generator, std::ptrdiff_t, const Grid_Generator *, const Grid_Generator & >`.

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.*

10.29.1 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`

10.30 Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator interface Class Reference

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

Public Member Functions

- virtual `const_iterator_interface * clone ()` const =0
- virtual void `operator++ ()`=0
- virtual void `operator-- ()`=0
- virtual reference `operator* ()` const =0
Returns the current element.
- virtual `Variable variable ()` const =0
*Returns the variable of the coefficient pointed to by *this.*
- virtual bool `operator== (const const_iterator_interface &x)` const =0
*Compares *this with x .*

10.30.1 Detailed Description

An interface for const iterators on the expression (homogeneous) coefficients that are nonzero.
These iterators are invalidated by operations that modify the expression.

10.30.2 Member Function Documentation

virtual const_iterator_interface* Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface::clone () const [pure virtual] Returns a copy of *this. This returns a pointer to dynamic-allocated memory. The caller has the duty to free the memory when it's not needed anymore.

virtual void Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface::operator++ () [pure virtual] Navigates to the next nonzero coefficient. Note that this method does **not** return a reference, to increase efficiency since it's virtual.

virtual void Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface::operator-- () [pure virtual] Navigates to the previous nonzero coefficient. Note that this method does **not** return a reference, to increase efficiency since it's virtual.

virtual Variable Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface::variable () const [pure virtual] Returns the variable of the coefficient pointed to by *this.

Returns

the variable of the coefficient pointed to by *this.

virtual bool Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface::operator== (const const_iterator_interface &x) const [pure virtual] Compares *this with x .

Parameters

<i>x</i>	The iterator that will be compared with *this.
----------	--

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

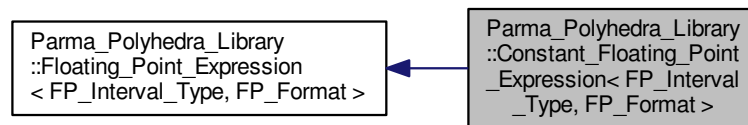
- ppl.hh

10.31 Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic Constant Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form](#) [FP_Linear_Form](#)
Alias for the Linear_Form<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Interval_Abstract_Store](#) [FP_Interval_Abstract_Store](#)
Alias for the Box<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form_Abstract_Store](#) [FP_Linear_Form_Abstract_Store](#)
Alias for the std::map<dimension_type, FP_Linear_Form> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[boundary_type](#) [boundary_type](#)
Alias for the FP_Interval_Type::boundary_type from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[info_type](#) [info_type](#)
Alias for the FP_Interval_Type::info_type from [Floating_Point_Expression](#).

Public Member Functions

- bool [linearize](#) (const [FP_Interval_Abstract_Store](#) &int_store, const [FP_Linear_Form_Abstract_Store](#) &lf_store, [FP_Linear_Form](#) &result) const
Linearizes the expression in a given astract store.

- void `m_swap (Constant_Floating_Point_Expression &y)`
*Swaps `*this` with `y`.*

Constructors and Destructor

- `Constant_Floating_Point_Expression` (const `boundary_type` `lower_bound`, const `boundary_type` `upper_bound`)
Constructor with two parameters: builds the constant floating point expression from a `lower_bound` and an `upper_bound` of its value in the concrete domain.
- `Constant_Floating_Point_Expression` (const char `*str_value`)
Builds a constant floating point expression with the value expressed by the string `str_value`.
- `~Constant_Floating_Point_Expression ()`
Destructor.

Related Functions

(Note that these are not member functions.)

- template<typename `FP_Interval_Type` , typename `FP_Format` >
void `swap (Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`
Swaps `x` with `y`.
- template<typename `FP_Interval_Type` , typename `FP_Format` >
void `swap (Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`

Additional Inherited Members

10.31.1 Detailed Description

template<typename `FP_Interval_Type`, typename `FP_Format`>class Parma_Polyhedra_Library::Constant_Floating_Point_Expression< `FP_Interval_Type`, `FP_Format` >

A generic Constant Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of floating-point constant expressions

The linearization of a constant floating point expression results in a linear form consisting of only the inhomogeneous term $[l, u]$, where l and u are the lower and upper bounds of the constant value given to the class constructor.

10.31.2 Member Function Documentation

template<typename `FP_Interval_Type` , typename `FP_Format` > bool Parma_Polyhedra_Library::Constant_Floating_Point_Expression< `FP_Interval_Type`, `FP_Format` >::linearize (const `FP_Interval_Abstract_Store` & `int_store`, const `FP_Linear_Form_Abstract_Store` & `lf_store`, `FP_Linear_Form` & `result`) const [`inline`], [`virtual`] Linearizes the expression in a given abstract store.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

See the class description for an explanation of how `result` is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.31.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y) [**related**] Swaps `x` with `y`.

template<typename FP_Interval_Type , typename FP_Format > void swap (Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y) [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.32 Parma_Polyhedra_Library::Constraint Class Reference

A linear equality or inequality.

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

Public Types

- enum [Type](#) { [EQUALITY](#), [NONSTRICT_INEQUALITY](#), [STRICT_INEQUALITY](#) }
The constraint type.
- typedef `Expression_Hide_Last`
`< Linear_Expression > expr_type`
The type of the (adapted) internal expression.

Public Member Functions

- [Constraint](#) ([Representation](#) `r=default_representation`)
Constructs the $0 \leq 0$ constraint.
- [Constraint](#) (const [Constraint](#) &c)
Ordinary copy constructor.
- [Constraint](#) (const [Constraint](#) &c, [dimension_type](#) `space_dim`)
Copy constructor with given size.
- [Constraint](#) (const [Constraint](#) &c, [Representation](#) `r`)
Copy constructor with given representation.
- [Constraint](#) (const [Constraint](#) &c, [dimension_type](#) `space_dim`, [Representation](#) `r`)
Copy constructor with given size and representation.
- [Constraint](#) (const [Congruence](#) &cg, [Representation](#) `r=default_representation`)
Copy-constructs from equality congruence `cg`.

- `~Constraint ()`
Destructor.
- `Representation representation () const`
*Returns the current representation of *this.*
- `void set_representation (Representation r)`
*Converts *this to the specified representation.*
- `Constraint & operator= (const Constraint &c)`
Assignment operator.
- `dimension_type space_dimension () const`
*Returns the dimension of the vector space enclosing *this.*
- `void set_space_dimension (dimension_type space_dim)`
- `void swap_space_dimensions (Variable v1, Variable v2)`
Swaps the coefficients of the variables v1 and v2.
- `bool remove_space_dimensions (const Variables_Set &vars)`
Removes all the specified dimensions from the constraint.
- `void permute_space_dimensions (const std::vector< Variable > &cycle)`
Permutes the space dimensions of the constraint.
- `void shift_space_dimensions (Variable v, dimension_type n)`
- `Type type () const`
*Returns the constraint type of *this.*
- `bool is_equality () const`
*Returns true if and only if *this is an equality constraint.*
- `bool is_inequality () const`
*Returns true if and only if *this is an inequality constraint (either strict or non-strict).*
- `bool is_nonstrict_inequality () const`
*Returns true if and only if *this is a non-strict inequality constraint.*
- `bool is_strict_inequality () const`
*Returns true if and only if *this is a strict inequality constraint.*
- `Coefficient_traits::const_reference coefficient (Variable v) const`
*Returns the coefficient of v in *this.*
- `Coefficient_traits::const_reference inhomogeneous_term () const`
*Returns the inhomogeneous term of *this.*
- `memory_size_type total_memory_in_bytes () const`
*Returns a lower bound to the total size in bytes of the memory occupied by *this.*
- `memory_size_type external_memory_in_bytes () const`
*Returns the size in bytes of the memory managed by *this.*
- `bool is_tautological () const`
*Returns true if and only if *this is a tautology (i.e., an always true constraint).*
- `bool is_inconsistent () const`
*Returns true if and only if *this is inconsistent (i.e., an always false constraint).*
- `bool is_equivalent_to (const Constraint &y) const`
*Returns true if and only if *this and y are equivalent constraints.*
- `bool is_equal_to (const Constraint &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.*
- void `m_swap` (`Constraint &y`)
*Swaps `*this` with `y`.*
- `expr_type expression` () const
Partial read access to the (adapted) internal expression.

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.
- static const `Constraint & epsilon_geq_zero` ()
Returns the zero-dimension space constraint $\epsilon \geq 0$.
- static const `Constraint & epsilon_leq_one` ()
The zero-dimension space constraint $\epsilon \leq 1$ (used to implement NNC polyhedra).

Static Public Attributes

- static const `Representation default_representation = SPARSE`
The representation used for new Constraints.

Related Functions

(Note that these are not member functions.)

- `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==** (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 $e1 \leq e2$.
- **Constraint operator<=** (Variable v1, Variable v2)
Returns the constraint $v1 \leq v2$.
- **Constraint operator<=** (const Linear_Expression &e, Coefficient_traits::const_reference n)
Returns the constraint $e \leq n$.
- **Constraint operator<=** (Coefficient_traits::const_reference n, const Linear_Expression &e)
Returns the constraint $n \leq e$.
- **Constraint operator>=** (const Linear_Expression &e1, const Linear_Expression &e2)
Returns the constraint $e1 \geq e2$.
- **Constraint operator>=** (Variable v1, Variable v2)
Returns the constraint $v1 \geq v2$.
- **Constraint operator>=** (const Linear_Expression &e, Coefficient_traits::const_reference n)
Returns the constraint $e \geq n$.
- **Constraint operator>=** (Coefficient_traits::const_reference n, const Linear_Expression &e)
Returns the constraint $n \geq e$.
- **std::ostream & operator<<** (std::ostream &s, const Constraint &c)
Output operator.
- **std::ostream & operator<<** (std::ostream &s, const Constraint::Type &t)
Output operator.
- **bool operator==** (const Constraint &x, const Constraint &y)
Returns `true` if and only if x is equivalent to y .
- **bool operator!=** (const Constraint &x, const Constraint &y)
Returns `true` if and only if x is not equivalent to y .
- **void swap** (Constraint &x, Constraint &y)
- **bool operator==** (const Constraint &x, const Constraint &y)
- **bool operator!=** (const Constraint &x, const Constraint &y)
- **Constraint operator==** (const Linear_Expression &e1, const Linear_Expression &e2)
- **Constraint operator==** (Variable v1, Variable v2)
- **Constraint operator>=** (const Linear_Expression &e1, const Linear_Expression &e2)
- **Constraint operator>=** (const Variable v1, const Variable v2)
- **Constraint operator>** (const Linear_Expression &e1, const Linear_Expression &e2)
- **Constraint operator>** (const Variable v1, const Variable v2)

- `Constraint operator==` (Coefficient_traits::const_reference n, const [Linear_Expression](#) &e)
- `Constraint operator>=` (Coefficient_traits::const_reference n, const [Linear_Expression](#) &e)
- `Constraint operator>` (Coefficient_traits::const_reference n, const [Linear_Expression](#) &e)
- `Constraint operator==` (const [Linear_Expression](#) &e, Coefficient_traits::const_reference n)
- `Constraint operator>=` (const [Linear_Expression](#) &e, Coefficient_traits::const_reference n)
- `Constraint operator>` (const [Linear_Expression](#) &e, Coefficient_traits::const_reference n)
- `Constraint operator<=` (const [Linear_Expression](#) &e1, const [Linear_Expression](#) &e2)
- `Constraint operator<=` (const [Variable](#) v1, const [Variable](#) v2)
- `Constraint operator<=` (Coefficient_traits::const_reference n, const [Linear_Expression](#) &e)
- `Constraint operator<=` (const [Linear_Expression](#) &e, Coefficient_traits::const_reference n)
- `Constraint operator<` (const [Linear_Expression](#) &e1, const [Linear_Expression](#) &e2)
- `Constraint operator<` (const [Variable](#) v1, const [Variable](#) v2)
- `Constraint operator<` (Coefficient_traits::const_reference n, const [Linear_Expression](#) &e)
- `Constraint operator<` (const [Linear_Expression](#) &e, Coefficient_traits::const_reference n)
- `void swap` ([Constraint](#) &x, [Constraint](#) &y)

10.32.1 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.

10.32.2 Member Enumeration Documentation

enum Parma_Polyhedra_Library::Constraint::Type The constraint type.

Enumerator

EQUALITY The constraint is an equality.

NONSTRICTINEQUALITY The constraint is a non-strict inequality.

STRICTINEQUALITY The constraint is a strict inequality.

10.32.3 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Constraint::Constraint (const Constraint & c) [inline] Ordinary copy constructor.

Note

The new **Constraint** will have the same representation as 'c', not default_representation, so that they are indistinguishable.

Parma_Polyhedra_Library::Constraint::Constraint (const Constraint & c, dimension_type space_dim) [inline] Copy constructor with given size.

Note

The new [Constraint](#) will have the same representation as 'c', not default_representation, so that they are indistinguishable.

Parma_Polyhedra_Library::Constraint::Constraint (const Congruence & cg, Representation r = default_representation) [explicit] Copy-constructs from equality congruence cg.

Exceptions

<i>std::invalid_argument</i>	Thrown if cg is a proper congruence.
------------------------------	--------------------------------------

10.32.4 Member Function Documentation

void Parma_Polyhedra_Library::Constraint::set_space_dimension (dimension_type space_dim) [inline]
Sets the dimension of the vector space enclosing *this to space_dim.

bool Parma_Polyhedra_Library::Constraint::remove_space_dimensions (const Variables_Set & vars) [inline] Removes all the specified dimensions from the constraint.

The space dimension of the variable with the highest space dimension in vars must be at most the space dimension of this.

Always returns true. The return value is needed for compatibility with the [Generator](#) class.

void Parma_Polyhedra_Library::Constraint::shift_space_dimensions (Variable v, dimension_type n) [inline] Shift by n positions the coefficients of variables, starting from the coefficient of v. This increases the space dimension by n.

Coefficient_traits::const_reference Parma_Polyhedra_Library::Constraint::coefficient (Variable v) const [inline] Returns the coefficient of v in *this.

Exceptions

<i>std::invalid_argument</i>	thrown if the index of v is greater than or equal to the space dimension of *this.
------------------------------	--

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 $b > 0$.

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 $b \leq 0$.

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.

bool Parma_Polyhedra_Library::Constraint::is_equal_to (const Constraint & y) const Returns `true` if `*this` is identical to `y`.

This is faster than [is_equivalent_to\(\)](#), but it may return ‘false’ even for equivalent constraints.

10.32.5 Friends And Related Function Documentation

Constraint operator< (const Linear_Expression & e1, const Linear_Expression & e2) [related]
Returns the constraint $e1 < e2$.

Constraint operator< (Variable v1, Variable v2) [related] Returns the constraint $v1 < v2$.

Constraint operator< (const Linear_Expression & e, Coefficient_traits::const_reference n) [related]
Returns the constraint $e < n$.

Constraint operator< (Coefficient_traits::const_reference n, const Linear_Expression & e) [related]
Returns the constraint $n < e$.

Constraint operator> (const Linear_Expression & e1, const Linear_Expression & e2) [related]
Returns the constraint $e1 > e2$.

Constraint operator> (Variable v1, Variable v2) [related] Returns the constraint $v1 > v2$.

Constraint operator> (const Linear_Expression & e, Coefficient_traits::const_reference n) [related]
Returns the constraint $e > n$.

Constraint operator> (Coefficient_traits::const_reference n, const Linear_Expression & e) [related]
Returns the constraint $n > e$.

Constraint operator==(const Linear_Expression & e1, const Linear_Expression & e2) [related]
Returns the constraint $e1 = e2$.

Constraint operator==(Variable v1, Variable v2) [related] Returns the constraint $v1 = v2$.

Constraint operator==(const Linear_Expression & e, Coefficient_traits::const_reference n) [related]
Returns the constraint $e = n$.

Constraint operator==(Coefficient_traits::const_reference n, const Linear_Expression & e) [related]
Returns the constraint $n = e$.

Constraint operator<= (const Linear_Expression & e1, const Linear_Expression & e2) [related]
Returns the constraint $e1 \leq e2$.

Constraint operator<= (Variable v1, Variable v2) [related] Returns the constraint $v1 \leq v2$.

Constraint operator<= (**const Linear_Expression & e**, **Coefficient_traits::const_reference n**) **[related]**
Returns the constraint $e \leq n$.

Constraint operator<= (**Coefficient_traits::const_reference n**, **const Linear_Expression & e**) **[related]**
Returns the constraint $n \leq e$.

Constraint operator>= (**const Linear_Expression & e1**, **const Linear_Expression & e2**) **[related]**
Returns the constraint $e1 \geq e2$.

Constraint operator>= (**Variable v1**, **Variable v2**) **[related]** Returns the constraint $v1 \geq v2$.

Constraint operator>= (**const Linear_Expression & e**, **Coefficient_traits::const_reference n**) **[related]**
Returns the constraint $e \geq n$.

Constraint operator>= (**Coefficient_traits::const_reference n**, **const Linear_Expression & e**) **[related]**
Returns the constraint $n \geq e$.

std::ostream & operator<< (**std::ostream & s**, **const Constraint & c**) **[related]** Output operator.

std::ostream & operator<< (**std::ostream & s**, **const Constraint::Type & t**) **[related]** Output operator.

bool operator== (**const Constraint & x**, **const Constraint & y**) **[related]** Returns `true` if and only if x is equivalent to y .

bool operator!= (**const Constraint & x**, **const Constraint & y**) **[related]** Returns `true` if and only if x is not equivalent to y .

void swap (**Constraint & x**, **Constraint & y**) **[related]**

bool operator== (**const Constraint & x**, **const Constraint & y**) **[related]**

bool operator!= (**const Constraint & x**, **const Constraint & y**) **[related]**

Constraint operator== (**const Linear_Expression & e1**, **const Linear_Expression & e2**) **[related]**

Constraint operator== (**Variable v1**, **Variable v2**) **[related]**

Constraint operator>= (**const Linear_Expression & e1**, **const Linear_Expression & e2**) **[related]**

Constraint operator>= (**const Variable v1**, **const Variable v2**) **[related]**

Constraint operator> (**const Linear_Expression & e1**, **const Linear_Expression & e2**) **[related]**

Constraint operator> (**const Variable v1**, **const Variable v2**) **[related]**

Constraint operator== (**Coefficient_traits::const_reference n**, **const Linear_Expression & e**) **[related]**

Constraint operator >= (**Coefficient_traits::const_reference** *n*, **const Linear_Expression & e**) [**related**]

Constraint operator > (**Coefficient_traits::const_reference** *n*, **const Linear_Expression & e**) [**related**]

Constraint operator == (**const Linear_Expression & e**, **Coefficient_traits::const_reference** *n*) [**related**]

Constraint operator >= (**const Linear_Expression & e**, **Coefficient_traits::const_reference** *n*) [**related**]

Constraint operator > (**const Linear_Expression & e**, **Coefficient_traits::const_reference** *n*) [**related**]

Constraint operator <= (**const Linear_Expression & e1**, **const Linear_Expression & e2**) [**related**]

Constraint operator <= (**const Variable v1**, **const Variable v2**) [**related**]

Constraint operator <= (**Coefficient_traits::const_reference** *n*, **const Linear_Expression & e**) [**related**]

Constraint operator <= (**const Linear_Expression & e**, **Coefficient_traits::const_reference** *n*) [**related**]

Constraint operator < (**const Linear_Expression & e1**, **const Linear_Expression & e2**) [**related**]

Constraint operator < (**const Variable v1**, **const Variable v2**) [**related**]

Constraint operator < (**Coefficient_traits::const_reference** *n*, **const Linear_Expression & e**) [**related**]

Constraint operator < (**const Linear_Expression & e**, **Coefficient_traits::const_reference** *n*) [**related**]

void swap (**Constraint & x**, **Constraint & y**) [**related**]

10.32.6 Member Data Documentation

const Representation Parma_Polyhedra_Library::Constraint::default_representation = SPARSE [**static**]

The representation used for new Constraints.

Note

The copy constructor and the copy constructor with specified size use the representation of the original object, so that it is indistinguishable from the original object.

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

- ppl.hh

10.33 Parma_Polyhedra_Library::Constraint_System Class Reference

A system of constraints.

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

Public Member Functions

- [Constraint_System](#) ([Representation](#) r=default_representation)
Default constructor: builds an empty system of constraints.
- [Constraint_System](#) (const [Constraint](#) &c, [Representation](#) r=default_representation)
Builds the singleton system containing only constraint c.
- [Constraint_System](#) (const [Congruence_System](#) &cgs, [Representation](#) r=default_representation)
Builds a system containing copies of any equalities in cgs.
- [Constraint_System](#) (const [Constraint_System](#) &cs)
Ordinary copy constructor.
- [Constraint_System](#) (const [Constraint_System](#) &cs, [Representation](#) r)
Copy constructor with specified representation.
- [~Constraint_System](#) ()
Destructor.
- [Constraint_System](#) & operator= (const [Constraint_System](#) &y)
Assignment operator.
- [Representation](#) representation () const
*Returns the current representation of *this.*
- void [set_representation](#) ([Representation](#) r)
*Converts *this to the specified representation.*
- [dimension_type](#) space_dimension () const
*Returns the dimension of the vector space enclosing *this.*
- void [set_space_dimension](#) ([dimension_type](#) space_dim)
Sets the space dimension of the rows in the system to space_dim.
- 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 [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.*
- void [clear](#) ()
Removes all the constraints from the constraint system and sets its space dimension to 0.
- [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 `m_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()`.

Related Functions

(Note that these are not member functions.)

- std::ostream & `operator<<` (std::ostream &s, const `Constraint_System` &cs)
Output operator.
- void `swap` (`Constraint_System` &x, `Constraint_System` &y)
- void `swap` (`Constraint_System` &x, `Constraint_System` &y)

10.33.1 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.

10.33.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Constraint_System::Constraint_System (const Constraint_System & cs) [inline] Ordinary copy constructor.

Note

The copy will have the same representation as 'cs', to make it indistinguishable from 'cs'.

10.33.3 Friends And Related Function Documentation

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 ", ".

void swap (Constraint_System & x, Constraint_System & y) [related]

void swap (Constraint_System & x, Constraint_System & y) [related] The documentation for this class was generated from the following file:

- ppl.hh

10.34 Parma_Polyhedra_Library::Constraint_System_const_iterator Class Reference

An iterator over a system of constraints.

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

Inherits `iterator< std::forward_iterator_tag, Constraint, std::ptrdiff_t, const Constraint *, const Constraint & >`.

Public Member Functions

- [Constraint_System_const_iterator](#) ()
Default constructor.
- [Constraint_System_const_iterator](#) (const [Constraint_System_const_iterator](#) &y)
Ordinary copy constructor.
- [~Constraint_System_const_iterator](#) ()
Destructor.
- [Constraint_System_const_iterator](#) & operator= (const [Constraint_System_const_iterator](#) &y)
Assignment operator.
- const [Constraint](#) & operator* () const
Dereference operator.
- const [Constraint](#) * operator-> () const
Indirect member selector.
- [Constraint_System_const_iterator](#) & operator++ ()
Prefix increment operator.
- [Constraint_System_const_iterator](#) operator++ (int)
Postfix increment operator.
- bool operator== (const [Constraint_System_const_iterator](#) &y) const
*Returns true if and only if *this and y are identical.*
- bool operator!= (const [Constraint_System_const_iterator](#) &y) const
*Returns true if and only if *this and y are different.*

10.34.1 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

10.35 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.

10.35.1 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.

10.35.2 Member Function Documentation

template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Constraints_Reduction<D1, D2 >::product_reduce (D1 & d1, D2 & d2) 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

<i>d1</i>	A pointset domain element;
<i>d2</i>	A pointset domain element;

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

- ppl.hh

10.36 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 &pset)
Constructs a COW-wrapped object corresponding to the pointset pset.
- [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 `m_swap` (`Determinate` &y)
*Swaps `*this` with `y`.*

Related Functions

(Note that these are not member functions.)

- `template<typename PSET >`
`void swap` (`Determinate`< PSET > &x, `Determinate`< PSET > &y)
Swaps `x` with `y`.
- `template<typename PSET >`
`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.
- `template<typename PSET >`
`bool operator!=` (const `Determinate`< PSET > &x, const `Determinate`< PSET > &y)
Returns `true` if and only if `x` and `y` are different COW-wrapped pointsets.
- `template<typename PSET >`
`std::ostream & operator<<` (`std::ostream` &, const `Determinate`< PSET > &)
Output operator.
- `template<typename PSET >`
`std::ostream & operator<<` (`std::ostream` &s, const `Determinate`< PSET > &x)
- `template<typename PSET >`
`bool operator==` (const `Determinate`< PSET > &x, const `Determinate`< PSET > &y)
- `template<typename PSET >`
`bool operator!=` (const `Determinate`< PSET > &x, const `Determinate`< PSET > &y)
- `template<typename PSET >`
`void swap` (`Determinate`< PSET > &x, `Determinate`< PSET > &y)

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 ()`

10.36.1 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.

10.36.2 Member Function Documentation

`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.

10.36.3 Friends And Related Function Documentation

`template<typename PSET > void swap (Determinate< PSET > & x, Determinate< PSET > & y) [related]` Swaps `x` with `y`.

`template<typename PSET > std::ostream & operator<< (std::ostream &, const Determinate< PSET > &) [related]` Output operator.

`template<typename PSET > std::ostream & operator<< (std::ostream & s, const Determinate< PSET > & x) [related]`

`template<typename PSET > bool operator==(const Determinate< PSET > & x, const Determinate< PSET > & y) [related]`

`template<typename PSET > bool operator!=(const Determinate< PSET > & x, const Determinate< PSET > & y) [related]`

`template<typename PSET > void swap (Determinate< PSET > & x, Determinate< PSET > & y) [related]` The documentation for this class was generated from the following file:

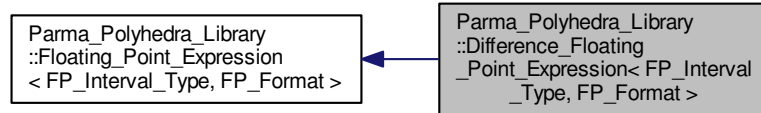
- `ppl.hh`

10.37 Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic Difference Floating Point Expression.

`#include <ppl.hh>`

Inheritance diagram for Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form](#) [FP_Linear_Form](#)
Alias for the Linear_Form<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Interval_Abstract_Store](#) [FP_Interval_Abstract_Store](#)
Alias for the Box<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form_Abstract_Store](#) [FP_Linear_Form_Abstract_Store](#)
Alias for the std::map<dimension_type, FP_Linear_Form> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[boundary_type](#) [boundary_type](#)
Alias for the FP_Interval_Type::boundary_type from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[info_type](#) [info_type](#)
Alias for the FP_Interval_Type::info_type from [Floating_Point_Expression](#).

Public Member Functions

- bool [linearize](#) (const [FP_Interval_Abstract_Store](#) &int_store, const [FP_Linear_Form_Abstract_Store](#) &lf_store, [FP_Linear_Form](#) &result) const
Linearizes the expression in a given astract store.
- void [m_swap](#) ([Difference_Floating_Point_Expression](#)< FP_Interval_Type, FP_Format > &y)
*Swaps *this with y.*

Constructors and Destructor

- `Difference_Floating_Point_Expression` (`Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > *const x, `Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > *const y)
Constructor with two parameters: builds the difference floating point expression corresponding to $x \ominus y$.
- `~Difference_Floating_Point_Expression` ()
Destructor.

Related Functions

(Note that these are not member functions.)

- `template<typename FP_Interval_Type, typename FP_Format > void swap` (`Difference_Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > &x, `Difference_Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > &y)
Swaps x with y.
- `template<typename FP_Interval_Type, typename FP_Format > void swap` (`Difference_Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > &x, `Difference_Floating_Point_Expression`< `FP_Interval_Type`, `FP_Format` > &y)

Additional Inherited Members

10.37.1 Detailed Description

`template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >`

A generic Difference Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of difference floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxminus^\#$ two sound abstract operators on linear form such that:

$$\begin{aligned} \left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) &= (i \oplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \oplus^\# i'_v) v, \\ \left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxminus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) &= (i \ominus^\# i') + \sum_{v \in \mathcal{V}} (i_v \ominus^\# i'_v) v. \end{aligned}$$

Given an expression $e_1 \ominus e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \ominus e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ on \mathcal{V} as follows:

$$\llbracket e_1 \ominus e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxminus^\# \llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \varepsilon_f \left(\llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \varepsilon_f \left(\llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# m_{f_f}[-1, 1]$$

where $\varepsilon_f(l)$ is the linear form computed by calling method `Floating_Point_Expression::relative_error` on l and m_{f_f} is a rounding error defined in `Floating_Point_Expression::absolute_error`.

10.37.2 Member Function Documentation

template<typename FP_Interval_Type , typename FP_Format > bool Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const **[virtual]** Linearizes the expression in a given astract store.

Makes *result* become the linearization of **this* in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

true if the linearization succeeded, *false* otherwise.

Note that all variables occuring in the expressions represented by *first_operand* and *second_operand* MUST have an associated value in *int_store*. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how *result* is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.37.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) **[related]** Swaps *x* with *y*.

template<typename FP_Interval_Type , typename FP_Format > void swap (Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) **[related]** The documentation for this class was generated from the following file:

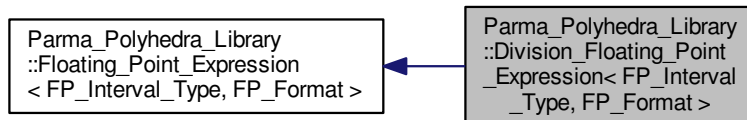
- *ppl.hh*

10.38 Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic Division Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Linear_Form FP_Linear_Form
Alias for the Linear_Form<FP.Interval_Type> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Interval_Abstract_Store FP_Interval_Abstract_Store
Alias for the Box<FP.Interval_Type> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Linear_Form_Abstract_Store FP_Linear_Form_Abstract_Store
Alias for the std::map<dimension_type, FP_Linear_Form> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::boundary_type boundary_type
Alias for the FP.Interval_Type::boundary_type from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::info_type info_type
Alias for the FP.Interval_Type::info_type from Floating_Point_Expression.

Public Member Functions

- bool linearize (const FP_Interval_Abstract_Store &int_store, const FP_Linear_Form_Abstract_Store &lf_store, FP_Linear_Form &result) const
Linearizes the expression in a given astract store.
- void m_swap (Division_Floating_Point_Expression< FP.Interval_Type, FP.Format > &y)
*Swaps *this with y.*

Constructors and Destructor

- Division_Floating_Point_Expression (Floating_Point_Expression< FP.Interval_Type, FP.Format > *const num, Floating_Point_Expression< FP.Interval_Type, FP.Format > *const den)
Constructor with two parameters: builds the division floating point expression corresponding to $num \oslash den$.
- ~Division_Floating_Point_Expression ()
Destructor.

Related Functions

(Note that these are not member functions.)

- template<typename FP.Interval_Type, typename FP.Format >
void swap (Division_Floating_Point_Expression< FP.Interval_Type, FP.Format > &x, Division_Floating_Point_Expression< FP.Interval_Type, FP.Format > &y)
Swaps x with y.

- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap (Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`

Additional Inherited Members

10.38.1 Detailed Description

template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >

A generic Division Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of division floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxminus^\#$ two sound abstract operator on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \boxplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxplus^\# i'_v) v,$$

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxminus^\# i' = (i \boxminus^\# i') + \sum_{v \in \mathcal{V}} (i_v \boxminus^\# i') v.$$

Given an expression $e_1 \odot [a, b]$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \odot [a, b] \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket e_1 \odot [a, b] \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \left(\llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxminus^\# [a, b] \right) \boxplus^\# \left(\varepsilon_f \left(\llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxminus^\# [a, b] \right) \boxplus^\# m_{f_f}[-1, 1],$$

given an expression $e_1 \odot e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \odot e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket e_1 \odot e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket e_1 \odot \iota \left(\llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \rho^\# \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket,$$

where $\varepsilon_f(l)$ is the linear form computed by calling method `Floating_Point_Expression::relative_error` on l , $\iota(l)\rho^\#$ is the linear form computed by calling method `Floating_Point_Expression::intervalize` on l and $\rho^\#$, and m_{f_f} is a rounding error defined in `Floating_Point_Expression::absolute_error`.

10.38.2 Member Function Documentation

template<typename FP_Interval_Type, typename FP_Format > bool Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const **[virtual]** Linearizes the expression in a given astract store.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

Note that all variables occurring in the expressions represented by `first_operand` and `second_operand` MUST have an associated value in `int_store`. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how `result` is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.38.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] Swaps `x` with `y`.

template<typename FP_Interval_Type , typename FP_Format > void swap (Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Division_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.39 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>
```

10.39.1 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:

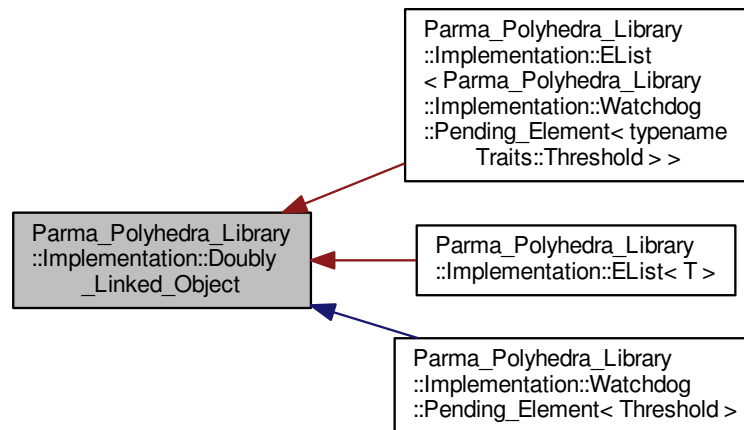
- `ppl.hh`

10.40 Parma_Polyhedra_Library::Implementation::Doubly_Linked_Object Class Reference

A (base) class for doubly linked objects.

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

Inheritance diagram for Parma_Polyhedra_Library::Implementation::Doubly_Linked_Object:



Public Member Functions

- [Doubly_Linked_Object](#) ()
Default constructor.
- [Doubly_Linked_Object](#) ([Doubly_Linked_Object](#) *f, [Doubly_Linked_Object](#) *b)
*Creates a chain element with forward link *f* and backward link *b*.*
- void [insert_before](#) ([Doubly_Linked_Object](#) &y)
*Inserts *y* before *this.*
- void [insert_after](#) ([Doubly_Linked_Object](#) &y)
*Inserts *y* after *this.*
- [Doubly_Linked_Object](#) * [erase](#) ()
*Erases *this from the chain and returns a pointer to the next element.*
- [~Doubly_Linked_Object](#) ()
*Erases *this from the chain.*

10.40.1 Detailed Description

A (base) class for doubly linked objects.

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

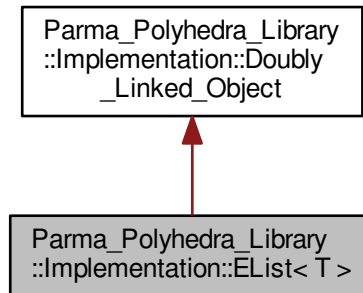
- ppl.hh

10.41 Parma_Polyhedra_Library::Implementation::EList< T > Class Template Reference

A simple kind of embedded list (i.e., a doubly linked objects where the links are embedded in the objects themselves).

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

Inheritance diagram for Parma_Polyhedra_Library::Implementation::EList< T >:



Public Types

- typedef [EList_Iterator](#)< const T > [const_iterator](#)
A const iterator to traverse the list.
- typedef [EList_Iterator](#)< T > [iterator](#)
A non-const iterator to traverse the list.

Public Member Functions

- [EList](#) ()
Constructs an empty list.
- [~EList](#) ()
Destructs the list and all the elements in it.
- void [push_front](#) (T &obj)
Pushes obj to the front of the list.
- void [push_back](#) (T &obj)
Pushes obj to the back of the list.
- [iterator insert](#) ([iterator](#) position, T &obj)
Inserts obj just before position and returns an iterator that points to the inserted object.
- [iterator erase](#) ([iterator](#) position)
Removes the element pointed to by position, returning an iterator pointing to the next element, if any, or end(), otherwise.
- bool [empty](#) () const
Returns true if and only if the list is empty.
- [iterator begin](#) ()
Returns an iterator pointing to the beginning of the list.
- [iterator end](#) ()
Returns an iterator pointing one past the last element in the list.
- [const_iterator begin](#) () const
Returns a const iterator pointing to the beginning of the list.
- [const_iterator end](#) () const

Returns a const iterator pointing one past the last element in the list.

- `bool OK () const`

Checks if all the invariants are satisfied.

Additional Inherited Members

10.41.1 Detailed Description

`template<typename T>class Parma_Polyhedra_Library::Implementation::EList< T >`

A simple kind of embedded list (i.e., a doubly linked objects where the links are embedded in the objects themselves).

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

- `ppl.hh`

10.42 Parma_Polyhedra_Library::Implementation::EList_Iterator< T > Class Template Reference

A class providing iterators for embedded lists.

`#include <ppl.hh>`

Public Member Functions

- `EList_Iterator ()`
Constructs an iterator pointing to nothing.
- `EList_Iterator (Doubly_Linked_Object *p)`
Constructs an iterator pointing to p.
- `EList_Iterator & operator= (Doubly_Linked_Object *p)`
*Changes *this so that it points to p.*
- `T * operator-> ()`
Indirect member selector.
- `T & operator* ()`
Dereference operator.
- `EList_Iterator & operator++ ()`
Preincrement operator.
- `EList_Iterator operator++ (int)`
Postincrement operator.
- `EList_Iterator & operator-- ()`
Predecrement operator.
- `EList_Iterator operator-- (int)`
Postdecrement operator.

Friends

- `bool operator== (const EList_Iterator &x, const EList_Iterator &y)`
Returns true if and only if x and y are equal.
- `bool operator!= (const EList_Iterator &x, const EList_Iterator &y)`
Returns true if and only if x and y are different.

10.42.1 Detailed Description

template<typename T>class Parma_Polyhedra_Library::Implementation::EList_Iterator< T >

A class providing iterators for embedded lists.

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

- ppl.hh

10.43 Parma_Polyhedra_Library::Floating_Point_Constant< Target > Class Template Reference

A floating-point constant concrete expression.

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

10.43.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Floating_Point_Constant< Target >

A floating-point constant concrete expression.

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

- ppl.hh

10.44 Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target > Class Template Reference

Base class for floating-point constant concrete expression.

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

10.44.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target >

Base class for floating-point constant concrete expression.

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

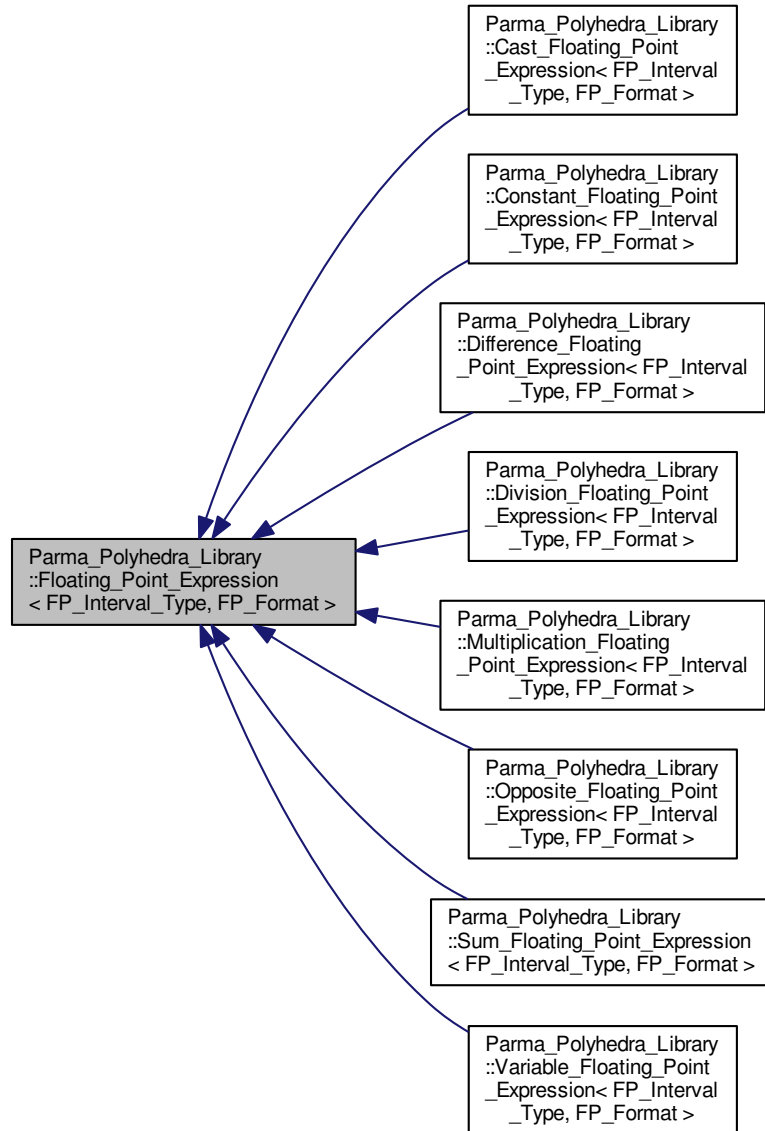
- ppl.hh

10.45 Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

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

Inheritance diagram for Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP-

_Format >:



Public Types

- typedef [Linear_Form](#)
< FP.Interval.Type > [FP.Linear_Form](#)
Alias for a linear form with template argument FP.Interval.Type.
- typedef [Box](#)< FP.Interval.Type > [FP.Interval.Abstract.Store](#)
Alias for a map that associates a variable index to an interval.
- typedef std::map
< [dimension.type](#),

[FP_Linear_Form](#) > [FP_Linear_Form_Abstract_Store](#)

Alias for a map that associates a variable index to a linear form.

- typedef [FP_Interval_Type::boundary_type](#) [boundary_type](#)

The floating point format used by the analyzer.

- typedef [FP_Interval_Type::info_type](#) [info_type](#)
- The interval policy used by [FP_Interval_Type](#).*

Public Member Functions

- virtual [~Floating_Point_Expression](#) ()
- Destructor.*
- virtual bool [linearize](#) (const [FP_Interval_Abstract_Store](#) &int_store, const [FP_Linear_Form_Abstract_Store](#) &lf_store, [FP_Linear_Form](#) &result) const =0
- Linearizes a floating point expression.*

Static Public Member Functions

- static bool [overflows](#) (const [FP_Linear_Form](#) &lf)
- Verifies if a given linear form overflows.*
- static void [relative_error](#) (const [FP_Linear_Form](#) &lf, [FP_Linear_Form](#) &result)
- Computes the relative error of a given linear form.*
- static void [intervalize](#) (const [FP_Linear_Form](#) &lf, const [FP_Interval_Abstract_Store](#) &store, [FP_Interval_Type](#) &result)
- Makes `result` become an interval that overapproximates all the possible values of `lf` in the interval abstract store `store`.*

Static Public Attributes

- static [FP_Interval_Type](#) [absolute_error](#) = [compute_absolute_error](#)()
- Absolute error.*

10.45.1 Detailed Description

template<typename [FP_Interval_Type](#), typename [FP_Format](#)>class [Parma_Polyhedra_Library::Floating_Point_Expression](#)< [FP_Interval_Type](#), [FP_Format](#) >

\ A floating point expression on a given format.

This class represents a concrete *floating point expression*. This includes constants, floating point variables, binary and unary arithmetic operators.

Template type parameters

- The class template type parameter [FP_Interval_Type](#) represents the type of the intervals used in the abstract domain. The interval bounds should have a floating point type.
- The class template type parameter [FP_Format](#) represents the floating point format used in the concrete domain. This parameter must be a struct similar to the ones defined in file [Float_defs.hh](#), even though it is sufficient to define the three fields `BASE`, `MANTISSA_BITS` and `EXPONENT_BIAS`.

10.45.2 Member Typedef Documentation

template<typename FP_Interval_Type , typename FP_Format > typedef Box<FP_Interval_Type> Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::FP_Interval_Abstract_Store Alias for a map that associates a variable index to an interval.

Alias for a [Box](#) storing lower and upper bounds for floating point variables.

The type a linear form abstract store associating each variable with an interval that correctly approximates its value.

template<typename FP_Interval_Type , typename FP_Format > typedef std::map<dimension_type, FP_Linear_Form> Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::FP_Linear_Form_Abstract_Store Alias for a map that associates a variable index to a linear form.

The type a linear form abstract store associating each variable with a linear form that correctly approximates its value.

10.45.3 Member Function Documentation

template<typename FP_Interval_Type , typename FP_Format > virtual bool Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const [pure virtual] Linearizes a floating point expression.

Makes *result* become a linear form that correctly approximates the value of the floating point expression in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	Becomes the linearized expression.

Returns

`true` if the linearization succeeded, `false` otherwise.

Formally, if **this* represents the expression *e*, *int_store* represents the interval abstract store $\rho^\#$ and *lf_store* represents the linear form abstract store $\rho_l^\#$, then *result* will become $\langle e \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ if the linearization succeeds.

All variables occurring in the floating point expression MUST have an associated interval in *int_store*. If this precondition is not met, calling the method causes an undefined behavior.

Implemented in [Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Division_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Difference_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), [Parma_Polyhedra_Library::Constant_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#), and [Parma_Polyhedra_Library::Cast_Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

template<typename FP_Interval_Type , typename FP_Format > bool Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::overflows (const FP_Linear_Form & *lf*) [inline], [static] Verifies if a given linear form overflows.

Parameters

<i>lf</i>	The linear form to verify.
-----------	----------------------------

Returns

Returns `false` if all coefficients in `lf` are bounded, `true` otherwise.

template<typename FP_Interval_Type , typename FP_Format > void Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::relative_error (const FP_Linear_Form & lf, FP_Linear_Form & result) [static] Computes the relative error of a given linear form.

Static helper method that is used by `linearize` to account for the relative errors on `lf`.

Parameters

<i>lf</i>	The linear form used to compute the relative error.
<i>result</i>	Becomes the linear form corresponding to a relative error committed on <code>lf</code> .

This method makes `result` become a linear form obtained by evaluating the function $\varepsilon_f(l)$ on the linear form `lf`. This function is defined such as:

$$\varepsilon_f \left([a, b] + \sum_{v \in \mathcal{V}} [a_v, b_v] v \right) \stackrel{\text{def}}{=} (\max(|a|, |b|) \otimes^\# [-\beta^{-p}, \beta^{-p}]) + \sum_{v \in \mathcal{V}} (\max(|a_v|, |b_v|) \otimes^\# [-\beta^{-p}, \beta^{-p}]) v$$

where p is the fraction size in bits for the format f and β the base.

template<typename FP_Interval_Type , typename FP_Format > void Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::intervalize (const FP_Linear_Form & lf, const FP_Interval_Abstract_Store & store, FP_Interval_Type & result) [static] Makes `result` become an interval that overapproximates all the possible values of `lf` in the interval abstract store `store`.

Parameters

<i>lf</i>	The linear form to aproximate.
<i>store</i>	The abstract store.
<i>result</i>	The linear form that will be modified.

This method makes `result` become $\iota(lf)\rho^\#$, that is an interval defined as:

$$\iota \left(i + \sum_{v \in \mathcal{V}} i_v v \right) \rho^\# \stackrel{\text{def}}{=} i \oplus^\# \left(\bigoplus_{v \in \mathcal{V}}^\# i_v \otimes^\# \rho^\#(v) \right)$$

10.45.4 Member Data Documentation

template<typename FP_Interval_Type , typename FP_Format > FP_Interval_Type Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >::absolute_error = compute_absolute_error() [static] Absolute error.

Represents the interval $[-\omega, \omega]$ where ω is the smallest non-zero positive number in the less precise floating point format between the analyzer format and the analyzed format.

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

- ppl.hh

10.46 Parma Polyhedra Library::FP_Oracle< Target, FP_Interval_Type > Class Template Reference

An abstract class to be implemented by an external analyzer such as ECLAIR in order to provide to the PPL the necessary information for performing the analysis of floating point computations.

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

Public Member Functions

- virtual bool [get_interval](#) ([dimension_type](#) dim, FP_Interval_Type &result) const =0
Asks the external analyzer for an interval that correctly approximates the floating point entity referenced by `dim`. Result is stored into `result`.
- virtual bool [get_fp_constant_value](#) (const [Floating_Point_Constant](#)< Target > &expr, FP_Interval_Type &result) const =0
Asks the external analyzer for an interval that correctly approximates the value of floating point constant `expr`. Result is stored into `result`.
- virtual bool [get_integer_expr_value](#) (const [Concrete_Expression](#)< Target > &expr, FP_Interval_Type &result) const =0
Asks the external analyzer for an interval that correctly approximates the value of `expr`, which must be of integer type. Result is stored into `result`.
- virtual bool [get_associated_dimensions](#) (const [Approximable_Reference](#)< Target > &expr, std::set< [dimension_type](#) > &result) const =0
Asks the external analyzer for the possible space dimensions that are associated to the approximable reference `expr`. Result is stored into `result`.

10.46.1 Detailed Description

template<typename Target, typename FP_Interval_Type>class Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >

An abstract class to be implemented by an external analyzer such as ECLAIR in order to provide to the PPL the necessary information for performing the analysis of floating point computations.

Template type parameters

- The class template parameter `Target` specifies the implementation of [Concrete_Expression](#) to be used.
- The class template parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain. The interval bounds should have a floating point type.

10.46.2 Member Function Documentation

template<typename Target, typename FP_Interval_Type> virtual bool Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >::get_interval ([dimension_type](#) *dim*, FP_Interval_Type &*result*) const [pure virtual] Asks the external analyzer for an interval that correctly approximates the floating point entity referenced by `dim`. Result is stored into `result`.

Returns

`true` if the analyzer was able to find a correct approximation, `false` otherwise.

template<typename Target , typename FP_Interval_Type > virtual bool Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >::get_fp_constant_value (const Floating_Point_Constant< Target > & *expr*, FP_Interval_Type & *result*) const [pure virtual] Asks the external analyzer for an interval that correctly approximates the value of floating point constant *expr*. Result is stored into *result*.

Returns

`true` if the analyzer was able to find a correct approximation, `false` otherwise.

template<typename Target , typename FP_Interval_Type > virtual bool Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >::get_integer_expr_value (const Concrete_Expression< Target > & *expr*, FP_Interval_Type & *result*) const [pure virtual] Asks the external analyzer for an interval that correctly approximates the value of *expr*, which must be of integer type. Result is stored into *result*.

Returns

`true` if the analyzer was able to find a correct approximation, `false` otherwise.

template<typename Target , typename FP_Interval_Type > virtual bool Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >::get_associated_dimensions (const Approximable_Reference< Target > & *expr*, std::set< dimension_type > & *result*) const [pure virtual] Asks the external analyzer for the possible space dimensions that are associated to the approximable reference *expr*. Result is stored into *result*.

Returns

`true` if the analyzer was able to return the (possibly empty!) set, `false` otherwise.

The resulting set MUST NOT contain `not_a_dimension()`.

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

- `ppl.hh`

10.47 Parma_Polyhedra_Library::Generator Class Reference

A line, ray, point or closure point.

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

Public Types

- enum `Type` { `LINE`, `RAY`, `POINT`, `CLOSURE_POINT` }
The generator type.
- typedef `Expression_Hide_Last`
 < `Expression_Hide_Inhomo`
 < `Linear_Expression` > > `expr_type`
The type of the (adapted) internal expression.

Public Member Functions

- `Generator` (`Representation` `r=default_representation`)
Constructs the point at the origin.
- `Generator` (const `Generator` &`g`)
- `Generator` (const `Generator` &`g`, `Representation` `r`)

- *Copy constructor with given representation.*
- **Generator** (const **Generator** &g, **dimension_type** space_dim)
- **Generator** (const **Generator** &g, **dimension_type** space_dim, **Representation** r)
 - *Copy constructor with given representation and space dimension.*
- **~Generator** ()
 - *Destructor.*
- **Generator & operator=** (const **Generator** &g)
 - *Assignment operator.*
- **Representation representation** () const
 - *Returns the current representation of *this.*
- void **set_representation** (**Representation** r)
 - *Converts *this to the specified representation.*
- **dimension_type space_dimension** () const
 - *Returns the dimension of the vector space enclosing *this.*
- void **set_space_dimension** (**dimension_type** space_dim)
- void **swap_space_dimensions** (**Variable** v1, **Variable** v2)
 - *Swaps the coefficients of the variables v1 and v2 .*
- bool **remove_space_dimensions** (const **Variables_Set** &vars)
 - *Removes all the specified dimensions from the generator.*
- void **permute_space_dimensions** (const std::vector< **Variable** > &cycle)
 - *Permutates the space dimensions of the generator.*
- void **shift_space_dimensions** (**Variable** v, **dimension_type** n)
- **Type type** () const
 - *Returns the generator type of *this.*
- bool **is_line** () const
 - *Returns true if and only if *this is a line.*
- bool **is_ray** () const
 - *Returns true if and only if *this is a ray.*
- bool **is_point** () const
 - *Returns true if and only if *this is a point.*
- bool **is_closure_point** () const
 - *Returns true if and only if *this is a closure point.*
- **Coefficient_traits::const_reference coefficient** (**Variable** v) const
 - *Returns the coefficient of v in *this.*
- **Coefficient_traits::const_reference divisor** () const
 - *If *this is either a point or a closure point, returns its divisor.*
- **memory_size_type total_memory_in_bytes** () const
 - *Returns a lower bound to the total size in bytes of the memory occupied by *this.*
- **memory_size_type external_memory_in_bytes** () const
 - *Returns the size in bytes of the memory managed by *this.*
- bool **is_equivalent_to** (const **Generator** &y) const
 - *Returns true if and only if *this and y are equivalent generators.*
- bool **is_equal_to** (const **Generator** &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.*
- `void m_swap (Generator &y)`
*Swaps `*this` with `y`.*
- `expr_type expression () const`
Partial read access to the (adapted) internal expression.

Static Public Member Functions

- `static Generator line (const Linear_Expression &e, Representation r=default_representation)`
Returns the line of direction `e`.
- `static Generator ray (const Linear_Expression &e, Representation r=default_representation)`
Returns the ray of direction `e`.
- `static Generator point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=default_representation)`
Returns the point at `e / d`.
- `static Generator point (Representation r)`
Returns the origin.
- `static Generator point (const Linear_Expression &e, Representation r)`
Returns the point at `e`.
- `static Generator closure_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=default_representation)`
Returns the closure point at `e / d`.
- `static Generator closure_point (Representation r)`
Returns the closure point at the origin.
- `static Generator closure_point (const Linear_Expression &e, Representation r)`
Returns the closure point at `e`.
- `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 .

Static Public Attributes

- `static const Representation default_representation = SPARSE`
The representation used for new [Generators](#).

Related Functions

(Note that these are not member functions.)

- `std::ostream & operator<< (std::ostream &s, const Generator &g)`
Output operator.
- `void swap (Generator &x, Generator &y)`
Swaps x with y .
- `Generator line (const Linear_Expression &e, Representation r=Generator::default_representation)`
Shorthand for `Generator::line(const Linear_Expression& e, Representation r)`.
- `Generator ray (const Linear_Expression &e, Representation r=Generator::default_representation)`
Shorthand for `Generator::ray(const Linear_Expression& e, Representation r)`.
- `Generator point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=Generator::default_representation)`
Shorthand for `Generator::point(const Linear_Expression& e, Coefficient_traits::const_reference d, Representation r)`.
- `Generator point (Representation r)`
Shorthand for `Generator::point(Representation r)`.
- `Generator point (const Linear_Expression &e, Representation r)`
Shorthand for `Generator::point(const Linear_Expression& e, Representation r)`.
- `Generator closure_point (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=Generator::default_representation)`
Shorthand for `Generator::closure_point(const Linear_Expression& e, Coefficient_traits::const_reference d, Representation r)`.
- `Generator closure_point (Representation r)`
Shorthand for `Generator::closure_point(Representation r)`.
- `Generator closure_point (const Linear_Expression &e, Representation r)`
Shorthand for `Generator::closure_point(const Linear_Expression& e, Representation r)`.
- `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)`
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 rectilinear_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)`
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.
- `Generator line (const Linear_Expression &e, Representation r)`
- `Generator ray (const Linear_Expression &e, Representation r)`
- `Generator point (const Linear_Expression &e, Coefficient_traits::const_reference d, Representation r)`
- `Generator point (Representation r)`
- `Generator point (const Linear_Expression &e, Representation r)`
- `Generator closure_point (const Linear_Expression &e, Coefficient_traits::const_reference d, Representation r)`
- `Generator closure_point (Representation r)`
- `Generator closure_point (const Linear_Expression &e, Representation r)`
- `bool operator== (const Generator &x, const Generator &y)`
- `bool operator!= (const Generator &x, const Generator &y)`
- `template<typename Temp , typename To >`
`bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `template<typename Temp , typename To >`
`bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`
- `template<typename To >`
`bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`
- `template<typename Temp , typename To >`
`bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `template<typename Temp , typename To >`
`bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`

- `template<typename To >`
`bool euclidean_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`
- `template<typename Temp, typename To >`
`bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `template<typename Temp, typename To >`
`bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`
- `template<typename To >`
`bool l_infinity_distance_assign (Checked_Number< To, Extended_Number_Policy > &r, const Generator &x, const Generator &y, const Rounding_Dir dir)`
- `void swap (Generator &x, Generator &y)`

10.47.1 Detailed Description

A line, ray, point or closure point.

An object of the class `Generator` is one of the following:

- a line $\mathbf{l} = (a_0, \dots, a_{n-1})^T$;
- a ray $\mathbf{r} = (a_0, \dots, a_{n-1})^T$;
- a point $\mathbf{p} = (\frac{a_0}{d}, \dots, \frac{a_{n-1}}{d})^T$;
- a closure point $\mathbf{c} = (\frac{a_0}{d}, \dots, \frac{a_{n-1}}{d})^T$;

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 $p = (1, 0, 2)^T \in \mathbb{R}^3$:

```
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 $0 \in \mathbb{R}^3$ 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 $0 \in \mathbb{R}^2$:

```
Generator origin2 = point(0*y);
```

The following two lines of code both define the only point having space dimension zero, namely $0 \in \mathbb{R}^0$. 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 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 $q = (-1.5, 3.2, 2.1)^T \in \mathbb{R}^3$ 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 $c = (1, 0, 2)^T \in \mathbb{R}^3$ 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 g_1 is a point having coordinates $(a_0, \dots, a_{n-1})^T$, we construct the closure point g_2 having coordinates $(a_0, 2a_1, \dots, (i+1)a_i, \dots, na_{n-1})^T$.

```
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.

10.47.2 Member Enumeration Documentation

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.

10.47.3 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Generator::Generator (const Generator & g) [inline] Ordinary copy constructor. The representation of the new [Generator](#) will be the same as g.

Parma_Polyhedra_Library::Generator::Generator (const Generator & g, dimension_type space_dim) [inline] Copy constructor with given space dimension. The representation of the new [Generator](#) will be the same as g.

10.47.4 Member Function Documentation

static Generator Parma_Polyhedra_Library::Generator::line (const Linear.Expression & e, Representation r = default_representation) [static] Returns the line of direction e.

Exceptions

<i>std::invalid_argument</i>	Thrown if the homogeneous part of e represents the origin of the vector space.
------------------------------	--

static Generator Parma_Polyhedra_Library::Generator::ray (const Linear_Expression & e , Representation r = default_representation) [static] Returns the ray of direction e .

Exceptions

<i>std::invalid_argument</i>	Thrown if the homogeneous part of e represents the origin of the vector space.
------------------------------	--

static Generator Parma_Polyhedra_Library::Generator::point (const Linear_Expression & e = Linear_Expression::zero(), Coefficient_traits::const_reference d = Coefficient_one(), Representation r = default_representation) [static] Returns the point at e / d .

Both e and d are optional arguments, with default values [Linear_Expression::zero\(\)](#) and [Coefficient_one\(\)](#), respectively.

Exceptions

<i>std::invalid_argument</i>	Thrown if d is zero.
------------------------------	------------------------

static Generator Parma_Polyhedra_Library::Generator::closure_point (const Linear_Expression & e = Linear_Expression::zero(), Coefficient_traits::const_reference d = Coefficient_one(), Representation r = default_representation) [static] Returns the closure point at e / d .

Both e and d are optional arguments, with default values [Linear_Expression::zero\(\)](#) and [Coefficient_one\(\)](#), respectively.

Exceptions

<i>std::invalid_argument</i>	Thrown if d is zero.
------------------------------	------------------------

void Parma_Polyhedra_Library::Generator::set_space_dimension (dimension_type $space_dim$) [inline]
Sets the dimension of the vector space enclosing $*this$ to $space_dim$.

bool Parma_Polyhedra_Library::Generator::remove_space_dimensions (const Variables_Set & $vars$)
Removes all the specified dimensions from the generator.

The space dimension of the variable with the highest space dimension in $vars$ must be at most the space dimension of $this$.

If all dimensions with nonzero coefficients are removed from a ray or a line, it is changed into a point and this method returns `false`. Otherwise, it returns `true`.

void Parma_Polyhedra_Library::Generator::permute_space_dimensions (const std::vector< Variable > & $cycle$) Permutes the space dimensions of the generator.

Parameters

<i>cycle</i>	A vector representing a cycle of the permutation according to which the space dimensions must be rearranged.
--------------	--

The `cycle` vector represents a cycle of a permutation of space dimensions. For example, the permutation $\{x_1 \mapsto x_2, x_2 \mapsto x_3, x_3 \mapsto x_1\}$ can be represented by the vector containing x_1, x_2, x_3 .

void Parma_Polyhedra_Library::Generator::shift_space_dimensions (Variable v , dimension_type n) **[inline]** Shift by n positions the coefficients of variables, starting from the coefficient of v . This increases the space dimension by n .

Coefficient_traits::const_reference Parma_Polyhedra_Library::Generator::coefficient (Variable v) **const [inline]** Returns the coefficient of v in $*this$.

Exceptions

<i>std::invalid_argument</i>	Thrown if the index of v is greater than or equal to the space dimension of $*this$.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if $*this$ is neither a point nor a closure point.
------------------------------	---

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.

bool Parma_Polyhedra_Library::Generator::is_equal_to (const Generator & y) **const** Returns `true` if $*this$ is identical to y .

This is faster than `is_equivalent_to()`, but it may return ‘false’ even for equivalent generators.

10.47.5 Friends And Related Function Documentation

std::ostream & operator<< (std::ostream & s , const Generator & g) **[related]** Output operator.

void swap (Generator & x , Generator & y) **[related]** Swaps x with y .

Generator line (const Linear_Expression & e , Representation r = Generator::default_representation) **[related]** Shorthand for `Generator::line(const Linear_Expression& e , Representation r)`.

Generator ray (const Linear_Expression & e , Representation r = Generator::default_representation) **[related]** Shorthand for `Generator::ray(const Linear_Expression& e , Representation r)`.

Generator point (Representation r) **[related]** Shorthand for `Generator::point(Representation r)`.

Generator closure_point (Representation r) **[related]** Shorthand for `Generator::closure_point(-Representation r)`.

bool operator== (const Generator & x , const Generator & y) **[related]** Returns `true` if and only if x is equivalent to y .

bool operator!= (const Generator & x , const Generator & y) **[related]** 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) [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>`.

Note

Distances are *only* defined between generators that are points and/or closure points; for rays or lines, `false` is returned.

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) [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<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.

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.

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.

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) [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<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.

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.

template<typename To > bool Linfinity_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.

template<typename Temp , typename To > bool Linfinity_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<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.

template<typename Temp , typename To > bool Linfinity_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.

std::ostream & operator<< (std::ostream & *s*, const Generator::Type & *t*) [related] Output operator.

Generator line (const Linear_Expression & *e*, Representation *r*) [related]

Generator ray (const Linear_Expression & *e*, Representation *r*) [related]

Generator point (const Linear_Expression & *e*, Coefficient_traits::const_reference *d*, Representation *r*) [related]

Generator point (Representation *r*) [related]

Generator point (const Linear_Expression & *e*, Representation *r*) [related]

Generator closure_point (const Linear_Expression & *e*, Coefficient_traits::const_reference *d*, Representation *r*) [related]

Generator closure_point (Representation *r*) [related]

Generator closure_point (const Linear_Expression & *e*, Representation *r*) [related]

bool operator== (const Generator & *x*, const Generator & *y*) [related]

bool operator!= (const Generator & *x*, const Generator & *y*) [related]

template<typename Temp , typename To > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, const Rounding_Dir *dir*, Temp & *tmp0*, Temp & *tmp1*, Temp & *tmp2*) [related]

template<typename Temp , typename To > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, const Rounding_Dir *dir*) [related]

template<typename To > bool rectilinear_distance_assign (Checked_Number< To, Extended_Number_Policy > & *r*, const Generator & *x*, const Generator & *y*, const Rounding_Dir *dir*) [related]

```
template<typename Temp , typename To > bool euclidean_distance_assign ( Checked_Number< To,
Extended_Number_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir
dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
template<typename Temp , typename To > bool euclidean_distance_assign ( Checked_Number< To,
Extended_Number_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir
dir ) [related]
```

```
template<typename To > bool euclidean_distance_assign ( Checked_Number< To, Extended_Number-
_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir dir ) [related]
```

```
template<typename Temp , typename To > bool l_infinity_distance_assign ( Checked_Number< To,
Extended_Number_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir
dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 ) [related]
```

```
template<typename Temp , typename To > bool l_infinity_distance_assign ( Checked_Number< To,
Extended_Number_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir
dir ) [related]
```

```
template<typename To > bool l_infinity_distance_assign ( Checked_Number< To, Extended_Number-
_Policy > & r, const Generator & x, const Generator & y, const Rounding_Dir dir ) [related]
```

```
void swap ( Generator & x, Generator & y ) [related]
```

10.47.6 Member Data Documentation

```
const Representation Parma_Polyhedra_Library::Generator::default_representation = SPARSE [static]
```

The representation used for new Generators.

Note

The copy constructor and the copy constructor with specified size use the representation of the original object, so that it is indistinguishable from the original object.

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

- ppl.hh

10.48 Parma_Polyhedra_Library::Generator_System Class Reference

A system of generators.

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

Public Member Functions

- [Generator_System](#) ([Representation](#) r=default_representation)
Default constructor: builds an empty system of generators.
- [Generator_System](#) (const [Generator](#) &g, [Representation](#) r=default_representation)
Builds the singleton system containing only generator g.
- [Generator_System](#) (const [Generator_System](#) &gs)
- [Generator_System](#) (const [Generator_System](#) &gs, [Representation](#) r)
Copy constructor with specified representation.
- [~Generator_System](#) ()

Destructor.

- `Generator_System & operator= (const Generator_System &y)`

Assignment operator.

- `Representation representation () const`

*Returns the current representation of *this.*

- `void set_representation (Representation r)`

*Converts *this to the specified representation.*

- `dimension_type space_dimension () const`

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

- `void set_space_dimension (dimension_type space_dim)`

Sets the space dimension of the rows in the system to space_dim.

- `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.*

- `void insert (Generator &g, Recycle.Input)`

*Inserts in *this the generator g, stealing its contents and 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 m_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 (Generator_System &x, Generator_System &y)`
- void `swap (Generator_System &x, Generator_System &y)`

10.48.1 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 $(0, 1)^T \in \mathbb{R}^2$.

```
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

$$\{ (x, 0)^T \in \mathbb{R}^2 \mid x \geq 0 \},$$

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.

10.48.2 Constructor & Destructor Documentation

Parma Polyhedra Library::Generator_System::Generator_System (const Generator_System & gs) [inline] Ordinary copy constructor. The new [Generator_System](#) will have the same representation as 'gs'.

10.48.3 Member Function Documentation

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*.

10.48.4 Friends And Related Function Documentation

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 `”, ”`.

void swap (Generator_System & x, Generator_System & y) [related]

void swap (Generator_System & x, Generator_System & y) [related] The documentation for this class was generated from the following file:

- `ppl.hh`

10.49 Parma Polyhedra Library::Generator_System_const_iterator Class Reference

An iterator over a system of generators.

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

Inherits `iterator< std::forward_iterator_tag, Generator, std::ptrdiff_t, const Generator *, const Generator & >`.

Public Member Functions

- [Generator_System_const_iterator \(\)](#)
Default constructor.
- [Generator_System_const_iterator \(const Generator_System_const_iterator &y\)](#)
Ordinary copy constructor.
- [~Generator_System_const_iterator \(\)](#)
Destructor.
- [Generator_System_const_iterator & operator= \(const Generator_System_const_iterator &y\)](#)
Assignment operator.
- [const Generator & operator* \(\) const](#)
Dereference operator.
- [const Generator * operator-> \(\) const](#)
Indirect member selector.
- [Generator_System_const_iterator & operator++ \(\)](#)
Prefix increment operator.
- [Generator_System_const_iterator operator++ \(int\)](#)
Postfix increment operator.
- [bool operator== \(const Generator_System_const_iterator &y\) const](#)
*Returns true if and only if *this and y are identical.*
- [bool operator!= \(const Generator_System_const_iterator &y\) const](#)
*Returns true if and only if *this and y are different.*

10.49.1 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

10.50 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.

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 $y \cdot 2^{\text{exp}}$.
- void [div_2exp_assign](#) ([GMP_Integer](#) &x, const [GMP_Integer](#) &y, unsigned int exp)
Assigns to x the value $y/2^{\text{exp}}$.
- 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.

10.50.1 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://gmplib.org/>

10.50.2 Friends And Related Function Documentation

const mpz_class & raw_value (const GMP_Integer & x) [related] Returns a const reference to the underlying integer value.

mpz_class & raw_value (GMP_Integer & x) [related] Returns a reference to the underlying integer value.

void neg_assign (GMP_Integer & x) [related] Assigns to x its negation.

void neg_assign (GMP_Integer & x , const GMP_Integer & y) [related] Assigns to x the negation of y .

void abs_assign (GMP_Integer & x) [related] Assigns to x its absolute value.

void abs_assign (GMP_Integer & x , const GMP_Integer & y) [related] Assigns to x the absolute value of y .

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 .

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 .

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 $y * s + z * t = x$.

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 .

void add_mul_assign (GMP_Integer & x , const GMP_Integer & y , const GMP_Integer & z) [related] Assigns to x the value $x + y * z$.

void sub_mul_assign (GMP_Integer & x , const GMP_Integer & y , const GMP_Integer & z) [related] Assigns to x the value $x - y * z$.

void mul_2exp_assign (GMP_Integer & x, const GMP_Integer & y, unsigned int exp) [related]
Assigns to x the value $y \cdot 2^{\text{exp}}$.

void div_2exp_assign (GMP_Integer & x, const GMP_Integer & y, unsigned int exp) [related]
Assigns to x the value $y/2^{\text{exp}}$.

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.

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

10.51 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](#) &ggs)
Builds a grid, copying a system of grid generators.
- [Grid](#) ([Grid_Generator_System](#) &ggs, [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* $var' = \frac{expr}{denominator} \pmod{modulus}$.*
- 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* $var' = \frac{expr}{denominator} \pmod{modulus}$.*
- 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* $lhs' = rhs \pmod{modulus}$.*
- 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* $lhs' = rhs \pmod{modulus}$.*
- 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* $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.*
- 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* $\frac{lb.expr}{denominator} \leq var' \leq \frac{ub.expr}{denominator}$.
- 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** *cs_p=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 `m_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.

Related Functions

(Note that these are not member functions.)

- std::ostream & `operator<<` (std::ostream &s, const `Grid` &gr)
Output operator.
- void `swap` (`Grid` &x, `Grid` &y)
Swaps x with y.
- bool `operator==` (const `Grid` &x, const `Grid` &y)
Returns true if and only if x and y are the same grid.
- bool `operator!=` (const `Grid` &x, const `Grid` &y)
Returns true if and only if x and y are different grids.
- bool `operator!=` (const `Grid` &x, const `Grid` &y)
- void `swap` (`Grid` &x, `Grid` &y)

10.51.1 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 % 2 == 0) / 2);  
cgs.insert((y % 2 == 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

$$\{(2, y)^T \in \mathbb{R}^2 \mid y \in \mathbb{R}\}.$$

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 $\{(2, 0)^T\} \subseteq \mathbb{R}^2$.

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 $\{(3, 1, 0, 2)^T\} \subseteq \mathbb{R}^4$, while the resulting grid is $\{(3, 2)^T\} \subseteq \mathbb{R}^2$. Be careful when removing space dimensions *incrementally*: since dimensions are automatically renamed after each application of the `remove_space_dimensions` operator, unexpected results can be obtained. For instance, by using the following code we would obtain a different result:

```
set<Variable> 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 $\{(3, 0)^T\} \subseteq \mathbb{R}^2$: 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.

10.51.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Grid::Grid (dimension_type num_dimensions = 0, Degenerate_Element kind = UNIVERSE) [inline], [explicit] Builds a grid having the specified properties.

Parameters

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the grid;
<i>kind</i>	Specifies whether the universe or the empty grid has to be built.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cgs</i>	The system of congruences defining the grid.
------------	--

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cgs</i>	The system of congruences defining the grid. Its data-structures may be recycled to build the grid.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cs</i>	The system of constraints defining the grid.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if the constraint system <code>cs</code> contains inequality constraints.
<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.

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

<i>cs</i>	The system of constraints defining the grid. Its data-structures may be recycled to build the grid.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the constraint system <code>cs</code> contains inequality constraints.
<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.

Parma_Polyhedra_Library::Grid::Grid (const Grid_Generator_System & *ggs*) [inline], [explicit] Builds a grid, copying a system of grid generators.

The grid inherits the space dimension of the generator system.

Parameters

<i>ggs</i>	The system of generators defining the grid.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.

Parma_Polyhedra_Library::Grid::Grid (Grid_Generator_System & *ggs*, Recycle_Input *dummy*) [inline] Builds a grid, recycling a system of grid generators.

The grid inherits the space dimension of the generator system.

Parameters

<i>ggs</i>	The system of generators defining the grid. Its data-structures may be recycled to build the grid.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.

template<typename Interval > Parma_Polyhedra_Library::Grid::Grid (const Box< Interval > & box, Complexity_Class complexity = ANY_COMPLEXITY) [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

<i>box</i>	The box representing the grid to be built.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>box</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>bd</i>	The BDS representing the grid to be built.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>bd</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>os</i>	The octagonal shape representing the grid to be built.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>os</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>ph</i>	The polyhedron.
<i>complexity</i>	The complexity class.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

Parma_Polyhedra_Library::Grid::Grid (const Grid & *y*, Complexity_Class *complexity* = ANY_COMPLEXITY) Ordinary copy constructor.

The complexity argument is ignored.

10.51.3 Member Function Documentation

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.

bool Parma_Polyhedra_Library::Grid::is_disjoint_from (const Grid & *y*) const Returns `true` if and only if **this* and *y* are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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.

bool Parma_Polyhedra_Library::Grid::constrains (Variable *var*) const Returns `true` if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if the supremum value can be reached in <code>this</code> . Always <code>true</code> when <code>this</code> bounds <code>expr</code> . Present for interface compatibility with class Polyhedron , where closure points can result in a value of <code>false</code> .

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if the supremum value can be reached in <code>this</code> . Always <code>true</code> when <code>this</code> bounds <code>expr</code> . Present for interface compatibility with class Polyhedron , where closure points can result in a value of <code>false</code> ;
<i>point</i>	When maximization succeeds, will be assigned a point where <code>expr</code> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if the is the infimum value can be reached in <i>this</i> . Always <code>true</code> when <i>this</i> bounds <i>expr</i> . Present for interface compatibility with class Polyhedron , where closure points can result in a value of <code>false</code> .

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if the is the infimum value can be reached in <i>this</i> . Always <code>true</code> when <i>this</i> bounds <i>expr</i> . Present for interface compatibility with class Polyhedron , where closure points can result in a value of <code>false</code> ;
<i>point</i>	When minimization succeeds, will be assigned a point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression for which the frequency is needed;
<i>freq_n</i>	The numerator of the maximum frequency of <i>expr</i> ;
<i>freq_d</i>	The denominator of the maximum frequency of <i>expr</i> ;
<i>val_n</i>	The numerator of them value of <i>expr</i> at a point in the grid that is closest to zero;
<i>val_d</i>	The denominator of a value of <i>expr</i> at a point in the grid that is closest to zero;

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

bool Parma_Polyhedra_Library::Grid::contains (const Grid & *y*) const Returns `true` if and only if **this* contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

bool Parma_Polyhedra_Library::Grid::strictly_contains (const Grid & y) const [inline] Returns *true* if and only if **this* strictly contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>check_not_empty</i>	<i>true</i> if and only if, in addition to checking the invariants, <i>*this</i> 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.

void Parma_Polyhedra_Library::Grid::add_congruence (const Congruence & cg) [inline] Adds a copy of congruence *cg* to **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible, or if <i>*this</i> is an empty grid and <i>g</i> is not a point.
------------------------------	--

void Parma_Polyhedra_Library::Grid::add_congruences (const Congruence_System & cgs) [inline] Adds a copy of each congruence in *cgs* to **this*.

Parameters

<i>cgs</i>	Contains the congruences that will be added to the system of congruences of <i>*this</i> .
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::Grid::add_recycled_congruences (Congruence_System & cgs) Adds the congruences in *cgs* to **this*.

Parameters

<i>cgs</i>	The congruence system to be added to <i>*this</i> . The congruences in <i>cgs</i> may be recycled.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> 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.

void Parma_Polyhedra_Library::Grid::add_constraint (const Constraint & c) [inline] Adds to **this* a congruence equivalent to constraint *c*.

Parameters

<i>c</i>	The constraint to be added.
----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible or if constraint <i>c</i> is not optimally supported by the grid domain.
------------------------------	---

void Parma_Polyhedra_Library::Grid::add_constraints (const Constraint_System & cs) Adds to **this* congruences equivalent to the constraints in *cs*.

Parameters

<i>cs</i>	The constraints to be added.
-----------	------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a constraint which is not optimally supported by the grid domain.
------------------------------	--

void Parma_Polyhedra_Library::Grid::add_recycled_constraints (Constraint_System & cs) [inline] Adds to **this* congruences equivalent to the constraints in *cs*.

Parameters

<i>cs</i>	The constraints to be added. They may be recycled.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a constraint which 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.

void Parma_Polyhedra_Library::Grid::refine_with_congruence (const Congruence & cg) [inline] Uses a copy of the congruence *cg* to refine **this*.

Parameters

<i>cg</i>	The congruence used.
-----------	----------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

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

<i>cgs</i>	The congruences used.
------------	-----------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::Grid::refine_with_constraint (const Constraint & *c*) Uses a copy of the constraint *c* to refine **this*.

Parameters

<i>c</i>	The constraint used. If it is not an equality, it will be ignored
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::Grid::refine_with_constraints (const Constraint_System & *cs*) Uses a copy of the constraints in *cs* to refine **this*.

Parameters

<i>cs</i>	The constraints used. Constraints that are not equalities are ignored.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

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

<i>gs</i>	Contains the generators that will be added to the system of generators of <i>*this</i> .
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>gs</i> are dimension-incompatible, or if <i>*this</i> is empty and the system of generators <i>gs</i> is not empty, but has no points.
------------------------------	--

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

<i>gs</i>	The generator system to be added to <i>*this</i> . The generators in <i>gs</i> may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>gs</i> 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.

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

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

void Parma_Polyhedra_Library::Grid::intersection_assign (const Grid & *y*) Assigns to **this* the intersection of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::Grid::upper_bound_assign (const Grid & *y*) Assigns to **this* the least upper bound of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>var</i>	The variable to which the affine expression is substituted;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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](#) $\text{var}' = \frac{\text{expr}}{\text{denominator}} \pmod{\text{modulus}}$.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol where EQUAL is the symbol for a congruence relation;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression. Optional argument with an automatic value of one;
<i>modulus</i>	The modulus of the congruence lhs %= rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of zero.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>this</i> .
------------------------------	---

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](#) $var' = \frac{expr}{denominator} \pmod{modulus}$.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol where EQUAL is the symbol for a congruence relation;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression. Optional argument with an automatic value of one;
<i>modulus</i>	The modulus of the congruence lhs %= rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of zero.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>this</i> .
------------------------------	---

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](#) $lhs' = rhs \pmod{modulus}$.

Parameters

<i>lhs</i>	The left hand side affine expression.
<i>relsym</i>	The relation symbol where EQUAL is the symbol for a congruence relation;
<i>rhs</i>	The right hand side affine expression.
<i>modulus</i>	The modulus of the congruence lhs %= rhs. A modulus of zero indicates lhs == rhs. Optional argument with an automatic value of zero.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> .
------------------------------	--

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](#) $lhs' = rhs \pmod{modulus}$.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol where EQUAL is the symbol for a congruence relation;
<i>rhs</i>	The right hand side affine expression;
<i>modulus</i>	The modulus of the congruence $lhs \% = rhs$. A modulus of zero indicates $lhs == rhs$. Optional argument with an automatic value of zero.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> .
------------------------------	--

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](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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 * cs_p = 0, unsigned complexity_threshold = 16, bool wrap_individually = true)
Wraps the specified dimensions of the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	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.
<i>complexity_threshold</i>	A precision parameter of the wrapping operator . This argument is for compatibility with wrap_assign() for the other domains and is ignored.
<i>wrap_individually</i>	true if the dimensions should be wrapped individually. As wrapping dimensions collectively does not improve the precision, this argument is ignored.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with one of the Variable objects contained in <i>vars</i> or with <i>*cs_p</i> .
------------------------------	---

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.

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

<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.
-------------------	---

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

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

<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.
-------------------	---

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 <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

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 <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

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 <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if *this and y are dimension-incompatible.
------------------------------	---

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 <i>must</i> 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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	--

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

<i>y</i>	A grid that <i>must</i> be contained in <i>*this</i> ;
<i>cgs</i>	The system of congruences used to improve the widened grid;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	--

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

<i>y</i>	A grid that <i>must</i> be contained in <i>*this</i> ;
<i>cgs</i>	The system of congruences used to improve the widened grid;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	--

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

<i>m</i>	The number of dimensions to add.
----------	----------------------------------

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> new space dimensions would cause the vector space to exceed dimension max_space_dimension() .
--------------------------	---

The new space dimensions will be those having the highest indexes in the new grid, which is characterized by a system of congruences in which the variables which are the new dimensions can have any value. For instance, when starting from the grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

$$\{ (x, y, z)^T \in \mathbb{R}^3 \mid (x, y)^T \in \mathcal{L} \}.$$

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

<i>m</i>	The number of space dimensions to add.
----------	--

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .
--------------------------	--

The new space dimensions will be those having the highest indexes in the new grid, which is characterized by a system of congruences in which the variables running through the new dimensions are all constrained to be equal to 0. For instance, when starting from the grid $\mathcal{L} \subseteq \mathbb{R}^2$ and adding a third space dimension, the result will be the grid

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

void Parma_Polyhedra_Library::Grid::concatenate_assign (const Grid & y) Assigns to **this* the concatenation of **this* and *y*, taken in this order.

Exceptions

<i>std::length_error</i>	Thrown if the concatenation would cause the vector space to exceed dimension <code>max_space_dimension()</code> .
--------------------------	---

void Parma_Polyhedra_Library::Grid::remove_space_dimensions (const Variables.Set & vars) Removes all the specified dimensions from the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be removed.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <code>new_dimensions</code> is greater than the space dimension of <i>*this</i> .
------------------------------	---

template<typename Partial_Function> void Parma_Polyhedra_Library::Grid::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a [partial function](#).

If *pfunc* maps only some of the dimensions of **this* then the rest will be projected away.

If the highest dimension mapped to by *pfunc* is higher than the highest dimension in **this* then the number of dimensions in *this* will be increased to the highest dimension mapped to by *pfunc*.

Parameters

<i>pfunc</i>	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](#).

void Parma_Polyhedra_Library::Grid::expand_space_dimension (Variable var, dimension_type m) Creates m copies of the space dimension corresponding to `var`.

Parameters

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding m new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

void Parma_Polyhedra_Library::Grid::fold_space_dimensions (const Variables.Set & vars, Variable dest) Folds the space dimensions in `vars` into `dest`.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> or with one of the Variable objects contained in <code>vars</code> . Also thrown if <code>dest</code> is contained in <code>vars</code> .
------------------------------	---

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.

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 $x.hash_code() == y.hash_code()$.

10.51.4 Friends And Related Function Documentation

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`.

void swap (Grid & x, Grid & y) [related] Swaps `x` with `y`.

bool operator== (const Grid & x, const Grid & y) [related] 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.

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.

bool operator!= (const Grid & x, const Grid & y) [related]

void swap (Grid & x, Grid & y) [related] The documentation for this class was generated from the following file:

- `ppl.hh`

10.52 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`.*

10.52.1 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.

10.52.2 Member Function Documentation

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

10.53 Parma_Polyhedra_Library::Grid_Generator Class Reference

A grid line, parameter or grid point.

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

Public Types

- enum [Kind](#)
The possible kinds of [Grid_Generator](#) objects.
- enum [Type](#) { [LINE](#), [PARAMETER](#), [POINT](#) }
The generator type.
- typedef Expression_Hide_Last
< Expression_Hide_Inhomo
< [Linear_Expression](#) > > [expr_type](#)
The type of the (adapted) internal expression.

Public Member Functions

- [Grid_Generator](#) ([Representation](#) r=default_representation)
Returns the origin of the zero-dimensional space \mathbb{R}^0 .
- [Grid_Generator](#) (const [Grid_Generator](#) &g)
- [Grid_Generator](#) (const [Grid_Generator](#) &g, [Representation](#) r)
Copy constructor with specified representation.
- [Grid_Generator](#) (const [Grid_Generator](#) &g, [dimension_type](#) space_dim)
- [Grid_Generator](#) (const [Grid_Generator](#) &g, [dimension_type](#) space_dim, [Representation](#) r)
Copy constructor with specified space dimension and representation.
- [~Grid_Generator](#) ()
Destructor.
- [Grid_Generator](#) & operator= (const [Grid_Generator](#) &g)
Assignment operator.
- [Representation](#) representation () const

- Returns the current representation of *this.*

 - void `set_representation` (`Representation` r)

*Converts *this to the specified representation.*
- `dimension_type space_dimension` () const

*Returns the dimension of the vector space enclosing *this.*
- void `set_space_dimension` (`dimension_type` space_dim)
- void `swap_space_dimensions` (`Variable` v1, `Variable` v2)
- Swaps the coefficients of the variables v1 and v2 .*
- bool `remove_space_dimensions` (const `Variables_Set` &vars)
- Removes all the specified dimensions from the grid generator.*
- void `permute_space_dimensions` (const std::vector< `Variable` > &cycle)
- Permutes the space dimensions of the grid generator.*
- void `shift_space_dimensions` (`Variable` v, `dimension_type` n)
- `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 identical to y.*
- bool `all_homogeneous_terms_are_zero` () const
- Returns true if and only if all the homogeneous terms of *this are 0.*
- 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.*
- void `m_swap` (Grid_Generator &y)
*Swaps `*this` with `y`.*
- void `scale_to_divisor` (Coefficient_traits::const_reference d)
*Scales `*this` to be represented with a divisor of `d` (if `*this` is a parameter or point). Does nothing at all on lines.*
- void `set_divisor` (Coefficient_traits::const_reference d)
*Sets the divisor of `*this` to `d`.*
- `expr_type expression` () const
Partial read access to the (adapted) internal expression.

Static Public Member Functions

- static Grid_Generator `grid_line` (const Linear_Expression &e, Representation r=default_representation)
Returns the line of direction `e`.
- static Grid_Generator `parameter` (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=default_representation)
- static Grid_Generator `parameter` (Representation r)
Returns the parameter of direction and size `Linear_Expression::zero()`.
- static Grid_Generator `parameter` (const Linear_Expression &e, Representation r)
Returns the parameter of direction and size `e`.
- static Grid_Generator `grid_point` (const Linear_Expression &e=Linear_Expression::zero(), Coefficient_traits::const_reference d=Coefficient_one(), Representation r=default_representation)
Returns the point at `e / d`.
- static Grid_Generator `grid_point` (Representation r)
Returns the point at `e`.
- static Grid_Generator `grid_point` (const Linear_Expression &e, Representation r)
Returns the point at `e`.
- 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 .

Static Public Attributes

- static const Representation `default_representation` = SPARSE
The representation used for new `Grid_Generators`.

Related Functions

(Note that these are not member functions.)

- `std::ostream & operator<<` (`std::ostream &s`, `const Grid_Generator &g`)
Output operator.
- `void swap` (`Grid_Generator &x`, `Grid_Generator &y`)
Swaps x with y .
- `Grid_Generator grid_line` (`const Linear_Expression &e`, `Representation r=Grid_Generator::default_representation`)
Shorthand for `Grid_Generator::grid_line(const Linear_Expression& e, Representation r)`.
- `Grid_Generator parameter` (`const Linear_Expression &e=Linear_Expression::zero()`, `Coefficient_traits::const_reference d=Coefficient_one()`, `Representation r=Grid_Generator::default_representation`)
Shorthand for `Grid_Generator::parameter(const Linear_Expression& e, Coefficient_traits::const_reference d, Representation r)`.
- `Grid_Generator parameter` (`Representation r`)
Shorthand for `Grid_Generator::parameter(Representation r)`.
- `Grid_Generator parameter` (`const Linear_Expression &e`, `Representation r`)
Shorthand for `Grid_Generator::parameter(const Linear_Expression& e, Representation r)`.
- `Grid_Generator grid_point` (`const Linear_Expression &e=Linear_Expression::zero()`, `Coefficient_traits::const_reference d=Coefficient_one()`, `Representation r=Grid_Generator::default_representation`)
Shorthand for `Grid_Generator::grid_point(const Linear_Expression& e, Coefficient_traits::const_reference d, Representation r)`.
- `Grid_Generator grid_point` (`Representation r`)
Shorthand for `Grid_Generator::grid_point(Representation r)`.
- `Grid_Generator grid_point` (`const Linear_Expression &e`, `Representation r`)
Shorthand for `Grid_Generator::grid_point(const Linear_Expression& e, Representation r)`.
- `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.
- `bool operator==` (`const Grid_Generator &x`, `const Grid_Generator &y`)
- `bool operator!=` (`const Grid_Generator &x`, `const Grid_Generator &y`)
- `Grid_Generator grid_line` (`const Linear_Expression &e`, `Representation r`)
- `Grid_Generator parameter` (`const Linear_Expression &e`, `Coefficient_traits::const_reference d`, `Representation r`)
- `Grid_Generator parameter` (`Representation r`)
- `Grid_Generator parameter` (`const Linear_Expression &e`, `Representation r`)
- `Grid_Generator grid_point` (`const Linear_Expression &e`, `Coefficient_traits::const_reference d`, `Representation r`)
- `Grid_Generator grid_point` (`Representation r`)
- `Grid_Generator grid_point` (`const Linear_Expression &e`, `Representation r`)
- `void swap` (`Grid_Generator &x`, `Grid_Generator &y`)

10.53.1 Detailed Description

A grid line, parameter or grid point.

An object of the class `Grid.Generator` is one of the following:

- a grid_line $l = (a_0, \dots, a_{n-1})^T$;
- a parameter $q = (\frac{a_0}{d}, \dots, \frac{a_{n-1}}{d})^T$;
- a grid_point $p = (\frac{a_0}{d}, \dots, \frac{a_{n-1}}{d})^T$;

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 $p = (1, -1, -1)^T \in \mathbb{R}^3$ 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 $p = (1, 0, 2)^T \in \mathbb{R}^3$:

```
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 $0 \in \mathbb{R}^3$ 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 \mathbf{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 $\mathbf{p1} = (-1.5, 3.2, 2.1)^T \in \mathbb{R}^3$ 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 $\mathbf{q} = (1, 0, 2)^T \in \mathbb{R}^3$ 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 $\mathbf{q} = (-1.5, 3.2, 2.1)^T \in \mathbb{R}^3$ 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 $\mathbf{g1}$ is a grid point having coordinates $(a_0, \dots, a_{n-1})^T$, we construct the parameter $\mathbf{g2}$ having coordinates $(a_0, 2a_1, \dots, (i+1)a_i, \dots, na_{n-1})^T$.

```
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.

10.53.2 Member Enumeration Documentation

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.

10.53.3 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Grid_Generator::Grid_Generator (const Grid_Generator & g) [inline]
Ordinary copy constructor. The new [Grid_Generator](#) will have the same representation as g.

Parma_Polyhedra_Library::Grid_Generator::Grid_Generator (const Grid_Generator & g, dimension_type space_dim) [inline] Copy constructor with specified space dimension. The new [Grid_Generator](#) will have the same representation as g.

10.53.4 Member Function Documentation

static Grid_Generator Parma_Polyhedra_Library::Grid_Generator::grid_line (const Linear_Expression & e, Representation r = default_representation) [static] Returns the line of direction e.

Exceptions

<i>std::invalid_argument</i>	Thrown if the homogeneous part of e represents the origin of the vector space.
------------------------------	--

static Grid_Generator Parma_Polyhedra_Library::Grid_Generator::parameter (const Linear_Expression & e = Linear_Expression::zero (), Coefficient_traits::const_reference d = Coefficient_one (), Representation r = default_representation) [static] Returns the parameter of direction e and size e/d, with the same representation as e.

Both e and d are optional arguments, with default values [Linear_Expression::zero\(\)](#) and [Coefficient_one\(\)](#), respectively.

Exceptions

<i>std::invalid_argument</i>	Thrown if d is zero.
------------------------------	----------------------

static Grid_Generator Parma_Polyhedra_Library::Grid_Generator::grid_point (const Linear_Expression & e = Linear_Expression::zero (), Coefficient_traits::const_reference d = Coefficient_one (), Representation r = default_representation) [static] Returns the point at e / d.

Both e and d are optional arguments, with default values [Linear_Expression::zero\(\)](#) and [Coefficient_one\(\)](#), respectively.

Exceptions

<i>std::invalid_argument</i>	Thrown if d is zero.
------------------------------	----------------------

void Parma_Polyhedra_Library::Grid_Generator::set_space_dimension (dimension_type space_dim) [inline] Sets the dimension of the vector space enclosing *this to space_dim.

bool Parma_Polyhedra_Library::Grid_Generator::remove_space_dimensions (const Variables_Set & vars) Removes all the specified dimensions from the grid generator.

The space dimension of the variable with the highest space dimension in `vars` must be at most the space dimension of `this`.

Always returns `true`. The return value is needed for compatibility with the [Generator](#) class.

void Parma_Polyhedra_Library::Grid_Generator::shift_space_dimensions (Variable v, dimension_type n) [inline] Shift by `n` positions the coefficients of variables, starting from the coefficient of `v`. This increases the space dimension by `n`.

Coefficient_traits::const_reference Parma_Polyhedra_Library::Grid_Generator::coefficient (Variable v) const [inline] Returns the coefficient of `v` in `*this`.

Exceptions

<i>std::invalid_argument</i>	Thrown if the index of <code>v</code> is greater than or equal to the space dimension of <code>*this</code> .
------------------------------	---

Coefficient_traits::const_reference Parma_Polyhedra_Library::Grid_Generator::divisor () const [inline] Returns the divisor of `*this`.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is a line.
------------------------------	---

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.

bool Parma_Polyhedra_Library::Grid_Generator::is_equal_to (const Grid_Generator & y) const Returns `true` if `*this` is identical to `y`.

This is faster than [is_equivalent_to\(\)](#), but it may return ‘false’ even for equivalent generators.

void Parma_Polyhedra_Library::Grid_Generator::scale_to_divisor (Coefficient_traits::const_reference d) Scales `*this` to be represented with a divisor of `d` (if `*this` is a parameter or point). Does nothing at all on lines.

It is assumed that `d` is a multiple of the current divisor and different from zero. The behavior is undefined if the assumption does not hold.

void Parma_Polyhedra_Library::Grid_Generator::set_divisor (Coefficient_traits::const_reference d) [inline] Sets the divisor of `*this` to `d`.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is a line.
------------------------------	---

10.53.5 Friends And Related Function Documentation

std::ostream & operator<< (std::ostream & s, const Grid_Generator & g) [related] Output operator.

void swap (Grid_Generator & x, Grid_Generator & y) [related] Swaps `x` with `y`.

Grid_Generator parameter (Representation r) [related] Shorthand for [Grid_Generator::parameter\(-Representation r\)](#).

Grid_Generator grid_point (Representation *r*) [related] Shorthand for [Grid_Generator::grid_point\(Representation *r*\)](#).

bool operator== (const Grid_Generator & *x*, const Grid_Generator & *y*) [related] Returns `true` if and only if *x* is equivalent to *y*.

bool operator!= (const Grid_Generator & *x*, const Grid_Generator & *y*) [related] Returns `true` if and only if *x* is not equivalent to *y*.

std::ostream & operator<< (std::ostream & *s*, const Grid_Generator::Type & *t*) [related] Output operator.

bool operator== (const Grid_Generator & *x*, const Grid_Generator & *y*) [related]

bool operator!= (const Grid_Generator & *x*, const Grid_Generator & *y*) [related]

Grid_Generator grid_line (const Linear_Expression & *e*, Representation *r*) [related]

Grid_Generator parameter (const Linear_Expression & *e*, Coefficient_traits::const_reference *d*, Representation *r*) [related]

Grid_Generator parameter (Representation *r*) [related]

Grid_Generator parameter (const Linear_Expression & *e*, Representation *r*) [related]

Grid_Generator grid_point (const Linear_Expression & *e*, Coefficient_traits::const_reference *d*, Representation *r*) [related]

Grid_Generator grid_point (Representation *r*) [related]

Grid_Generator grid_point (const Linear_Expression & *e*, Representation *r*) [related]

void swap (Grid_Generator & *x*, Grid_Generator & *y*) [related]

10.53.6 Member Data Documentation

const Representation Parma_Polyhedra_Library::Grid_Generator::default_representation = SPARSE [static] The representation used for new Grid_Generators.

Note

The copy constructor and the copy constructor with specified size use the representation of the original object, so that it is indistinguishable from the original object.

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

- `ppl.hh`

10.54 Parma Polyhedra Library::Grid_Generator_System Class Reference

A system of grid generators.

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

Classes

- class `const_iterator`

An iterator over a system of grid generators.

Public Member Functions

- `Grid_Generator_System` (`Representation` `r=default_representation`)
Default constructor: builds an empty system of generators.
- `Grid_Generator_System` (`const Grid_Generator` `&g`, `Representation` `r=default_representation`)
Builds the singleton system containing only generator `g`.
- `Grid_Generator_System` (`dimension_type` `dim`, `Representation` `r=default_representation`)
Builds an empty system of generators of dimension `dim`.
- `Grid_Generator_System` (`const Grid_Generator_System` `&gs`)
- `Grid_Generator_System` (`const Grid_Generator_System` `&gs`, `Representation` `r`)
Copy constructor with specified representation.
- `~Grid_Generator_System` ()
Destructor.
- `Grid_Generator_System` `& operator=` (`const Grid_Generator_System` `&y`)
Assignment operator.
- `Representation` `representation` () `const`
*Returns the current representation of `*this`.*
- `void` `set_representation` (`Representation` `r`)
*Converts `*this` to the specified representation.*
- `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` `insert` (`Grid_Generator` `&g`, `Recycle_Input`)
*Inserts into `*this` the generator `g`, increasing the number of space dimensions if needed.*
- `void` `insert` (`Grid_Generator_System` `&gs`, `Recycle_Input`)
*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 `m_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()`.

Related Functions

(Note that these are not member functions.)

- std::ostream & `operator<<` (std::ostream &s, const `Grid_Generator_System` &gs)

Output operator.
- void `swap` (`Grid_Generator_System` &x, `Grid_Generator_System` &y)

Swaps x with y.
- bool `operator==` (const `Grid_Generator_System` &x, const `Grid_Generator_System` &y)

Returns true if and only if x and y are identical.
- bool `operator==` (const `Grid_Generator_System` &x, const `Grid_Generator_System` &y)
- void `swap` (`Grid_Generator_System` &x, `Grid_Generator_System` &y)

10.54.1 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 $(0, 1)^T \in \mathbb{R}^2$.

```
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

$$\{ (x, 0)^T \in \mathbb{R}^2 \mid x \pmod{1} = 0 \},$$

```
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 gsl;
gsl.insert(grid_point(3*x + 3*y));
gsl.insert(parameter(0*x + 3*y));
gsl.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.

10.54.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Grid_Generator_System::Grid_Generator_System (const Grid_Generator_System &gs) [inline] Ordinary copy constructor. The new [Grid_Generator_System](#) will have the same representation as 'gs'.

10.54.3 Member Function Documentation

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*.

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*.

10.54.4 Friends And Related Function Documentation

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 ", ".

void swap (Grid_Generator_System &x, Grid_Generator_System &y) [related] Swaps *x* with *y*.

bool operator== (const Grid_Generator_System &x, const Grid_Generator_System &y) [related] Returns *true* if and only if *x* and *y* are identical.

bool operator== (const Grid_Generator_System &x, const Grid_Generator_System &y) [related]

void swap (Grid_Generator_System & x, Grid_Generator_System & y) [**related**] The documentation for this class was generated from the following file:

- ppl.hh

10.55 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.*

10.55.1 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](#).

10.55.2 Member Function Documentation

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:

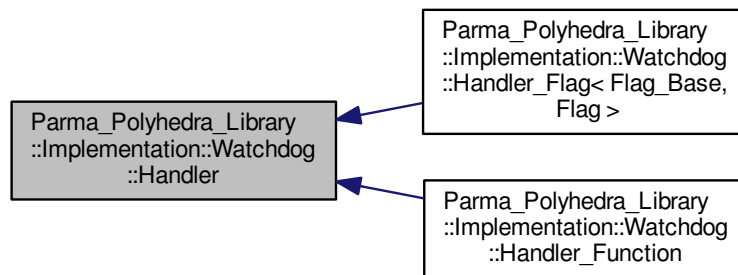
- `ppl.hh`

10.56 Parma_Polyhedra_Library::Implementation::Watchdog::Handler Class Reference

Abstract base class for handlers of the watchdog events.

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

Inheritance diagram for Parma_Polyhedra_Library::Implementation::Watchdog::Handler:



Public Member Functions

- virtual void `act` () const =0

Does the job.

- virtual `~Handler` ()

Virtual destructor.

10.56.1 Detailed Description

Abstract base class for handlers of the watchdog events.

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

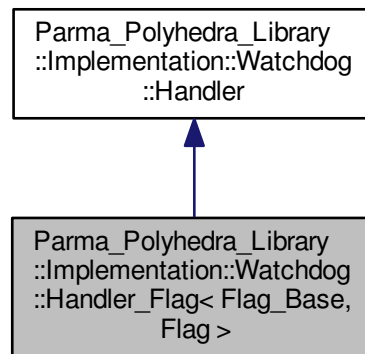
- `ppl.hh`

10.57 Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag<Flag_Base, Flag> Class Template Reference

A kind of `Handler` that installs a flag onto a flag-holder.

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

Inheritance diagram for Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag >:



Public Member Functions

- [Handler_Flag](#) (const Flag_Base *volatile &holder, Flag &flag)
Constructor with a given function.
- virtual void [act](#) () const
Does its job: installs the flag onto the holder, if a flag with an higher priority has not already been installed.

10.57.1 Detailed Description

template<typename Flag_Base, typename Flag>class Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag >

A kind of [Handler](#) that installs a flag onto a flag-holder.

The template class [Handler_Flag](#)<Flag_Base, Flag> is an handler whose job is to install a flag onto an *holder* for the flag. The flag is of type Flag and the holder is a (volatile) pointer to Flag_Base. Installing the flag onto the holder means making the holder point to the flag, so that it must be possible to assign a value of type Flag* to an entity of type Flag_Base*. The class Flag must provide the method

```
int priority() const
```

returning an integer priority associated to the flag.

The handler will install its flag onto the holder only if the holder is empty, namely, it is the null pointer, or if the holder holds a flag of strictly lower priority.

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

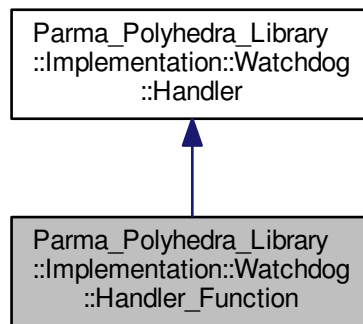
- ppl.hh

10.58 Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function Class Reference

A kind of [Handler](#) calling a given function.

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

Inheritance diagram for Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function:



Public Member Functions

- [Handler_Function](#) (void(*const function)())
Constructor with a given function.
- virtual void [act](#) () const
Does its job: calls the embedded function.

10.58.1 Detailed Description

A kind of [Handler](#) calling a given function.

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

- ppl.hh

10.59 Parma_Polyhedra_Library::Integer_Constant< Target > Class Template Reference

An integer constant concrete expression.

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

10.59.1 Detailed Description

```
template<typename Target>class Parma_Polyhedra_Library::Integer_Constant< Target >
```

An integer constant concrete expression.

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

- ppl.hh

10.60 Parma_Polyhedra_Library::Integer_Constant_Common< Target > Class Template Reference

Base class for integer constant concrete expressions.

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

10.60.1 Detailed Description

```
template<typename Target>class Parma_Polyhedra_Library::Integer_Constant_Common< Target
>
```

Base class for integer constant concrete expressions.

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

- ppl.hh

10.61 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, and Info.

Public Member Functions

- void [m_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, L_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)), L_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, L_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 `t0` 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 `t0` 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](#) ([Interval](#)< Boundary, Info > &x, [Interval](#)< Boundary, Info > &y)
Swaps `x` with `y`.
- `template<typename Boundary , typename Info >`
`void` [swap](#) ([Interval](#)< Boundary, Info > &x, [Interval](#)< Boundary, Info > &y)

10.61.1 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;

10.61.2 Member Function Documentation

```
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 ) Assigns to *this a meet-preserving simplifica-
tion of *this with respect to y.
```

Returns

false if and only if the meet of `*this` and `y` is empty.

```
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 τ_0 interval is restricted to become, upon successful exit, the smallest interval of its type that contains the set

$$\{a \in \mathbf{to} \mid \exists b \in \mathbf{x} . a \text{ rel } b\}.$$

Returns

???

```
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 τ_0 interval is restricted to become, upon successful exit, the smallest interval of its type that contains the set

$$\{a \in \mathbf{to} \mid \forall b \in \mathbf{x} : a \text{ rel } b\}.$$

Returns

???

```

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*y|,xu*yu|xu*y|,xl*y| | xu*y|,xu*yu | +-----+-----+-----+
-----+ | xu < 0 |xl*yu,xu*y| |xu*yu,xl*y| | xl*yu,xl*y| | +-----+-----+-----+
|xl<0 xu>0|xl*yu,xu*yu|xu*y|,xl*y| |min(xl*yu,xu*y|),| | | |max(xl*y|,xu*yu) | +-----+-----+
-----+-----+

```

```
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| +-----+-----+-----+
```

10.61.3 Friends And Related Function Documentation

template<typename Boundary , typename Info > void swap (Interval< Boundary, Info > &x,
Interval< Boundary, Info > &y) [**related**] Swaps x with y.

template<typename Boundary , typename Info > void swap (Interval< Boundary, Info > &x,
Interval< Boundary, Info > &y) [**related**] The documentation for this class was generated
from the following file:

- ppl.hh

10.62 Parma_Polyhedra_Library::CO_Tree::iterator Class Reference

An iterator on the tree elements, ordered by key.

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

Public Member Functions

- [iterator](#) ()
Constructs an invalid iterator.
- [iterator](#) (CO_Tree &tree)
Constructs an iterator pointing to first element of the tree.
- [iterator](#) (CO_Tree &tree, [dimension_type](#) i)
Constructs an iterator pointing to the i-th node.
- [iterator](#) (const tree_iterator &itr)
The constructor from a tree_iterator.
- [iterator](#) (const [iterator](#) &itr)
The copy constructor.
- void [m_swap](#) ([iterator](#) &itr)
*Swaps itr with *this.*
- [iterator](#) & [operator=](#) (const [iterator](#) &itr)
*Assigns itr to *this.*
- [iterator](#) & [operator=](#) (const tree_iterator &itr)
*Assigns itr to *this.*
- [iterator](#) & [operator++](#) ()
Navigates to the next element in the tree.
- [iterator](#) & [operator--](#) ()
Navigates to the previous element in the tree.
- [iterator](#) [operator++](#) (int)
Navigates to the next element in the tree.
- [iterator](#) [operator--](#) (int)
Navigates to the previous element in the tree.
- [data_type](#) & [operator*](#) ()

- *Returns the current element.*
data_type.const_reference operator* () const
- *Returns the current element.*
dimension_type index () const
- *Returns the index of the element pointed to by *this.*
bool operator== (const iterator &x) const
- *Compares *this with x.*
bool operator!= (const iterator &x) const
- *Compares *this with x.*

10.62.1 Detailed Description

An iterator on the tree elements, ordered by key.

Iterator increment and decrement operations are $O(1)$ time. These iterators are invalidated by operations that add or remove elements from the tree.

10.62.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::CO_Tree::iterator::iterator () [inline] Constructs an invalid iterator.

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::iterator::iterator (CO_Tree &tree) [inline],[explicit] Constructs an iterator pointing to first element of the tree.

Parameters

<i>tree</i>	The tree to which the new iterator will point to.
-------------	---

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::iterator::iterator (CO_Tree &tree, dimension_type i) [inline] Constructs an iterator pointing to the i-th node.

Parameters

<i>tree</i>	The tree to which the new iterator will point to.
<i>i</i>	The index of the element in <i>tree</i> to which the new iterator will point to.

The i-th node must be a node with a value or end().

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::iterator::iterator (const tree_iterator &itr) [inline],[explicit] The constructor from a tree_iterator.

Parameters

<i>itr</i>	The tree_iterator that will be converted into an iterator.
------------	--

This is meant for use by CO_Tree only. This is not private to avoid the friend declaration.

This constructor takes $O(1)$ time.

Parma_Polyhedra_Library::CO_Tree::iterator::iterator (const iterator &itr) [inline] The copy constructor.

Parameters

<i>itr</i>	The iterator that will be copied.
------------	-----------------------------------

This constructor takes $O(1)$ time.

10.62.3 Member Function Documentation

void Parma_Polyhedra_Library::CO_Tree::iterator::m_swap (iterator & *itr*) [inline] Swaps *itr* with *this.

Parameters

<i>itr</i>	The iterator that will be swapped with *this.
------------	---

This method takes $O(1)$ time.

CO_Tree::iterator & Parma_Polyhedra_Library::CO_Tree::iterator::operator= (const iterator & *itr*) [inline] Assigns *itr* to *this.

Parameters

<i>itr</i>	The iterator that will be assigned into *this.
------------	--

This method takes $O(1)$ time.

CO_Tree::iterator & Parma_Polyhedra_Library::CO_Tree::iterator::operator= (const tree_iterator & *itr*) [inline] Assigns *itr* to *this.

Parameters

<i>itr</i>	The iterator that will be assigned into *this.
------------	--

This method takes $O(1)$ time.

CO_Tree::iterator & Parma_Polyhedra_Library::CO_Tree::iterator::operator++ () [inline]

Navigates to the next element in the tree.

This method takes $O(1)$ time.

CO_Tree::iterator & Parma_Polyhedra_Library::CO_Tree::iterator::operator-- () [inline]

Navigates to the previous element in the tree.

This method takes $O(1)$ time.

CO_Tree::iterator Parma_Polyhedra_Library::CO_Tree::iterator::operator++ (int) [inline]

Navigates to the next element in the tree.

This method takes $O(1)$ time.

CO_Tree::iterator Parma_Polyhedra_Library::CO_Tree::iterator::operator-- (int) [inline]

Navigates to the previous element in the tree.

This method takes $O(1)$ time.

dimension_type Parma_Polyhedra_Library::CO_Tree::iterator::index () const [inline] Returns the index of the element pointed to by *this.

Returns

the index of the element pointed to by *this.

bool Parma_Polyhedra_Library::CO_Tree::iterator::operator== (const iterator & *x*) const [inline]

Compares *this with *x*.

Parameters

x	The iterator that will be compared with <code>*this</code> .
-----	--

bool Parma_Polyhedra_Library::CO_Tree::iterator::operator!=(const iterator & x)const **[inline]**

Compares `*this` with x .

Parameters

x	The iterator that will be compared with <code>*this</code> .
-----	--

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

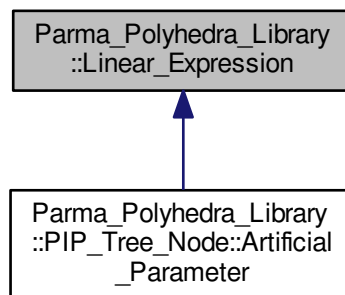
- ppl.hh

10.63 Parma_Polyhedra_Library::Linear_Expression Class Reference

A linear expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Linear_Expression:



Classes

- class [const_iterator](#)

Public Member Functions

- [Linear_Expression](#) ([Representation](#) r =default_representation)
Default constructor: returns a copy of [Linear_Expression::zero\(\)](#).
- [Linear_Expression](#) (const [Linear_Expression](#) & e)
Ordinary copy constructor.
- [Linear_Expression](#) (const [Linear_Expression](#) & e , [Representation](#) r)
Copy constructor that takes also a Representation.
- `template<typename LE_Adapter >`
[Linear_Expression](#) (const LE_Adapter & e , `typename Enable_If< Is_Same_Or_Derived< Expression-Adapter_Base, LE_Adapter >::value, void * >::type=0`)
Copy constructor from a linear expression adapter.

- `template<typename LE_Adapter >`
`Linear_Expression` (const LE_Adapter &e, `Representation` r, typename Enable_If< Is_Same_Or_Derived< Expression_Adapter_Base, LE_Adapter >::value, void * >::type=0)
Copy constructor from a linear expression adapter that takes a Representation.
- `template<typename LE_Adapter >`
`Linear_Expression` (const LE_Adapter &e, `dimension_type` space_dim, typename Enable_If< Is_Same_Or_Derived< Expression_Adapter_Base, LE_Adapter >::value, void * >::type=0)
Copy constructor from a linear expression adapter that takes a space dimension.
- `template<typename LE_Adapter >`
`Linear_Expression` (const LE_Adapter &e, `dimension_type` space_dim, `Representation` r, typename Enable_If< Is_Same_Or_Derived< Expression_Adapter_Base, LE_Adapter >::value, void * >::type=0)
Copy constructor from a linear expression adapter that takes a space dimension and a Representation.
- `Linear_Expression & operator=` (const `Linear_Expression` &e)
Assignment operator.
- `~Linear_Expression` ()
Destructor.
- `Linear_Expression` (Coefficient_traits::const_reference n, `Representation` r=default_representation)
Builds the linear expression corresponding to the inhomogeneous term n.
- `Linear_Expression` (`Variable` v, `Representation` r=default_representation)
Builds the linear expression corresponding to the variable v.
- `Representation representation` () const
*Returns the current representation of *this.*
- void `set_representation` (`Representation` r)
*Converts *this to the specified representation.*
- `const_iterator begin` () const
- `const_iterator end` () const
- `const_iterator lower_bound` (`Variable` v) const
- `dimension_type space_dimension` () const
*Returns the dimension of the vector space enclosing *this.*
- void `set_space_dimension` (`dimension_type` n)
*Sets the dimension of the vector space enclosing *this to n.*
- `Coefficient_traits::const_reference coefficient` (`Variable` v) const
*Returns the coefficient of v in *this.*
- void `set_coefficient` (`Variable` v, `Coefficient_traits::const_reference` n)
*Sets the coefficient of v in *this to n.*
- `Coefficient_traits::const_reference inhomogeneous_term` () const
*Returns the inhomogeneous term of *this.*
- void `set_inhomogeneous_term` (`Coefficient_traits::const_reference` n)
*Sets the inhomogeneous term of *this to n.*
- void `linear_combine` (const `Linear_Expression` &y, `Variable` v)
- void `linear_combine` (const `Linear_Expression` &y, `Coefficient_traits::const_reference` c1, `Coefficient_traits::const_reference` c2)
- void `linear_combine_lax` (const `Linear_Expression` &y, `Coefficient_traits::const_reference` c1, `Coefficient_traits::const_reference` c2)
- void `swap_space_dimensions` (`Variable` v1, `Variable` v2)
Swaps the coefficients of the variables v1 and v2.
- void `remove_space_dimensions` (const `Variables_Set` &vars)
Removes all the specified dimensions from the expression.

- void `shift_space_dimensions` (`Variable` v, `dimension_type` n)
- void `permute_space_dimensions` (const std::vector< `Variable` > &cycle)
Permutes the space dimensions of the expression.
- 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.*
- 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.*
- void `m_swap` (`Linear_Expression` &y)
*Swaps *this with y.*
- `Linear_Expression` (const `Linear_Expression` &e, `dimension_type` space_dim)
Copy constructor with a specified space dimension.
- `Linear_Expression` (const `Linear_Expression` &e, `dimension_type` space_dim, `Representation` r)
Copy constructor with a specified space dimension and representation.
- bool `is_equal_to` (const `Linear_Expression` &x) const
- void `normalize` ()
- void `sign_normalize` ()
- bool `all_zeroes` (const `Variables_Set` &vars) const
Returns true if the coefficient of each variable in vars[i] is 0.

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.

Related Functions

(Note that these are not member functions.)

- [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+](#) ([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+](#) (const [Linear_Expression](#) &e)
Returns the linear expression e .
- [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-](#) ([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*](#) ([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 & operator/=(Linear_Expression &e, Coefficient_traits::const_reference n)`
Returns the linear expression n / e and assigns it to e .
- `void neg_assign (Linear_Expression &e)`
Assigns to e its own negation.
- `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 .*
- `void add_mul_assign (Linear_Expression &e1, Coefficient_traits::const_reference factor, const Linear_Expression &e2)`
Sums $e2$ multiplied by $factor$ into $e1$.
- `void sub_mul_assign (Linear_Expression &e1, Coefficient_traits::const_reference factor, const Linear_Expression &e2)`
Subtracts $e2$ multiplied by $factor$ from $e1$.
- `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 .*
- `std::ostream & operator<< (std::ostream &s, const Linear_Expression &e)`
Output operator.
- `void swap (Linear_Expression &x, Linear_Expression &y)`
Swaps x with y .
- `Linear_Expression operator+ (const Linear_Expression &e)`
- `Linear_Expression operator+ (const Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression operator+ (const Linear_Expression &e, const Variable v)`
- `Linear_Expression operator- (const Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression operator- (const Variable v, const Variable w)`
- `Linear_Expression operator* (const Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression & operator+=(Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression & operator-=(Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression operator+ (const Linear_Expression &e1, const Linear_Expression &e2)`
- `Linear_Expression operator+ (const Variable v, const Linear_Expression &e)`
- `Linear_Expression operator+ (Coefficient_traits::const_reference n, const Linear_Expression &e)`
- `Linear_Expression operator+ (const Variable v, const Variable w)`
- `Linear_Expression operator- (const Linear_Expression &e)`
- `Linear_Expression operator- (const Linear_Expression &e1, const Linear_Expression &e2)`
- `Linear_Expression operator- (const Variable v, const Linear_Expression &e)`
- `Linear_Expression operator- (const Linear_Expression &e, const Variable v)`
- `Linear_Expression operator- (Coefficient_traits::const_reference n, const Linear_Expression &e)`
- `Linear_Expression operator* (Coefficient_traits::const_reference n, const Linear_Expression &e)`
- `Linear_Expression & operator+=(Linear_Expression &e1, const Linear_Expression &e2)`
- `Linear_Expression & operator+=(Linear_Expression &e, const Variable v)`
- `Linear_Expression & operator-=(Linear_Expression &e1, const Linear_Expression &e2)`
- `Linear_Expression & operator-=(Linear_Expression &e, const Variable v)`
- `Linear_Expression & operator*=(Linear_Expression &e, Coefficient_traits::const_reference n)`
- `Linear_Expression & operator/=(Linear_Expression &e, Coefficient_traits::const_reference n)`
- `void neg_assign (Linear_Expression &e)`
- `Linear_Expression & add_mul_assign (Linear_Expression &e, Coefficient_traits::const_reference n, const Variable v)`

- `Linear_Expression & sub_mul_assign (Linear_Expression &e, Coefficient_traits::const_reference n, const Variable v)`
- `std::ostream & operator<< (std::ostream &s, const Linear_Expression &e)`
- `void swap (Linear_Expression &x, Linear_Expression &y)`

10.63.1 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.

10.63.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const Linear_Expression & e)

Ordinary copy constructor.

Note

The new expression will have the same representation as e (not necessarily the default representation).

template<typename LE_Adapter > Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (const LE_Adapter & e, typename Enable_If< Is_Same_Or_Derived< Expression_Adapter_Base, LE_Adapter >::value, void * >::type = 0) [inline], [explicit] Copy constructor from a linear expression adapter.

Note

The new expression will have the same representation as e (not necessarily the default representation).

template<typename LE_Adapter > Parma_Polyhedra_Library::Linear_Expression::Linear_Expression
(const LE_Adapter & e, dimension_type space_dim, typename Enable_If< Is_Same_Or_Derived<
Expression_Adapter_Base, LE_Adapter >::value, void * >::type = 0) [inline], [explicit]
Copy constructor from a linear expression adapter that takes a space dimension.

Note

The new expression will have the same representation as `e` (not necessarily `default_representation`).

Parma_Polyhedra_Library::Linear_Expression::Linear_Expression (Variable v, Representation r =
default_representation) Builds the linear expression corresponding to the variable `v`.

Exceptions

<code>std::length_error</code>	Thrown if the space dimension of <code>v</code> exceeds Linear_Expression::max_space_dimension() .
--------------------------------	--

10.63.3 Member Function Documentation

Linear_Expression::const_iterator Parma_Polyhedra_Library::Linear_Expression::begin () const
[inline] Returns an iterator that points to the first nonzero coefficient in the expression.

Linear_Expression::const_iterator Parma_Polyhedra_Library::Linear_Expression::end () const
[inline] Returns an iterator that points to the last nonzero coefficient in the expression.

Linear_Expression::const_iterator Parma_Polyhedra_Library::Linear_Expression::lower_bound (Vari-
able v) const [inline] Returns an iterator that points to the first nonzero coefficient of a variable
bigger than or equal to `v`.

void Parma_Polyhedra_Library::Linear_Expression::linear_combine (const Linear_Expression & y,
Variable v) Linearly combines `*this` with `y` so that the coefficient of `v` is 0.

Parameters

<code>y</code>	The expression that will be combined with <code>*this</code> object;
<code>v</code>	The variable whose coefficient has to become 0.

Computes a linear combination of `*this` and `y` having the coefficient of variable `v` equal to 0. Then it assigns the resulting expression to `*this`.

`*this` and `y` must have the same space dimension.

void Parma_Polyhedra_Library::Linear_Expression::linear_combine (const Linear_Expression & y,
Coefficient_traits::const_reference c1, Coefficient_traits::const_reference c2) [inline] Equiv-
alent to `*this = *this * c1 + y * c2`, but assumes that `c1` and `c2` are not 0.

void Parma_Polyhedra_Library::Linear_Expression::linear_combine_lax (const Linear_Expression
& y, Coefficient_traits::const_reference c1, Coefficient_traits::const_reference c2) [inline]
Equivalent to `*this = *this * c1 + y * c2`. `c1` and `c2` may be 0.

void Parma_Polyhedra_Library::Linear_Expression::remove_space_dimensions (const Variables_-
Set & vars) [inline] Removes all the specified dimensions from the expression.

The space dimension of the variable with the highest space dimension in `vars` must be at most the space dimension of `this`.

void Parma_Polyhedra_Library::Linear_Expression::shift_space_dimensions (Variable *v*, dimension-type *n*) [inline] Shift by *n* positions the coefficients of variables, starting from the coefficient of *v*. This increases the space dimension by *n*.

void Parma_Polyhedra_Library::Linear_Expression::permute_space_dimensions (const std::vector< Variable > & *cycle*) [inline] Permutes the space dimensions of the expression.

Parameters

<i>cycle</i>	A vector representing a cycle of the permutation according to which the space dimensions must be rearranged.
--------------	--

The *cycle* vector represents a cycle of a permutation of space dimensions. For example, the permutation $\{x_1 \mapsto x_2, x_2 \mapsto x_3, x_3 \mapsto x_1\}$ can be represented by the vector containing x_1, x_2, x_3 .

bool Parma_Polyhedra_Library::Linear_Expression::is_equal_to (const Linear_Expression & *x*) const [inline] Returns `true` if **this* is equal to *x*. Note that *(*this == x)* has a completely different meaning.

void Parma_Polyhedra_Library::Linear_Expression::normalize () [inline] Normalizes the modulo of the coefficients and of the inhomogeneous term so that they are mutually prime.

Computes the Greatest Common Divisor (GCD) among the coefficients and the inhomogeneous term and normalizes them by the GCD itself.

void Parma_Polyhedra_Library::Linear_Expression::sign_normalize () [inline] Ensures that the first nonzero homogeneous coefficient is positive, by negating the row if necessary.

10.63.4 Friends And Related Function Documentation

Linear_Expression operator+ (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [related] Returns the linear expression $e1 + e2$.

Linear_Expression operator+ (Variable *v*, Variable *w*) [related] Returns the linear expression $v + w$.

Linear_Expression operator+ (Variable *v*, const Linear_Expression & *e*) [related] Returns the linear expression $v + e$.

Linear_Expression operator+ (const Linear_Expression & *e*, Variable *v*) [related] Returns the linear expression $e + v$.

Linear_Expression operator+ (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*) [related] Returns the linear expression $n + e$.

Linear_Expression operator+ (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related] Returns the linear expression $e + n$.

Linear_Expression operator+ (const Linear_Expression & *e*) [related] Returns the linear expression *e*.

Linear_Expression operator- (const Linear_Expression & *e*) [related] Returns the linear expression $- e$.

Linear_Expression operator- (**const Linear_Expression & *e1***, **const Linear_Expression & *e2***)
[related] Returns the linear expression $e1 - e2$.

Linear_Expression operator- (**Variable *v***, **Variable *w***) **[related]** Returns the linear expression $v - w$.

Linear_Expression operator- (**Variable *v***, **const Linear_Expression & *e***) **[related]** Returns the linear expression $v - e$.

Linear_Expression operator- (**const Linear_Expression & *e***, **Variable *v***) **[related]** Returns the linear expression $e - v$.

Linear_Expression operator- (**Coefficient_traits::const_reference *n***, **const Linear_Expression & *e***)
[related] Returns the linear expression $n - e$.

Linear_Expression operator- (**const Linear_Expression & *e***, **Coefficient_traits::const_reference *n***)
[related] Returns the linear expression $e - n$.

Linear_Expression operator* (**Coefficient_traits::const_reference *n***, **const Linear_Expression & *e***)
[related] Returns the linear expression $n * e$.

Linear_Expression operator* (**const Linear_Expression & *e***, **Coefficient_traits::const_reference *n***)
[related] Returns the linear expression $e * n$.

Linear_Expression & operator+= (**Linear_Expression & *e1***, **const Linear_Expression & *e2***)
[related] Returns the linear expression $e1 + e2$ and assigns it to $e1$.

Linear_Expression & operator+= (**Linear_Expression & *e***, **Variable *v***) **[related]** Returns the linear expression $e + v$ and assigns it to e .

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of v exceeds Linear_Expression::max_space_dimension() .
--------------------------	---

Linear_Expression & operator+= (**Linear_Expression & *e***, **Coefficient_traits::const_reference *n***)
[related] Returns the linear expression $e + n$ and assigns it to e .

Linear_Expression & operator-= (**Linear_Expression & *e1***, **const Linear_Expression & *e2***) **[related]**
Returns the linear expression $e1 - e2$ and assigns it to $e1$.

Linear_Expression & operator-= (**Linear_Expression & *e***, **Variable *v***) **[related]** Returns the linear expression $e - v$ and assigns it to e .

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of v exceeds Linear_Expression::max_space_dimension() .
--------------------------	---

Linear_Expression & operator-= (**Linear_Expression & *e***, **Coefficient_traits::const_reference *n***)
[related] Returns the linear expression $e - n$ and assigns it to e .

Linear_Expression & operator*= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*)
[related] Returns the linear expression $n * e$ and assigns it to *e*.

Linear_Expression & operator/= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*)
[related] Returns the linear expression n / e and assigns it to *e*.

void neg_assign (Linear_Expression & *e*) [related] Assigns to *e* its own negation.

Linear_Expression & add_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, Variable *v*) [related] Returns the linear expression $e + n * v$ and assigns it to *e*.

void add_mul_assign (Linear_Expression & *e1*, Coefficient_traits::const_reference *factor*, const Linear_Expression & *e2*) [related] Sums *e2* multiplied by *factor* into *e1*.

void sub_mul_assign (Linear_Expression & *e1*, Coefficient_traits::const_reference *factor*, const Linear_Expression & *e2*) [related] Subtracts *e2* multiplied by *factor* from *e1*.

Linear_Expression & sub_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, Variable *v*) [related] Returns the linear expression $e - n * v$ and assigns it to *e*.

std::ostream & operator<< (std::ostream & *s*, const Linear_Expression & *e*) [related] Output operator.

void swap (Linear_Expression & *x*, Linear_Expression & *y*) [related] Swaps *x* with *y*.

Linear_Expression operator+ (const Linear_Expression & *e*) [related]

Linear_Expression operator+ (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related]

Linear_Expression operator+ (const Linear_Expression & *e*, const Variable *v*) [related]

Linear_Expression operator- (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related]

Linear_Expression operator- (const Variable *v*, const Variable *w*) [related]

Linear_Expression operator* (const Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related]

Linear_Expression & operator+= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related]

Linear_Expression & operator-= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*) [related]

Linear_Expression operator+ (const Linear_Expression & *e1*, const Linear_Expression & *e2*) [related]

Linear_Expression operator+ (const Variable *v*, const Linear_Expression & *e*) [related]

Linear_Expression operator+ (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*)
[related]

Linear_Expression operator+ (const Variable *v*, const Variable *w*) **[related]**

Linear_Expression operator- (const Linear_Expression & *e*) **[related]**

Linear_Expression operator- (const Linear_Expression & *e1*, const Linear_Expression & *e2*)
[related]

Linear_Expression operator- (const Variable *v*, const Linear_Expression & *e*) **[related]**

Linear_Expression operator- (const Linear_Expression & *e*, const Variable *v*) **[related]**

Linear_Expression operator- (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*)
[related]

Linear_Expression operator* (Coefficient_traits::const_reference *n*, const Linear_Expression & *e*)
[related]

Linear_Expression & operator+= (Linear_Expression & *e1*, const Linear_Expression & *e2*)
[related]

Linear_Expression & operator+= (Linear_Expression & *e*, const Variable *v*) **[related]**

Linear_Expression & operator-= (Linear_Expression & *e1*, const Linear_Expression & *e2*) **[related]**

Linear_Expression & operator-= (Linear_Expression & *e*, const Variable *v*) **[related]**

Linear_Expression & operator*= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*)
[related]

Linear_Expression & operator/= (Linear_Expression & *e*, Coefficient_traits::const_reference *n*)
[related]

void neg_assign (Linear_Expression & *e*) **[related]**

Linear_Expression & add_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, const Variable *v*) **[related]**

Linear_Expression & sub_mul_assign (Linear_Expression & *e*, Coefficient_traits::const_reference *n*, const Variable *v*) **[related]**

std::ostream & operator<< (std::ostream & *s*, const Linear_Expression & *e*) **[related]**

void swap (Linear_Expression & *x*, Linear_Expression & *y*) **[related]** The documentation for this class was generated from the following file:

- ppl.hh

10.64 Parma Polyhedra Library::Linear_Form< C > Class Template Reference

A linear form with interval coefficients.

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

Public Member Functions

- [Linear_Form](#) ()
Default constructor: returns a copy of `Linear_Form::zero()`.
- [Linear_Form](#) (const [Linear_Form](#) &f)
Ordinary copy constructor.
- [~Linear_Form](#) ()
Destructor.
- [Linear_Form](#) (const C &n)
Builds the linear form corresponding to the inhomogeneous term `n`.
- [Linear_Form](#) ([Variable](#) v)
Builds the linear form corresponding to the variable `v`.
- [Linear_Form](#) (const [Linear_Expression](#) &e)
Builds a linear form approximating the linear expression `e`.
- [dimension_type](#) [space_dimension](#) () const
*Returns the dimension of the vector space enclosing `*this`.*
- const C & [coefficient](#) ([Variable](#) v) const
*Returns the coefficient of `v` in `*this`.*
- const C & [inhomogeneous_term](#) () const
*Returns the inhomogeneous term of `*this`.*
- void [negate](#) ()
*Negates all the coefficients 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`.*
- 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 [m_swap](#) ([Linear_Form](#) &y)
*Swaps `*this` with `y`.*
- bool [overflows](#) () const
Verifies if the linear form overflows.
- void [relative_error](#) ([Floating_Point_Format](#) analyzed_format, [Linear_Form](#) &result) const
*Computes the relative error associated to floating point computations that operate on a quantity that is overapproximated by `*this`.*

- template<typename Target >
bool [intervalize](#) (const [FP_Oracle](#)< Target, C > &oracle, C &result) const
*Makes result become an interval that overapproximates all the possible values of *this.*

Static Public Member Functions

- static [dimension_type](#) [max_space_dimension](#) ()
Returns the maximum space dimension a [Linear_Form](#) can handle.

Related Functions

(Note that these are not member functions.)

- template<typename FP.Interval_Type >
void [discard_occurrences](#) (std::map< [dimension_type](#), [Linear_Form](#)< FP.Interval_Type > > &lf, [Variable](#) var)
- template<typename FP.Interval_Type >
void [affine_form_image](#) (std::map< [dimension_type](#), [Linear_Form](#)< FP.Interval_Type > > &lf, [Variable](#) var, const [Linear_Form](#)< FP.Interval_Type > &f)
- template<typename FP.Interval_Type >
void [upper_bound_assign](#) (std::map< [dimension_type](#), [Linear_Form](#)< FP.Interval_Type > > &ls1, const std::map< [dimension_type](#), [Linear_Form](#)< FP.Interval_Type > > &ls2)
- template<typename C >
void [swap](#) ([Linear_Form](#)< C > &x, [Linear_Form](#)< C > &y)
Swaps x with y.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) (const [Linear_Form](#)< C > &f1, const [Linear_Form](#)< C > &f2)
Returns the linear form $f1 + f2$.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) ([Variable](#) v, const [Linear_Form](#)< C > &f)
Returns the linear form $v + f$.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) (const [Linear_Form](#)< C > &f, [Variable](#) v)
Returns the linear form $f + v$.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) (const C &n, const [Linear_Form](#)< C > &f)
Returns the linear form $n + f$.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) (const [Linear_Form](#)< C > &f, const C &n)
Returns the linear form $f + n$.
- template<typename C >
[Linear_Form](#)< C > [operator+](#) (const [Linear_Form](#)< C > &f)
Returns the linear form f .
- template<typename C >
[Linear_Form](#)< C > [operator-](#) (const [Linear_Form](#)< C > &f)
Returns the linear form $- f$.
- template<typename C >
[Linear_Form](#)< C > [operator-](#) (const [Linear_Form](#)< C > &f1, const [Linear_Form](#)< C > &f2)
Returns the linear form $f1 - f2$.
- template<typename C >
[Linear_Form](#)< C > [operator-](#) ([Variable](#) v, const [Linear_Form](#)< C > &f)

- Returns the linear form $v - f$.*

• `template<typename C >`
`Linear_Form< C > operator-` (const `Linear_Form< C >` &f, `Variable v`)

Returns the linear form $f - v$.
- `template<typename C >`
`Linear_Form< C > operator-` (const `C &n`, const `Linear_Form< C >` &f)

Returns the linear form $n - f$.
- `template<typename C >`
`Linear_Form< C > operator-` (const `Linear_Form< C >` &f, const `C &n`)

Returns the linear form $f - n$.
- `template<typename C >`
`Linear_Form< C > operator*` (const `C &n`, const `Linear_Form< C >` &f)

*Returns the linear form $n * f$.*
- `template<typename C >`
`Linear_Form< C > operator*` (const `Linear_Form< C >` &f, const `C &n`)

*Returns the linear form $f * n$.*
- `template<typename C >`
`Linear_Form< C > & operator+=` (`Linear_Form< C >` &f1, const `Linear_Form< C >` &f2)

Returns the linear form $f1 + f2$ and assigns it to $e1$.
- `template<typename C >`
`Linear_Form< C > & operator+=` (`Linear_Form< C >` &f, `Variable v`)

Returns the linear form $f + v$ and assigns it to f .
- `template<typename C >`
`Linear_Form< C > & operator+=` (`Linear_Form< C >` &f, const `C &n`)

Returns the linear form $f + n$ and assigns it to f .
- `template<typename C >`
`Linear_Form< C > & operator-=` (`Linear_Form< C >` &f1, const `Linear_Form< C >` &f2)

Returns the linear form $f1 - f2$ and assigns it to $f1$.
- `template<typename C >`
`Linear_Form< C > & operator-=` (`Linear_Form< C >` &f, `Variable v`)

Returns the linear form $f - v$ and assigns it to f .
- `template<typename C >`
`Linear_Form< C > & operator-=` (`Linear_Form< C >` &f, const `C &n`)

Returns the linear form $f - n$ and assigns it to f .
- `template<typename C >`
`Linear_Form< C > & operator*=` (`Linear_Form< C >` &f, const `C &n`)

*Returns the linear form $n * f$ and assigns it to f .*
- `template<typename C >`
`Linear_Form< C > & operator/=` (`Linear_Form< C >` &f, const `C &n`)

Returns the linear form f / n and assigns it to f .
- `template<typename C >`
`bool operator==` (const `Linear_Form< C >` &x, const `Linear_Form< C >` &y)

Returns `true` if and only if x and y are equal.
- `template<typename C >`
`bool operator!=` (const `Linear_Form< C >` &x, const `Linear_Form< C >` &y)

Returns `true` if and only if x and y are different.
- `template<typename C >`
`std::ostream & operator<<` (`std::ostream &s`, const `Linear_Form< C >` &f)

Output operator.

- [illegible]

- `template<typename C >`
`std::ostream & operator<< (std::ostream &s, const Linear_Form< C > &f)`

10.64.1 Detailed Description

`template<typename C>class Parma_Polyhedra_Library::Linear_Form< C >`

A linear form with interval coefficients.

An object of the class [Linear_Form](#) represents the interval linear form

$$\sum_{i=0}^{n-1} a_i x_i + b$$

where n is the dimension of the vector space, each a_i is the coefficient of the i -th variable x_i and b is the inhomogeneous term. The coefficients and the inhomogeneous term of the linear form have the template parameter C as their type. C must be the type of an [Interval](#).

How to build a linear form.

A full set of functions is defined in order to provide a convenient interface for building complex linear forms starting from simpler ones and from objects of the classes [Variable](#) and C . Available operators include binary addition and subtraction, as well as multiplication and division by a coefficient. The space dimension of a linear form is defined as the highest variable dimension among variables that have a nonzero coefficient in the linear form, or zero if no such variable exists. The space dimension for each variable x_i is given by $i + 1$.

Example

Given the type T of an [Interval](#) with floating point coefficients (though any integral type may also be used), the following code builds the interval linear form $lf = x_5 - x_2 + 1$ with space dimension 6:

```
Variable x5(5);
Variable x2(2);
T x5_coefficient;
x5_coefficient.lower() = 2.0;
x5_coefficient.upper() = 3.0;
T inhomogeneous_term;
inhomogeneous_term.lower() = 4.0;
inhomogeneous_term.upper() = 8.0;
Linear_Form<T> lf(x2);
lf = -lf;
lf += Linear_Form<T>(x2);
Linear_Form<T> lf_x5(x5);
lf_x5 *= x5_coefficient;
lf += lf_x5;
```

Note that `lf_x5` is created with space dimension 6, while `lf` is created with space dimension 0 and then extended first to space dimension 2 when `x2` is subtracted and finally to space dimension 6 when `lf_x5` is added.

10.64.2 Constructor & Destructor Documentation

`template<typename C > Parma_Polyhedra_Library::Linear_Form< C >::Linear_Form (Variable v)` Builds the linear form corresponding to the variable v .

Exceptions

<code>std::length_error</code>	Thrown if the space dimension of v exceeds <code>Linear_Form::max_space_dimension()</code> .
--------------------------------	--

10.64.3 Member Function Documentation

template<typename C > bool Parma_Polyhedra_Library::Linear_Form< C >::overflows () const **[inline]** Verifies if the linear form overflows.

Returns

Returns `false` if all coefficients in `lf` are bounded, `true` otherwise.

`T` must be the type of possibly unbounded quantities.

template<typename C > void Parma_Polyhedra_Library::Linear_Form< C >::relative_error (Floating-Point_Format *analyzed_format*, Linear_Form< C > & *result*) const Computes the relative error associated to floating point computations that operate on a quantity that is overapproximated by `*this`.

Parameters

<i>analyzed_format</i>	The floating point format used by the analyzed program.
<i>result</i>	Becomes the linear form corresponding to the relative error committed.

This method makes `result` become a linear form obtained by evaluating the function $\varepsilon_f(l)$ on the linear form. This function is defined as:

$$\varepsilon_f \left([a, b] + \sum_{v \in \mathcal{V}} [a_v, b_v] v \right) \stackrel{\text{def}}{=} (\max(|a|, |b|) \otimes^\# [-\beta^{-p}, \beta^{-p}]) + \sum_{v \in \mathcal{V}} (\max(|a_v|, |b_v|) \otimes^\# [-\beta^{-p}, \beta^{-p}]) v$$

where `p` is the fraction size in bits for the format `f` and β the base.

The result is undefined if `T` is not the type of an interval with floating point boundaries.

template<typename C > template<typename Target > bool Parma_Polyhedra_Library::Linear_Form< C >::intervalize (const FP_Oracle< Target, C > & *oracle*, C & *result*) const Makes `result` become an interval that overapproximates all the possible values of `*this`.

Parameters

<i>oracle</i>	The FP_Oracle to be queried.
<i>result</i>	The linear form that will store the result.

Returns

`true` if the operation was successful, `false` otherwise (the possibility of failure depends on the oracle's implementation).

Template type parameters

- The class template parameter `Target` specifies the implementation of [Concrete_Expression](#) to be used.

This method makes `result` become $\iota(lf)\rho^\#$, that is an interval defined as:

$$\iota \left(i + \sum_{v \in \mathcal{V}} i_v v \right) \rho^\# \stackrel{\text{def}}{=} i \oplus^\# \left(\bigoplus_{v \in \mathcal{V}}^\# i_v \otimes^\# \rho^\#(v) \right)$$

where $\rho^\#(v)$ is an interval (provided by the oracle) that correctly approximates the value of `v`.

The result is undefined if `C` is not the type of an interval with floating point boundaries.

10.64.4 Friends And Related Function Documentation

template<typename FP_Interval_Type > void discard_occurrences (std::map< dimension_type, Linear_Form< FP_Interval_Type > > & *lf_store*, Variable *var*) [related] Discards all linear forms containing variable `var` from the linear form abstract store `lf_store`.

template<typename FP_Interval_Type > void affine_form_image (std::map< dimension_type, Linear_Form< FP_Interval_Type > > & lf_store, Variable var, const Linear_Form< FP_Interval_Type > & lf) [related] Assigns the linear form `lf` to `var` in the linear form abstract store `lf_store`, then discards all occurrences of `var` from it.

template<typename FP_Interval_Type > void upper_bound_assign (std::map< dimension_type, Linear_Form< FP_Interval_Type > > & ls1, const std::map< dimension_type, Linear_Form< FP_Interval_Type > > & ls2) [related] Discards from `ls1` all linear forms but those that are associated to the same variable in `ls2`.

template<typename C > void swap (Linear_Form< C > & x, Linear_Form< C > & y) [related] Swaps `x` with `y`.

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > & f1, const Linear_Form< C > & f2) [related] Returns the linear form $f1 + f2$.

template<typename C > Linear_Form< C > operator+ (Variable v, const Linear_Form< C > & f) [related] Returns the linear form $v + f$.

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > & f, Variable v) [related] Returns the linear form $f + v$.

template<typename C > Linear_Form< C > operator+ (const C & n, const Linear_Form< C > & f) [related] Returns the linear form $n + f$.

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > & f, const C & n) [related] Returns the linear form $f + n$.

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > & f) [related] Returns the linear form f .

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > & f) [related] Returns the linear form $-f$.

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > & f1, const Linear_Form< C > & f2) [related] Returns the linear form $f1 - f2$.

template<typename C > Linear_Form< C > operator- (Variable v, const Linear_Form< C > & f) [related] Returns the linear form $v - f$.

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > & f, Variable v) [related] Returns the linear form $f - v$.

template<typename C > Linear_Form< C > operator- (const C & n, const Linear_Form< C > & f) [related] Returns the linear form $n - f$.

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > & f, const C & n) [related] Returns the linear form $f - n$.

template<typename C > Linear_Form< C > operator* (const C & n, const Linear_Form< C > & f) [related] Returns the linear form $n * f$.

template<typename C > Linear_Form< C > operator* (const Linear_Form< C > &f, const C &n) [related] Returns the linear form $f * n$.

template<typename C > Linear_Form< C > & operator+= (Linear_Form< C > &f1, const Linear_Form< C > &f2) [related] Returns the linear form $f1 + f2$ and assigns it to $e1$.

template<typename C > Linear_Form< C > & operator+= (Linear_Form< C > &f, Variable v) [related] Returns the linear form $f + v$ and assigns it to f .

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of v exceeds <code>Linear_Form::max_space_dimension()</code> .
--------------------------	--

template<typename C > Linear_Form< C > & operator+= (Linear_Form< C > &f, const C &n) [related] Returns the linear form $f + n$ and assigns it to f .

template<typename C > Linear_Form< C > & operator-= (Linear_Form< C > &f1, const Linear_Form< C > &f2) [related] Returns the linear form $f1 - f2$ and assigns it to $f1$.

template<typename C > Linear_Form< C > & operator-= (Linear_Form< C > &f, Variable v) [related] Returns the linear form $f - v$ and assigns it to f .

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of v exceeds <code>Linear_Form::max_space_dimension()</code> .
--------------------------	--

template<typename C > Linear_Form< C > & operator-= (Linear_Form< C > &f, const C &n) [related] Returns the linear form $f - n$ and assigns it to f .

template<typename C > Linear_Form< C > & operator*= (Linear_Form< C > &f, const C &n) [related] Returns the linear form $n * f$ and assigns it to f .

template<typename C > Linear_Form< C > & operator/= (Linear_Form< C > &f, const C &n) [related] Returns the linear form f / n and assigns it to f .

Performs the division of a linear form by a scalar. It is up to the user to ensure that division by 0 is not performed.

template<typename C > bool operator== (const Linear_Form< C > &x, const Linear_Form< C > &y) [related] Returns `true` if and only if x and y are equal.

template<typename C > bool operator!= (const Linear_Form< C > &x, const Linear_Form< C > &y) [related] Returns `true` if and only if x and y are different.

template<typename C > std::ostream & operator<< (std::ostream &s, const Linear_Form< C > &f) [related] Output operator.

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > &f, const C &n) [related]

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > &f, const Variable v) [related]

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > &f, const C &n) [related]

template<typename C > Linear_Form< C > operator- (const Variable v, const Variable w) [related]

template<typename C > Linear_Form< C > operator* (const Linear_Form< C > &f, const C &n) [related]

template<typename C > Linear_Form< C > & operator+= (Linear_Form< C > &f, const C &n) [related]

template<typename C > Linear_Form< C > & operator-= (Linear_Form< C > &f, const C &n) [related]

template<typename C > bool operator!= (const Linear_Form< C > &x, const Linear_Form< C > &y) [related]

template<typename C > void swap (Linear_Form< C > &x, Linear_Form< C > &y) [related]

template<typename C > Linear_Form< C > operator+ (const Linear_Form< C > &f1, const Linear_Form< C > &f2) [related]

template<typename C > Linear_Form< C > operator+ (const Variable v, const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator+ (const C &n, const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > &f1, const Linear_Form< C > &f2) [related]

template<typename C > Linear_Form< C > operator- (const Variable v, const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator- (const Linear_Form< C > &f, const Variable v) [related]

template<typename C > Linear_Form< C > operator- (const C &n, const Linear_Form< C > &f) [related]

template<typename C > Linear_Form< C > operator* (const C &n, const Linear_Form< C > &f) [related]

```
template<typename C > Linear_Form< C > & operator+=( Linear_Form< C > & f1, const
Linear_Form< C > & f2 ) [related]
```

```
template<typename C > Linear_Form< C > & operator+=( Linear_Form< C > & f, const Vari-
able v ) [related]
```

```
template<typename C > Linear_Form< C > & operator-= ( Linear_Form< C > & f1, const Linear-
_Form< C > & f2 ) [related]
```

```
template<typename C > Linear_Form< C > & operator-= ( Linear_Form< C > & f, const Variable
v ) [related]
```

```
template<typename C > Linear_Form< C > & operator*= ( Linear_Form< C > & f, const C & n
) [related]
```

```
template<typename C > Linear_Form< C > & operator/= ( Linear_Form< C > & f, const C & n
) [related]
```

```
template<typename C > bool operator==( const Linear_Form< C > & x, const Linear_Form< C
> & y ) [related]
```

```
template<typename C > std::ostream & operator<< ( std::ostream & s, const Linear_Form< C >
& f ) [related]
```

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

- `ppl.hh`

10.65 Parma Polyhedra Library::MIP_Problem Class Reference

A Mixed Integer (linear) Programming problem.

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

Classes

- class `const_iterator`

A read-only iterator on the constraints defining the feasible region.

Public Types

- enum `Control_Parameter_Name` { `PRICING` }
- enum `Control_Parameter_Value` { `PRICING_STEEPEST_EDGE.FLOAT`, `PRICING_STEEPEST_-EDGE.EXACT`, `PRICING_TEXTBOOK` }

Possible values for MIP problem's control parameters.

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 `[first, 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 `[first, 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` &numer, `Coefficient` &denom) const
Sets `numer` and `denom` so that $\frac{\text{numer}}{\text{denom}}$ 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` &numer, `Coefficient` &denom) const
Sets `numer` and `denom` so that $\frac{\text{numer}}{\text{denom}}$ 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 `m_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` &mip)
Output operator.
- void `swap` (`MIP_Problem` &x, `MIP_Problem` &y)
Swaps `x` with `y`.
- void `swap` (`MIP_Problem` &x, `MIP_Problem` &y)

10.65.1 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.

10.65.2 Member Enumeration Documentation

enum Parma_Polyhedra_Library::MIP_Problem::Control_Parameter_Name Names of MIP problems' control parameters.

Enumerator

PRICING The pricing rule.

enum Parma_Polyhedra_Library::MIP_Problem::Control_Parameter_Value Possible values for M-IP 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.

10.65.3 Constructor & Destructor Documentation

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

<i>dim</i>	The dimension of the vector space enclosing <i>*this</i> (optional argument with default value 0).
------------	--

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds <code>max_space_dimension()</code> .
--------------------------	---

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) Builds an MIP problem having space dimension *dim* from the sequence of constraints in the range [*first*,*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

<i>dim</i>	The dimension of the vector space enclosing <i>*this</i> .
<i>first</i>	An input iterator to the start of the sequence of constraints.
<i>last</i>	A past-the-end input iterator to the sequence of constraints.
<i>int_vars</i>	The set of variables' indexes that are constrained to take integer values.
<i>obj</i>	The objective function (optional argument with default value 0).
<i>mode</i>	The optimization mode (optional argument with default value MAXIMIZATION).

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds <code>max_space_dimension()</code> .
<i>std::invalid_argument</i>	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 <i>dim</i> .

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) Builds an MIP problem having space dimension *dim* from the sequence of constraints in the range [*first*,*last*), the objective function *obj* and optimization mode *mode*.

Parameters

<i>dim</i>	The dimension of the vector space enclosing <i>*this</i> .
<i>first</i>	An input iterator to the start of the sequence of constraints.
<i>last</i>	A past-the-end input iterator to the sequence of constraints.
<i>obj</i>	The objective function (optional argument with default value 0).
<i>mode</i>	The optimization mode (optional argument with default value MAXIMIZATION).

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds <code>max_space_dimension()</code> .
<i>std::invalid_argument</i>	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 <i>dim</i> .

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

<i>dim</i>	The dimension of the vector space enclosing <i>*this</i> .
<i>cs</i>	The constraint system defining the feasible region.
<i>obj</i>	The objective function (optional argument with default value 0).
<i>mode</i>	The optimization mode (optional argument with default value MAXIMIZATION).

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds max_space_dimension() .
<i>std::invalid_argument</i>	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 <i>dim</i> .

10.65.4 Member Function Documentation

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.

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

<i>m</i>	The number of dimensions to add.
----------	----------------------------------

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> 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.

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

<i>std::invalid_argument</i>	Thrown if some index in <i>i_vars</i> does not correspond to a space dimension in <i>*this</i> .
------------------------------	--

void Parma_Polyhedra_Library::MIP_Problem::add_constraint (const Constraint & *c*) Adds a copy of constraint *c* to the MIP problem.

Exceptions

<i>std::invalid_argument</i>	Thrown if the constraint <i>c</i> is a strict inequality or if its space dimension is strictly greater than the space dimension of <i>*this</i> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if the constraint system <code>cs</code> contains any strict inequality or if its space dimension is strictly greater than the space dimension of <code>*this</code> .
------------------------------	---

void Parma_Polyhedra_Library::MIP_Problem::set_objective_function (const Linear_Expression & *obj*) Sets the objective function to `obj`.

Exceptions

<i>std::invalid_argument</i>	Thrown if the space dimension of <code>obj</code> is strictly greater than the space dimension of <code>*this</code> .
------------------------------	--

bool Parma_Polyhedra_Library::MIP_Problem::is_satisfiable () const Checks satisfiability of `*this`.

Returns

`true` if and only if the MIP problem is satisfiable.

MIP_Problem_Status Parma_Polyhedra_Library::MIP_Problem::solve () const Optimizes the M-IP problem.

Returns

An `MIP_Problem_Status` flag indicating the outcome of the optimization attempt (unfeasible, unbounded or optimized problem).

void Parma_Polyhedra_Library::MIP_Problem::evaluate_objective_function (const Generator & *evaluating_point*, Coefficient & *numer*, Coefficient & *denom*) const Sets `num` and `denom` so that $\frac{\text{numer}}{\text{denom}}$ is the result of evaluating the objective function on `evaluating_point`.

Parameters

<i>evaluating_point</i>	The point on which the objective function will be evaluated.
<i>numer</i>	On exit will contain the numerator of the evaluated value.
<i>denom</i>	On exit will contain the denominator of the evaluated value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>evaluating_point</code> are dimension-incompatible or if the generator <code>evaluating_point</code> is not a point.
------------------------------	---

const Generator& Parma_Polyhedra_Library::MIP_Problem::feasible_point () const Returns a feasible point for `*this`, if it exists.

Exceptions

<i>std::domain_error</i>	Thrown if the MIP problem is not satisfiable.
--------------------------	---

const Generator& Parma_Polyhedra_Library::MIP_Problem::optimizing_point () const Returns an optimal point for `*this`, if it exists.

Exceptions

<code>std::domain_error</code>	Thrown if <code>*this</code> does not have an optimizing point, i.e., if the MIP problem is unbounded or not satisfiable.
--------------------------------	---

void Parma_Polyhedra_Library::MIP_Problem::optimal_value (Coefficient & *numer*, Coefficient & *denom*) const [inline] Sets *numer* and *denom* so that $\frac{\text{numer}}{\text{denom}}$ is the solution of the optimization problem.

Exceptions

<code>std::domain_error</code>	Thrown if <code>*this</code> does not have an optimizing point, i.e., if the MIP problem is unbounded or not satisfiable.
--------------------------------	---

10.65.5 Friends And Related Function Documentation

std::ostream & operator<< (std::ostream & *s*, const MIP_Problem & *mip*) [related] Output operator.

void swap (MIP_Problem & *x*, MIP_Problem & *y*) [related] Swaps *x* with *y*.

void swap (MIP_Problem & *x*, MIP_Problem & *y*) [related] The documentation for this class was generated from the following file:

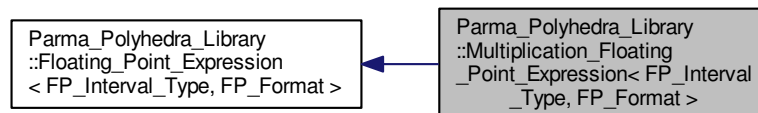
- `ppl.hh`

10.66 Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP-Interval_Type, FP_Format > Class Template Reference

A generic Multiplication Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
[::FP_Linear_Form](#) [FP_Linear_Form](#)

Alias for the Linear_Form<FP_Interval_Type> from [Floating_Point_Expression](#).

- typedef
[Floating_Point_Expression](#)
 < FP.Interval.Type, FP.Format >
 ::[FP.Interval.Abstract.Store](#) [FP.Interval.Abstract.Store](#)
Alias for the Box<FP.Interval.Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP.Interval.Type, FP.Format >
 ::[FP.Linear.Form.Abstract.Store](#) [FP.Linear.Form.Abstract.Store](#)
Alias for the std::map<dimension_type, FP.Linear.Form> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP.Interval.Type, FP.Format >
 ::[boundary_type](#) [boundary_type](#)
Alias for the FP.Interval.Type::boundary_type from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP.Interval.Type, FP.Format >
 ::[info_type](#) [info_type](#)
Alias for the FP.Interval.Type::info_type from [Floating_Point_Expression](#).

Public Member Functions

- bool [linearize](#) (const [FP.Interval.Abstract.Store](#) &int_store, const [FP.Linear.Form.Abstract.Store](#) &lf_store, [FP.Linear.Form](#) &result) const
Linearizes the expression in a given astract store.
- void [m_swap](#) ([MultiplicationFloating_Point_Expression](#)< FP.Interval.Type, FP.Format > &y)
*Swaps *this* with y.*

Constructors and Destructor

- [MultiplicationFloating_Point_Expression](#) ([Floating_Point_Expression](#)< FP.Interval.Type, FP.Format > *const x, [Floating_Point_Expression](#)< FP.Interval.Type, FP.Format > *const y)
Constructor with two parameters: builds the multiplication floating point expression corresponding to $x \otimes y$.
- [~MultiplicationFloating_Point_Expression](#) ()
Destructor.

Related Functions

(Note that these are not member functions.)

- template<typename FP.Interval.Type, typename FP.Format >
 void [swap](#) ([MultiplicationFloating_Point_Expression](#)< FP.Interval.Type, FP.Format > &x, [MultiplicationFloating_Point_Expression](#)< FP.Interval.Type, FP.Format > &y)
Swaps x with y.
- template<typename FP.Interval.Type, typename FP.Format >
 void [swap](#) ([MultiplicationFloating_Point_Expression](#)< FP.Interval.Type, FP.Format > &x, [MultiplicationFloating_Point_Expression](#)< FP.Interval.Type, FP.Format > &y)

Additional Inherited Members

10.66.1 Detailed Description

template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >

A generic Multiplication Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of multiplication floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms, $\boxplus^\#$ and $\boxtimes^\#$ two sound abstract operators on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \oplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \oplus^\# i'_v) v,$$

$$i \boxtimes^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \otimes^\# i') + \sum_{v \in \mathcal{V}} (i \otimes^\# i'_v) v.$$

Given an expression $[a, b] \otimes e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle [a, b] \otimes e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle [a, b] \otimes e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \left([a, b] \boxtimes^\# \langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \left([a, b] \boxtimes^\# \varepsilon_f \left(\langle e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \right) \boxplus^\# mf_f[-1, 1].$$

Given an expression $e_1 \otimes [a, b]$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \otimes [a, b] \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle e_1 \otimes [a, b] \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \langle [a, b] \otimes e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket.$$

Given an expression $e_1 \otimes e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\langle e_1 \otimes e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\langle e_1 \otimes e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket = \langle \iota \left(\langle e_1 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \rho^\# \otimes e_2 \rangle \llbracket \rho^\#, \rho_l^\# \rrbracket,$$

where $\varepsilon_f(l)$ is the linear form computed by calling method `Floating_Point_Expression::relative_error` on l , $\iota(l)\rho^\#$ is the linear form computed by calling method `Floating_Point_Expression::intervalize` on l and $\rho^\#$, and mf_f is a rounding error defined in `Floating_Point_Expression::absolute_error`.

Even though we intervalize the first operand in the above example, the actual implementation utilizes an heuristics for choosing which of the two operands must be intervalized in order to obtain the most precise result.

10.66.2 Member Function Documentation

template<typename FP_Interval_Type , typename FP_Format > bool Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const **[virtual]** Linearizes the expression in a given astract store.

Makes *result* become the linearization of **this* in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

true if the linearization succeeded, *false* otherwise.

Note that all variables occuring in the expressions represented by *first_operand* and *second_operand* MUST have an associated value in *int_store*. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how *result* is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.66.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) **[related]** Swaps *x* with *y*.

template<typename FP_Interval_Type , typename FP_Format > void swap (Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Multiplication_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) **[related]** The documentation for this class was generated from the following file:

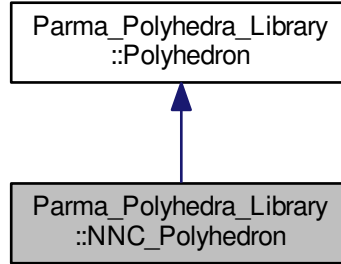
- *ppl.hh*

10.67 Parma_Polyhedra_Library::NNC_Polyhedron Class Reference

A not necessarily closed convex polyhedron.

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


Inheritance diagram for Parma_Polyhedra_Library::NNC_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)`.
- void `positive_time_elapse_assign` (const `Polyhedron` &y)
- Assigns to **this* (the best approximation of) the result of computing the *positive time-elapse* between **this* and *y*.

Additional Inherited Members

10.67.1 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.

10.67.2 Constructor & Destructor Documentation

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

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the NNC polyhedron;
<i>kind</i>	Specifies whether a universe or an empty NNC polyhedron should be built.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

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

<i>cs</i>	The system of constraints defining the polyhedron.
-----------	--

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

<i>cs</i>	The system of constraints defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

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

<i>gs</i>	The system of generators defining the polyhedron.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

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

<i>gs</i>	The system of generators defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

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

<i>cgs</i>	The system of congruences defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
------------	---

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

<i>cgs</i>	The system of congruences defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

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

<i>y</i>	The C polyhedron to be used;
<i>complexity</i>	This argument is ignored.

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

<i>box</i>	The box representing the polyhedron to be built;
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>box</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>grid</i>	The grid used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

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

<i>bd</i>	The BD shape used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

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

<i>os</i>	The octagonal shape used to build the polyhedron.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

10.67.3 Member Function Documentation

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::NNC_Polyhedron::positive_time_elapse_assign (const Polyhedron & y) [inline] Assigns to **this* (the best approximation of) the result of computing the [positive time-elapse](#) between **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

- ppl.hh

10.68 Parma_Polyhedra_Library::PIP_Solution_Node::No_Constraints Struct Reference

A tag type to select the alternative copy constructor.

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

10.68.1 Detailed Description

A tag type to select the alternative copy constructor.

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

- ppl.hh

10.69 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.

10.69.1 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.

10.69.2 Member Function Documentation

template<typename D1 , typename D2 > void Parma_Polyhedra_Library::No_Reduction< D1, D2 >::product_reduce (D1 & d1, D2 & d2) The null reduction operator.

The parameters d1 and d2 are ignored.

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

- ppl.hh

10.70 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** &y, **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 **m_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`.*

- template<typename Interval_Info >
void `refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< T, Interval_Info > > &`left`,
const `Linear_Form`< `Interval`< T, Interval_Info > > &`right`)

*Refines the system of octagonal constraints defining `*this` using the constraint expressed by `left` \leq `right`.*

- template<typename Interval_Info >
void `generalized_refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< T, Interval_Info > > &`left`, const `Linear_Form`< `Interval`< T, Interval_Info > > &`right`, `Relation_Symbol` `relysym`)

*Refines the system of octagonal constraints defining `*this` using the constraint expressed by `left` \bowtie `right`, where \bowtie is the relation symbol specified by `relysym`.*

- 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`.*

- template<typename Interval_Info >
void `affine_form_image` (`Variable` `var`, const `Linear_Form`< `Interval`< T, Interval_Info > > &`lf`)

*Assigns to `*this` the *affine form image* of `*this` under the function mapping variable `var` into the affine expression(s) specified by `lf`.*

- 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 $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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 $\text{lhs}' \bowtie \text{rhs}$, 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 $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.*
- 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 $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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 $\text{lhs}' \bowtie \text{rhs}$, 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 $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.*
- 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 *cs_p=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 U >
void [export_interval_constraints](#) (U &dest) const
Applies to *dest* the interval constraints embedded in **this*.
- 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.

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 >
void [swap](#) ([Octagonal_Shape](#)< T > &x, [Octagonal_Shape](#)< T > &y)
Swaps *x* with *y*.

- `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 the same octagon.
- `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)`
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)`
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)`
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 >`
`bool operator== (const Octagonal_Shape< T > &x, const Octagonal_Shape< T > &y)`
- `template<typename T >`
`bool operator!= (const Octagonal_Shape< T > &x, const Octagonal_Shape< T > &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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir, Temp &tmp0, Temp &tmp1, Temp &tmp2)`
- `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, const Rounding_Dir dir)`
- `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, const Rounding_Dir dir)`
- `template<typename T >`
`void swap (Octagonal_Shape< T > &x, Octagonal_Shape< T > &y)`
- `template<typename T >`
`std::ostream & operator<< (std::ostream &s, const Octagonal_Shape< T > &oct)`

10.70.1 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 \mathbb{T} , may lose 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 \mathbb{T} is a (bounded or unbounded) integer type;
- $x + y \leq \frac{1}{3}$, if \mathbb{T} is the unbounded rational type `mpq_class`;
- $x + y \leq k$, where $k > \frac{1}{3}$, if \mathbb{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 \mathbb{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);
```

10.70.2 Constructor & Destructor Documentation

```
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

<i>num_-dimensions</i>	The number of dimensions of the vector space enclosing the OS;
<i>kind</i>	Specifies whether the universe or the empty OS has to be built.

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

Ordinary copy constructor.

The complexity argument is ignored.

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.

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

<i>cs</i>	A system of octagonal constraints.
-----------	------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if cs contains a constraint which is not optimally supported by the Octagonal shape domain.
------------------------------	--

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

<i>cgs</i>	A system of congruences.
------------	--------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if cgs contains a congruence which is not optimally supported by the Octagonal shape domain.
------------------------------	---

template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Generator_System &gs) [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

<i>std::invalid_argument</i>	Thrown if the system of generators is not empty but has no points.
------------------------------	--

template<typename T > Parma_Polyhedra_Library::Octagonal_Shape< T >::Octagonal_Shape (const Polyhedron &ph, Complexity_Class *complexity* = ANY_COMPLEXITY) [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`.

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

<i>box</i>	The box representing the OS to be built.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>box</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>grid</i>	The grid used to build the OS.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>grid</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>bd</i>	The BD shape used to build the OS.
<i>complexity</i>	This argument is ignored as the algorithm used has polynomial complexity.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>bd</code> exceeds the maximum allowed space dimension.
--------------------------	---

10.70.3 Member Function Documentation

template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::contains (const Octagonal_Shape< T > & *y*) const Returns `true` if and only if `*this` contains `y`.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T> bool Parma_Polyhedra_Library::Octagonal_Shape< T >::is_disjoint_from (const Octagonal_Shape< T > & y) const Returns `true` if and only if **this* and *y* are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

template<typename T> Poly_Con_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Constraint & c) const Returns the relations holding between **this* and the constraint *c*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

template<typename T> Poly_Con_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Congruence & cg) const Returns the relations holding between **this* and the congruence *cg*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cg</i> are dimension-incompatible.
------------------------------	--

template<typename T> Poly_Gen_Relation Parma_Polyhedra_Library::Octagonal_Shape< T >::relation_with (const Generator & g) const Returns the relations holding between **this* and the generator *g*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible.
------------------------------	---

template<typename T> bool Parma_Polyhedra_Library::Octagonal_Shape< T >::constrains (Variable var) const Returns `true` if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be maximized subject to <code>*this</code> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value;
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <code>expr</code> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if and only if the infimum is also the minimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if and only if the infimum is also the minimum value;
<i>g</i>	When minimization succeeds, will be assigned a point or closure point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

Parameters

<i>expr</i>	The linear expression for which the frequency is needed;
<i>freq_n</i>	If <code>true</code> is returned, the value is set to 0; Present for interface compatibility with class Grid , where the frequency can have a non-zero value;
<i>freq_d</i>	If <code>true</code> is returned, the value is set to 1;
<i>val_n</i>	The numerator of <i>val</i> ;
<i>val_d</i>	The denominator of <i>val</i> ;

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_constraint (const Constraint & *c*) Adds a copy of constraint *c* to the system of constraints defining **this*.

Parameters

<i>c</i>	The constraint to be added.
----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible, or <i>c</i> is not optimally supported by the OS domain.
------------------------------	---

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

<i>cs</i>	The constraints that will be added.
-----------	-------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible, or <i>cs</i> contains a constraint which is not optimally supported by the OS domain.
------------------------------	--

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

<i>cs</i>	The constraint system to be added to <i>*this</i> . The constraints in <i>cs</i> may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible, or <i>cs</i> 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.

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_congruence (const Congruence & cg) Adds to **this* a constraint equivalent to the congruence *cg*.

Parameters

<i>cg</i>	The congruence to be added.
-----------	-----------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible, or <i>cg</i> is not optimally supported by the OS domain.
------------------------------	---

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

<i>cgs</i>	The congruences to be added.
------------	------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or <i>cgs</i> contains a congruence which is not optimally supported by the OS domain.
------------------------------	--

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

<i>cgs</i>	The congruence system to be added to <i>*this</i> . The congruences in <i>cgs</i> may be recycled.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or <i>cgs</i> 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.

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

<i>c</i>	The constraint. If it is not a octagonal constraint, it will be ignored.
----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

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

<i>cg</i>	The congruence. If it is not a octagonal equality, it will be ignored.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

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

<i>cs</i>	The constraint system to be used. Constraints that are not octagonal are ignored.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_congruences (const Congruence_System & cgs) Uses a copy of the congruences in *cgs* to refine the system of octagonal constraints defining **this*.

Parameters

<i>cgs</i>	The congruence system to be used. Congruences that are not octagonal equalities are ignored.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::Octagonal_Shape< T >::refine_with_linear_form_inequality (const Linear_Form< Interval< T, Interval_Info > > & left, const Linear_Form< Interval< T, Interval_Info > > & right) Refines the system of octagonal constraints defining **this* using the constraint expressed by $\text{left} \leq \text{right}$.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is at the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is at the right of the comparison operator. All of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>left</i> (or <i>right</i>) is dimension-incompatible with <i>*this</i> .
------------------------------	--

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by *left* and *right* respectively.

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::Octagonal_Shape< T >::generalized_refine_with_linear_form_inequality (const Linear_Form< Interval< T, Interval_Info > > & left, const Linear_Form< Interval< T, Interval_Info > > & right, Relation_Symbol relsym) [inline] Refines the system of octagonal constraints defining **this* using the constraint expressed by $\text{left} \bowtie \text{right}$, where \bowtie is the relation symbol specified by *relsym*.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is at the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is at the right of the comparison operator. All of its coefficients MUST be bounded.
<i>relsym</i>	The relation symbol.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>left</i> (or <i>right</i>) is dimension-incompatible with <i>*this</i> .
<i>std::runtime_error</i>	Thrown if <i>relsym</i> is not a valid relation symbol.

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by *left* and *right* respectively.

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::unconstrain (Variable var) Computes the [cylindrification](#) of **this* with respect to space dimension *var*, assigning the result to **this*.

Parameters

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::intersection.-assign (const Octagonal_Shape< T > & y) Assigns to **this* the intersection of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::upper_bound.-assign (const Octagonal_Shape< T > & y) Assigns to **this* the smallest OS that contains the convex union of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::upper_bound.-assign_if_exact (const Octagonal_Shape< T > & 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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

Implementation is based on Theorem 6.3 of [\[BHZ09b\]](#).

template<typename T > bool Parma_Polyhedra_Library::Octagonal_Shape< T >::integer_upper.-bound_assign_if_exact (const Octagonal_Shape< T > & 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.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

Note

This operator is only available when the class template parameter `T` is bound to an integer data type. 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].

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::difference_assign (const Octagonal_Shape< T > & y) Assigns to `*this` the smallest octagon containing the set difference of `*this` and `y`.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

template<typename T> bool Parma_Polyhedra_Library::Octagonal_Shape< T >::simplify_using_-context_assign (const Octagonal_Shape< T > & y) Assigns to `*this` a [meet-preserving simplification](#) of `*this` with respect to `y`. If `false` is returned, then the intersection is empty.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are topology-incompatible or dimension-incompatible.
------------------------------	--

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()) Assigns to `*this` the [affine image](#) of `*this` under the function mapping variable `var` into the affine expression specified by `expr` and `denominator`.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>expr</i>	The numerator of the affine expression.
<i>denominator</i>	The denominator of the affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>denominator</code> is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a dimension of <code>*this</code> .
------------------------------	--

template<typename T> template<typename Interval_Info> void Parma_Polyhedra_Library::Octagonal_Shape< T >::affine_form_image (Variable var, const Linear_Form< Interval< T, Interval_Info > > & lf) Assigns to `*this` the [affine form image](#) of `*this` under the function mapping variable `var` into the affine expression(s) specified by `lf`.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>lf</i>	The linear form on intervals with floating point boundaries that defines the affine expression(s). ALL of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>lf</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a dimension of <code>*this</code> .
------------------------------	---

This function is used in abstract interpretation to model an assignment of a value that is correctly overapproximated by `lf` to the floating point variable represented by `var`.

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()) Assigns to **this* the [affine preimage](#) of **this* under the function mapping variable *var* into the affine expression specified by *expr* and *denominator*.

Parameters

<i>var</i>	The variable to which the affine expression is substituted.
<i>expr</i>	The numerator of the affine expression.
<i>denominator</i>	The denominator of the affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> .
------------------------------	--

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()) Assigns to **this* the image of **this* with respect to the [generalized affine transfer function](#) $var' \bowtie \frac{expr}{denominator}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine transfer function.
<i>relsym</i>	The relation symbol.
<i>expr</i>	The numerator of the right hand side affine expression.
<i>denominator</i>	The denominator of the right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T>::generalized_affine_image (const Linear_Expression & lhs, Relation_Symbol relsym, const Linear_Expression & rhs) Assigns to **this* the image of **this* with respect to the [generalized affine transfer function](#) $lhs' \bowtie rhs$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression.
<i>relsym</i>	The relation symbol.
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

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()) Assigns to **this* the image of **this* with respect to the [bounded affine relation](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the preimage of **this* with respect to the [affine relation](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine transfer function.
<i>relsym</i>	The relation symbol.
<i>expr</i>	The numerator of the right hand side affine expression.
<i>denominator</i>	The denominator of the right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a dimension of <i>*this</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>relsym</i> is a strict relation symbol.
------------------------------	---

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 ()) Assigns to **this* the preimage of **this* with respect to the [bounded affine relation](#) $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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 * cs_p = 0, unsigned complexity_threshold = 16, bool wrap_individually = true) Wraps the specified dimensions of the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	Possibly null pointer to a constraint system whose variables are contained in <i>vars</i> . If <i>*cs_p</i> depends on variables not in <i>vars</i> , 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 <i>*cs_p</i> .
<i>complexity_threshold</i>	A precision parameter of the wrapping operator : higher values result in possibly improved precision.
<i>wrap_individually</i>	true if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*cs_p</i> is dimension-incompatible with <i>vars</i> , or if <i>*this</i> is dimension-incompatible <i>vars</i> or with <i>*cs_p</i> .
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::drop_some_non_integer_points (Complexity_Class complexity = ANY_COMPLEXITY) Possibly tightens **this* by dropping some points with non-integer coordinates.

Parameters

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

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

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T>::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`.

Parameters

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

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

<i>y</i>	An OS that <i>must</i> be contained in <code>*this</code> .
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

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) Assigns to `*this` the result of computing the [CC76-extrapolation](#) between `*this` and `y`.

Parameters

<i>y</i>	An OS that <i>must</i> be contained in <code>*this</code> .
<i>first</i>	An iterator that points to the first stop_point.
<i>last</i>	An iterator that points to the last stop_point.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T>::BHMZ05_widening-assign (const Octagonal_Shape< T> & y, unsigned *tp = 0) Assigns to `*this` the result of computing the [BHMZ05-widening](#) between `*this` and `y`.

Parameters

<i>y</i>	An OS that <i>must</i> be contained in <i>*this</i> .
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::limited_BHM-Z05_extrapolation_assign (const Octagonal_Shape< T > & y, const Constraint_System & cs, unsigned * tp = 0) Improves the result of the [BHMZ05-widening](#) computation by also enforcing those constraints in *cs* that are satisfied by all the points of **this*.

Parameters

<i>y</i>	An OS that <i>must</i> be contained in <i>*this</i> .
<i>cs</i>	The system of constraints used to improve the widened OS.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible or if there is in <i>cs</i> a strict inequality.
------------------------------	---

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::CC76_narrowing_assign (const Octagonal_Shape< T > & y) Restores from *y* the constraints of **this*, lost by [CC76-extrapolation](#) applications.

Parameters

<i>y</i>	An OS that <i>must</i> contain <i>*this</i> .
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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) 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

<i>y</i>	An OS that <i>must</i> be contained in <i>*this</i> .
<i>cs</i>	The system of constraints used to improve the widened OS.
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are dimension-incompatible or if <i>cs</i> contains a strict inequality.
------------------------------	--

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_space_dimensions_and_embed (dimension_type m) Adds *m* new dimensions and embeds the old OS into the new space.

Parameters

<i>m</i>	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

$$\{ (x, y, z)^T \in \mathbb{R}^3 \mid (x, y)^T \in \mathcal{O} \}.$$

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::add_space_dimensions_and_project (dimension_type m) Adds *m* new dimensions to the OS and does not embed it in the new space.

Parameters

<i>m</i>	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

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

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::concatenate_and_assign (const Octagonal_Shape< T > & y) Assigns to **this* the [concatenation](#) of **this* and *y*, taken in this order.

Exceptions

<i>std::length_error</i>	Thrown if the concatenation would cause the vector space to exceed dimension max_space_dimension .
--------------------------	--

template<typename T > void Parma_Polyhedra_Library::Octagonal_Shape< T >::remove_space_dimensions (const Variables.Set & vars) Removes all the specified dimensions.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the dimensions to be removed.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <i>new_dimension</i> is greater than the space dimension of <i>*this</i> .
------------------------------	--

template<typename T > template<typename Partial_Function > void Parma_Polyhedra_Library::Octagonal_Shape< T >::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a [partial function](#).

Parameters

<i>pfunc</i>	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](#).

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::expand_space.-dimension (Variable var, dimension_type m) Creates `m` copies of the space dimension corresponding to `var`.

Parameters

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding <code>m</code> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

template<typename T> void Parma_Polyhedra_Library::Octagonal_Shape< T >::fold_space_dimensions (const Variables_Set & vars, Variable dest) Folds the space dimensions in `vars` into `dest`.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> or with one of the Variable objects contained in <code>vars</code> . Also thrown if <code>dest</code> is contained in <code>vars</code> .
------------------------------	---

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.

template<typename T> template<typename U> void Parma_Polyhedra_Library::Octagonal_Shape< T >::export_interval_constraints (U & dest) const Applies to `dest` the interval constraints embedded in `*this`.

Parameters

<i>dest</i>	The object to which the constraints will be added.
-------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>dest</i> .
------------------------------	---

The template type parameter *U* must provide the following methods.

```
dimension_type space_dimension() const
```

returns the space dimension of the object.

```
void set_empty()
```

sets the object to an empty object.

```
bool restrict_lower(dimension_type dim, const T& lb)
```

restricts the object by applying the lower bound *lb* to the space dimension *dim* and returns *false* if and only if the object becomes empty.

```
bool restrict_upper(dimension_type dim, const T& ub)
```

restricts the object by applying the upper bound *ub* to the space dimension *dim* and returns *false* if and only if the object becomes empty.

```
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

<i>store</i>	The interval floating point abstract store to refine.
--------------	---

```
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().
```

10.70.4 Friends And Related Function Documentation

```
template<typename T> std::ostream & operator<< ( std::ostream & s, const Octagonal_Shape<
T> & oct ) [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 “,”.

```
template<typename T> void swap ( Octagonal_Shape< T> & x, Octagonal_Shape< T> & y )
[related] Swaps x with y.
```

```
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 the same octagon.
```

Note that *x* and *y* may be dimension-incompatible shapes: in this case, the value *false* is returned.

```
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.

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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

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>`.

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) [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<Temp, Extended_Number_Policy>`.

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`.

template<typename T > bool operator== (const Octagonal_Shape< T > & x, const Octagonal_Shape< T > & y) [related]

template<typename T > bool operator!= (const Octagonal_Shape< T > & x, const Octagonal_Shape< T > & y) [related]

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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2) [related]

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, const Rounding_Dir dir) [related]

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, const Rounding_Dir dir) [related]

```
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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 )
[related]
```

```
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, const Rounding_Dir dir ) [related]
```

```
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, const Rounding_Dir dir ) [related]
```

```
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, const Rounding_Dir dir, Temp & tmp0, Temp & tmp1, Temp & tmp2 )
[related]
```

```
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, const Rounding_Dir dir ) [related]
```

```
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, const Rounding_Dir dir ) [related]
```

```
template<typename T > void swap ( Octagonal_Shape< T > & x, Octagonal_Shape< T > & y )
[related]
```

```
template<typename T > std::ostream & operator<< ( std::ostream & s, const Octagonal_Shape<
T > & oct ) [related] The documentation for this class was generated from the following file:
```

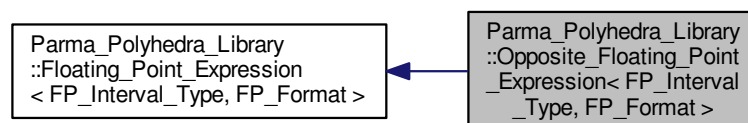
- ppl.hh

10.71 Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval- _Type, FP_Format > Class Template Reference

A generic Opposite Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval-
_Type, FP_Format >:



Public Types

- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Linear_Form FP_Linear_Form
Alias for the Linear_Form<FP.Interval_Type> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Interval_Abstract_Store FP_Interval_Abstract_Store
Alias for the std::map<dimension_type, FP.Interval_Type> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::FP_Linear_Form_Abstract_Store FP_Linear_Form_Abstract_Store
Alias for the std::map<dimension_type, FP.Linear_Form> from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::boundary_type boundary_type
Alias for the FP.Interval_Type::boundary_type from Floating_Point_Expression.
- typedef
Floating_Point_Expression
< FP.Interval_Type, FP.Format >
::info_type info_type
Alias for the FP.Interval_Type::info_type from Floating_Point_Expression.

Public Member Functions

- bool linearize (const FP.Interval_Abstract_Store &int_store, const FP.Linear_Form_Abstract_Store &lf_store, FP.Linear_Form &result) const
Linearizes the expression in a given astract store.
- void m_swap (Opposite_Floating_Point_Expression &y)
*Swaps *this with y.*

Constructors and Destructor

- Opposite_Floating_Point_Expression (Floating_Point_Expression< FP.Interval_Type, FP.Format > *const op)
Constructor with one parameter: builds the opposite floating point expression $\ominus op$.
- ~Opposite_Floating_Point_Expression ()
Destructor.

Related Functions

(Note that these are not member functions.)

- template<typename FP.Interval_Type, typename FP.Format >
void swap (Opposite_Floating_Point_Expression< FP.Interval_Type, FP.Format > &x, Opposite_Floating_Point_Expression< FP.Interval_Type, FP.Format > &y)
Swaps x with y.
- template<typename FP.Interval_Type, typename FP.Format >
void swap (Opposite_Floating_Point_Expression< FP.Interval_Type, FP.Format > &x, Opposite_Floating_Point_Expression< FP.Interval_Type, FP.Format > &y)

Additional Inherited Members

10.71.1 Detailed Description

template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Opposite-Floating_Point_Expression< FP_Interval_Type, FP_Format >

A generic Opposite Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of opposite floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ be an interval linear form and let $\boxminus^\#$ be a sound unary operator on linear forms such that:

$$\boxminus^\# \left(i + \sum_{v \in \mathcal{V}} i_v v \right) = (\boxminus^\# i) + \sum_{v \in \mathcal{V}} (\boxminus^\# i_v) v,$$

Given a floating point expression $\ominus e$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket \ominus e \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket \ominus e \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \boxminus^\# \left(\llbracket e \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right).$$

10.71.2 Member Function Documentation

template<typename FP_Interval_Type , typename FP_Format > bool Parma_Polyhedra_Library::Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const [**inline**], [**virtual**] Linearizes the expression in a given astract store.

Makes *result* become the linearization of **this* in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

Note that all variables occuring in the expression represented by *operand* MUST have an associated value in *int_store*. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how *result* is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.71.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) [**related**] Swaps *x* with *y*.

`template<typename FP_Interval_Type , typename FP_Format > void swap (Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Opposite_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y)` [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.72 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` &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 `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* $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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* $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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* $\text{lhs}' \bowtie \text{rhs}$, 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* $\text{lhs}' \bowtie \text{rhs}$, 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* $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
- 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* $\text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}} \leq \frac{\text{lb_expr}}{\text{denominator}}$.
- 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 [m_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.

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 >
void `swap` (`Partially_Reduced_Product`< D1, D2, R > &x, `Partially_Reduced_Product`< D1, D2, R > &y)
Swaps `x` with `y`.
- 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 pairwise equal.
- 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 >
bool `operator==` (const `Partially_Reduced_Product`< D1, D2, R > &x, const `Partially_Reduced_Product`< D1, D2, R > &y)
- 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)
- template<typename D1 , typename D2 , typename R >
std::ostream & `operator<<` (std::ostream &s, const `Partially_Reduced_Product`< D1, D2, R > &dp)
- template<typename D1 , typename D2 , typename R >
void `swap` (`Partially_Reduced_Product`< D1, D2, R > &x, `Partially_Reduced_Product`< D1, D2, R > &y)

10.72.1 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: $\gamma_1: D_1 \rightarrow \mathbb{R}^n$ and $\gamma_2: D_2 \rightarrow \mathbb{R}^n$, respectively.

The partially reduced product $D = D_1 \times D_2$, for any reduction class R, has a concretization $\gamma: D \rightarrow \mathbb{R}^n$ 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 % 2) / 2);
cgs.insert((y % 2) / 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 % 2) / 2);
dp.add_congruence((y % 2) / 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;
```

10.72.2 Constructor & Destructor Documentation

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

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the pair;
<i>kind</i>	Specifies whether a universe or an empty pair has to be built.

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cgs</i>	The system of congruences to be approximated by the pair.
------------	---

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cgs</i>	The system of congruences to be approximates by the pair. Its data-structures may be recycled to build the pair.
------------	--

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cs</i>	The system of constraints to be approximated by the pair.
-----------	---

Exceptions

<i>std::length_error</i>	Thrown if <code>num_dimensions</code> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>cs</i>	The system of constraints to be approximated by the pair.
-----------	---

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>cs</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>ph</i>	The polyhedron to be approximated by the product.
<i>complexity</i>	The complexity that will not be exceeded.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>ph</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>ph</i>	The polyhedron to be approximated by the product.
<i>complexity</i>	The complexity that will not be exceeded.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>ph</i> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>gr</i>	The grid to be approximated by the product.
<i>complexity</i>	The complexity is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>gr</i> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>box</i>	The box representing the pair to be built.
<i>complexity</i>	The complexity is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>box</i> exceeds the maximum allowed space dimension.
--------------------------	--

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

<i>bd</i>	The BD shape representing the product to be built.
<i>complexity</i>	The complexity is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>bd</code> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>os</i>	The octagonal shape representing the product to be built.
<i>complexity</i>	The complexity is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <code>os</code> exceeds the maximum allowed space dimension.
--------------------------	---

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.

10.72.3 Member Function Documentation

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

<i>std::invalid_argument</i>	Thrown if <code>x</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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 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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if the supremum value can be reached in <i>this</i> .

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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 & *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.

Parameters

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if the supremum value can be reached in <i>this</i> .
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <i>expr</i> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

If **this* is empty or *expr* is not bounded by **this*, `false` is returned and *sup_n*, *sup_d*, *maximum* and *g* are left untouched.

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 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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if the infimum value can be reached in <i>this</i> .

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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 & *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.

Parameters

<i>expr</i>	The linear expression to be minimized subject to <code>*this</code> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	<code>true</code> if the infimum value can be reached in <code>this</code> .
<i>g</i>	When minimization succeeds, will be assigned the point or closure point where <code>expr</code> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>expr</code> and <code>*this</code> 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.

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

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>c</code> are dimension-incompatible.
------------------------------	---

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

<i>c</i>	The constraint to be used for refinement.
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

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

<i>cg</i>	The congruence to be used for refinement.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cg</i> are dimension-incompatible.
------------------------------	--

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

<i>cgs</i>	The congruence system to be added.
------------	------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

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

<i>cgs</i>	The congruences to be used for refinement.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_recycled_congruences (Congruence_System & *cgs*) Adds the congruences in *cgs* to **this*.

Parameters

<i>cgs</i>	The congruence system to be added that may be recycled.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> 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.

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

<i>cs</i>	The constraint system to be added.
-----------	------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

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

<i>cs</i>	The constraints to be used for refinement.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::add_recycled_constraints (Constraint_System & cs) Adds the constraint system in *cs* to **this*.

Parameters

<i>cs</i>	The constraint system to be added that may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> 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.

template<typename D1 , typename D2 , typename R > void Parma_Polyhedra_Library::Partially_Reduced_Product< D1, D2, R >::unconstrain (Variable var) [inline] Computes the [cylindrication](#) of **this* with respect to space dimension *var*, assigning the result to **this*.

Parameters

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>var</i>	The variable to which the affine expression is substituted;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relsym* (see also [generalized affine relation](#).)

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> or if <code>*this</code> is a C_Polyhedron and <code>relsym</code> is a strict relation symbol.
------------------------------	---

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](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by `relsym`. (see also [generalized affine relation](#).)

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> or if <code>*this</code> is a C_Polyhedron and <code>relsym</code> is a strict relation symbol.
------------------------------	---

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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by `relsym`. (see also [generalized affine relation](#).)

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>lhs</code> or <code>rhs</code> or if <code>*this</code> is a C_Polyhedron and <code>relsym</code> is a strict relation symbol.
------------------------------	--

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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by `relsym`. (see also [generalized affine relation](#).)

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>*this</i> is a C_Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>y</i>	A product that <i>must</i> be contained in <i>*this</i> ;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

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

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

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

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

<i>m</i>	The number of dimensions to add.
----------	----------------------------------

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> new space dimensions would cause the vector space to exceed dimension max_space_dimension() .
--------------------------	---

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

<i>m</i>	The number of space dimensions to add.
----------	--

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .
--------------------------	--

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

<i>std::length_error</i>	Thrown if the concatenation would cause the vector space to exceed dimension <code>max_space_dimension()</code> .
--------------------------	---

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

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be removed.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <i>new_dimensions</i> is greater than the space dimension of <i>*this</i> .
------------------------------	---

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

<i>pfunc</i>	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](#).

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

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding <code>m</code> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

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

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> or with one of the Variable objects contained in <code>vars</code> . Also thrown if <code>dest</code> is contained in <code>vars</code> .
------------------------------	---

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.

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()`.

10.72.4 Friends And Related Function Documentation

template<typename D1 , typename D2 , typename R > std::ostream & operator<< (std::ostream & s, const Partially_Reduced_Product< D1, D2, R > & dp) [related] Output operator.

Writes a textual representation of `dp` on `s`.

template<typename D1 , typename D2 , typename R > void swap (Partially_Reduced_Product< D1, D2, R > & x, Partially_Reduced_Product< D1, D2, R > & y) [related] Swaps `x` with `y`.

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 pairwise equal.

Note that `x` and `y` may be dimension-incompatible: in those cases, the value `false` is returned.

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.

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]

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]

template<typename D1 , typename D2 , typename R > std::ostream & operator<< (std::ostream & s, const Partially_Reduced_Product< D1, D2, R > & dp) [related]

template<typename D1 , typename D2 , typename R > void swap (Partially_Reduced_Product< D1, D2, R > & x, Partially_Reduced_Product< D1, D2, R > & y) [related] The documentation for this class was generated from the following file:

- `ppl.hh`

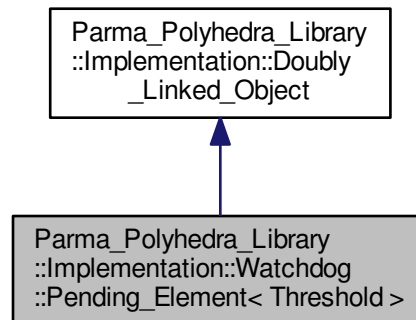
10.73 Parma Polyhedra Library::Implementation::Watchdog::Pending_Element< Threshold > Class Template Reference

A class for pending watchdog events with embedded links.

`#include <ppl.hh>`

Inheritance diagram for `Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< Thresh-`

old >:



Public Member Functions

- **Pending_Element** (const Threshold &**deadline**, const **Handler** &**handler**, bool &**expired_flag**)
Constructs an element with the given attributes.
- void **assign** (const Threshold &**deadline**, const **Handler** &**handler**, bool &**expired_flag**)
*Modifies *this so that it has the given attributes.*
- const Threshold & **deadline** () const
Returns the deadline of the event.
- const **Handler** & **handler** () const
Returns the handler associated to the event.
- bool & **expired_flag** () const
Returns a reference to the "event-expired" flag.
- bool **OK** () const
Checks if all the invariants are satisfied.

10.73.1 Detailed Description

template<class Threshold>class Parma_Polyhedra_Library::Implementation::Watchdog::Pending_Element< Threshold >

A class for pending watchdog events with embedded links.

Each pending watchdog event is characterized by a deadline (a positive time interval), an associated handler that will be invoked upon event expiration, and a Boolean flag that indicates whether the event has already expired or not.

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

- ppl.hh

10.74 Parma_Polyhedra_Library::Implementation::Watchdog::Pending_List< Traits > Class Template Reference

An ordered list for recording pending watchdog events.

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

Public Types

- typedef [EList](#)< [Pending_Element](#)
< typename Traits::Threshold >
>::iterator iterator
A non-const iterator to traverse the list.
- typedef [EList](#)< [Pending_Element](#)
< typename Traits::Threshold >
>::const_iterator const_iterator
A const iterator to traverse the list.

Public Member Functions

- [Pending_List](#) ()
Constructs an empty list.
- [~Pending_List](#) ()
Destructor.
- [iterator insert](#) (const typename Traits::Threshold &deadline, const [Handler](#) &handler, bool &expired-
_flag)
Inserts a new [Pending_Element](#) object with the given attributes.
- [iterator erase](#) ([iterator](#) position)
Removes the element pointed to by `position`, returning an iterator pointing to the next element, if any, or `end()`, otherwise.
- bool [empty](#) () const
Returns `true` if and only if the list is empty.
- [iterator begin](#) ()
Returns an iterator pointing to the beginning of the list.
- [iterator end](#) ()
Returns an iterator pointing one past the last element in the list.
- bool [OK](#) () const
Checks if all the invariants are satisfied.

10.74.1 Detailed Description

template<typename Traits>class Parma_Polyhedra_Library::Implementation::Watchdog::Pending_List< Traits >

An ordered list for recording pending watchdog events.

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

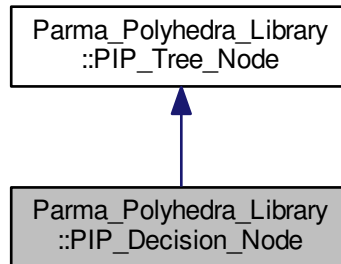
- ppl.hh

10.75 Parma_Polyhedra_Library::PIP_Decision_Node Class Reference

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

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

Inheritance diagram for Parma_Polyhedra_Library::PIP_Decision_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`.
- virtual `const PIP_Solution_Node * as_solution ()` const
Returns 0, since `this` is not a solution node.
- `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` ¶meters)

Implements pure virtual method `PIP_Tree_Node::update_tableau`.

- virtual `PIP_Tree_Node` * `solve` (const `PIP_Problem` &pip, bool check_feasible_context, const `Matrix`< `Row` > &context, const `Variables_Set` ¶ms, `dimension_type` space_dim, int indent_level)

Implements pure virtual method `PIP_Tree_Node::solve`.

- virtual void `print_tree` (std::ostream &s, int indent, const std::vector< bool > &pip_dim_is_param, `dimension_type` first_art_dim) const

*Prints on `s` the tree rooted in `*this`.*

Additional Inherited Members

10.75.1 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`

10.76 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 `[first, 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, int 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 m_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` &pip)
Output operator.
- void `swap` (`PIP_Problem` &x, `PIP_Problem` &y)
Swaps `x` with `y`.
- void `swap` (`PIP_Problem` &x, `PIP_Problem` &y)

10.76.1 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 $7*n \geq 10$. 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 constraints 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 recursively 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);
ConstraintSystem 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:

$$\begin{cases} x' = M - \lfloor \frac{p+1}{3} \rfloor - 1 \\ y' = M - p + \lfloor \frac{p+1}{3} \rfloor + 1 \end{cases}$$

Performing substitution again provides us with the values of the original variables:

$$\begin{cases} x = \lfloor \frac{p+1}{3} \rfloor + 1 \\ y = p - \lfloor \frac{p+1}{3} \rfloor - 1 \end{cases}$$

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);
ConstraintSystem 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 $p_i^+ - p_i^-$, 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, \dots, x_n , and the parameters p_1, p_2, \dots, p_m . You can minimize a linear cost function $f(x_2, \dots, x_n, p_1, \dots, p_m)$ by simply adding the constraint $x_1 \geq f(x_2, \dots, x_n, p_1, \dots, p_m)$ 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.

10.76.2 Member Enumeration Documentation

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.

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 a row that generates a lexicographically maximal pivot column.

10.76.3 Constructor & Destructor Documentation

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

<i>dim</i>	The dimension of the vector space enclosing <i>*this</i> (optional argument with default value 0).
------------	--

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds max_space_dimension() .
--------------------------	--

template<typename In > Parma_Polyhedra_Library::PIP_Problem::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 [*first*, *last*); those dimensions whose indices occur in *p_vars* are interpreted as parameters.

Parameters

<i>dim</i>	The dimension of the vector space (variables and parameters) enclosing <i>*this</i> .
<i>first</i>	An input iterator to the start of the sequence of constraints.
<i>last</i>	A past-the-end input iterator to the sequence of constraints.
<i>p_vars</i>	The set of variables' indexes that are interpreted as parameters.

Exceptions

<i>std::length_error</i>	Thrown if <i>dim</i> exceeds max_space_dimension() .
<i>std::invalid_argument</i>	Thrown if the space dimension of a constraint in the sequence (resp., the parameter variables) is strictly greater than <i>dim</i> .

10.76.4 Member Function Documentation

void Parma_Polyhedra_Library::PIP_Problem::clear () Resets **this* to be equal to the trivial PIP problem.

The space dimension is reset to 0.

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

<i>m_vars</i>	The number of space dimensions to add that are interpreted as PIP problem variables (i.e., non parameters). These are added <i>before</i> adding the <i>m_params</i> parameters.
<i>m_params</i>	The number of space dimensions to add that are interpreted as PIP problem parameters. These are added <i>after</i> having added the <i>m_vars</i> problem variables.

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m_vars</i> + <i>m_params</i> 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.

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

<i>std::invalid_argument</i>	Thrown if some index in <code>p_vars</code> does not correspond to a space dimension in <code>*this</code> .
------------------------------	--

void Parma_Polyhedra_Library::PIP_Problem::add_constraint (const Constraint & c) Adds a copy of constraint `c` to the PIP problem.

Exceptions

<i>std::invalid_argument</i>	Thrown if the space dimension of <code>c</code> is strictly greater than the space dimension of <code>*this</code> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if the space dimension of constraint system <code>cs</code> is strictly greater than the space dimension of <code>*this</code> .
------------------------------	---

bool Parma_Polyhedra_Library::PIP_Problem::is_satisfiable () const Checks satisfiability of `*this`.

Returns

`true` if and only if the PIP problem is satisfiable.

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).

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.

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.

void Parma_Polyhedra_Library::PIP_Problem::print_solution (std::ostream & s, int indent = 0) const Prints on `s` the solution computed for `*this`.

Parameters

<i>s</i>	The output stream.
<i>indent</i>	An indentation parameter (default value 0).

Exceptions

<code>std::logic_error</code>	Thrown if trying to print the solution when the PIP problem still has to be solved.
-------------------------------	---

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()`.

10.76.5 Friends And Related Function Documentation

std::ostream & operator<< (std::ostream & s, const PIP_Problem & pip) **[related]** Output operator.

void swap (PIP_Problem & x, PIP_Problem & y) **[related]** Swaps `x` with `y`.

void swap (PIP_Problem & x, PIP_Problem & y) **[related]** The documentation for this class was generated from the following file:

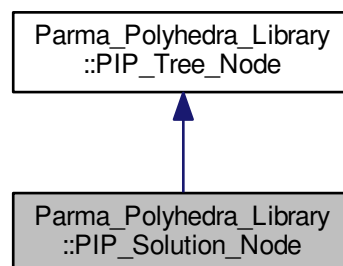
- `ppl.hh`

10.77 Parma_Polyhedra_Library::PIP_Solution_Node Class Reference

A tree node representing part of the space of solutions.

`#include <ppl.hh>`

Inheritance diagram for Parma_Polyhedra_Library::PIP_Solution_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.
- virtual const [PIP_Decision_Node](#) * [as_decision](#) () const
Returns 0, since this is not a decision node.
- const [Linear_Expression](#) & [parametric_values](#) ([Variable](#) var) const
Returns a parametric expression for the values of problem variable var.
- void [ascii_dump](#) (std::ostream &os) const
*Dumps to os an ASCII representation of *this.*
- bool [ascii_load](#) (std::istream &is)
*Loads from is 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](#) ¶meters)
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](#)< [Row](#) > &context, const [Variables_Set](#) ¶ms, [dimension_type](#) space_dim, int indent_level)
Implements pure virtual method [PIP_Tree_Node::solve](#).
- void [generate_cut](#) ([dimension_type](#) index, [Variables_Set](#) ¶meters, [Matrix](#)< [Row](#) > &context, [dimension_type](#) &space_dimension, int indent_level)

Generate a Gomory cut using non-integer tableau row `index`.

- virtual void [print_tree](#) (std::ostream &s, int indent, const std::vector< bool > &pip_dim_is_param, [dimension_type](#) first_art_dim) const

*Prints on `s` the tree rooted in `*this`.*

Additional Inherited Members

10.77.1 Detailed Description

A tree node representing part of the space of solutions.

10.77.2 Constructor & Destructor Documentation

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.

10.77.3 Member Function Documentation

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

<i>var</i>	The problem variable which is queried about.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> is dimension-incompatible with the PIP_Problem owning this solution node, or if <code>var</code> is a problem parameter.
------------------------------	---

void Parma_Polyhedra_Library::PIP_Solution_Node::update_solution (const std::vector< bool > &pip_dim_is_param) const [protected] Update the solution values.

Parameters

<i>pip_dim_is_param</i>	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).
-------------------------	--

void Parma_Polyhedra_Library::PIP_Solution_Node::generate_cut (dimension_type index, Variables_Set ¶meters, Matrix< Row > &context, dimension_type &space_dimension, int indent_level) [protected] Generate a Gomory cut using non-integer tableau row `index`.

Parameters

<i>index</i>	Row index in simplex tableau from which the cut is generated.
<i>parameters</i>	A std::set of the current parameter dimensions (including artificials); to be updated if a new artificial parameter is to be created.
<i>context</i>	A set of linear inequalities on the parameters, in matrix form; to be updated if a new artificial parameter is to be created.

<i>space_-dimension</i>	The current space dimension, including variables and all parameters; to be updated if an extra parameter is to be created.
<i>indent_level</i>	The indentation level (for debugging output only).

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

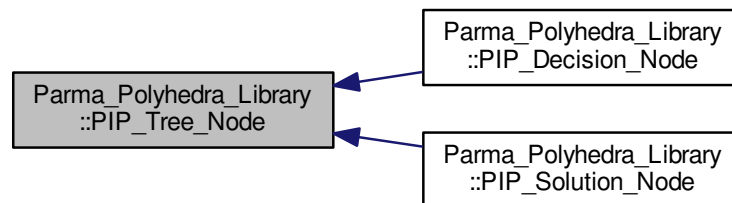
- ppl.hh

10.78 Parma_Polyhedra_Library::PIP_Tree_Node Class Reference

A node of the PIP solution tree.

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

Inheritance diagram for Parma_Polyhedra_Library::PIP_Tree_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 =0
*Returns true if and only if *this is well formed.*
- virtual const [PIP_Solution_Node](#) * [as_solution](#) () const =0
*Returns this if *this is a solution node, 0 otherwise.*
- virtual const [PIP_Decision_Node](#) * [as_decision](#) () const =0
*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, int 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 are owned by *owner.*
- 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](#) ¶meters)=0

Populates the parametric simplex tableau using external data.
- virtual [PIP_Tree_Node](#) * [solve](#) (const [PIP_Problem](#) &pip, bool check_feasible_context, const Matrix< Row > &context, const [Variables_Set](#) ¶ms, dimension_type space_dim, int indent_level)=0

Executes a parametric simplex on the tableau, under specified context.
- void [add_constraint](#) (const Row &row, const [Variables_Set](#) ¶meters)

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, int indent, const std::vector< bool > &pip_dim_is_param, [dimension_type](#) first_art_dim) const =0

*Prints on s the tree rooted in *this.*

Static Protected Member Functions

- static void [indent_and_print](#) (std::ostream &s, int indent, const char *str)

A helper function used when printing PIP trees.

- static bool [compatibility_check](#) (Matrix< Row > &s)

Checks whether a context matrix is satisfiable.

- static bool [compatibility_check](#) (const Matrix< Row > &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.

10.78.1 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.

10.78.2 Member Function Documentation

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.

void [Parma_Polyhedra_Library::PIP_Tree_Node::print](#) (std::ostream & s, int indent = 0) const

Prints on s the tree rooted in *this.

Parameters

<i>s</i>	The output stream.
<i>indent</i>	The amount of indentation.

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

<i>pip</i>	The PIP_Problem object containing this node.
<i>external_space_dim</i>	The number of all problem variables and problem parameters (excluding artificial parameters).
<i>first_pending_constraint</i>	The first element in <i>input_cs</i> to be added to the tableau, which already contains the previous elements.
<i>input_cs</i>	All the constraints of the PIP problem.
<i>parameters</i>	The set of indices of the problem parameters.

Implemented in [Parma_Polyhedra_Library::PIP_Decision_Node](#), and [Parma_Polyhedra_Library::PIP_Solution_Node](#).

virtual PIP_Tree_Node* Parma_Polyhedra_Library::PIP_Tree_Node::solve (const PIP_Problem & *pip*, bool *check_feasible_context*, const Matrix< Row > & *context*, const Variables_Set & *params*, dimension_type *space_dim*, int *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

<i>pip</i>	The PIP_Problem object containing this node.
<i>check_feasible_context</i>	Whether the resolution process should (re-)check feasibility of context (since the initial context may have been modified).
<i>context</i>	The context, being a set of constraints on the parameters.
<i>params</i>	The local parameter set, including parent's artificial parameters.
<i>space_dim</i>	The space dimension of parent, including artificial parameters.
<i>indent_level</i>	The indentation level (for debugging output only).

Implemented in [Parma_Polyhedra_Library::PIP_Decision_Node](#), and [Parma_Polyhedra_Library::PIP_Solution_Node](#).

virtual void Parma_Polyhedra_Library::PIP_Tree_Node::print_tree (std::ostream & *s*, int *indent*, const std::vector< bool > & *pip_dim_is_param*, dimension_type *first_art_dim*) const **[protected]**, **[pure virtual]** Prints on *s* the tree rooted in **this*.

Parameters

<i>s</i>	The output stream.
<i>indent</i>	The amount of indentation.

<i>pip_dim_is_param</i>	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).
<i>first_art_dim</i>	The first space dimension corresponding to an artificial parameter that was created in this node (if any).

Implemented in [Parma_Polyhedra_Library::PIP_Decision_Node](#), and [Parma_Polyhedra_Library::PIP_Solution_Node](#).

static bool Parma_Polyhedra_Library::PIP_Tree_Node::compatibility_check (Matrix< Row > & 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.

10.78.3 Friends And Related Function Documentation

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:

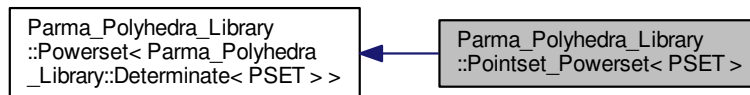
- ppl.hh

10.79 Parma_Polyhedra_Library::Pointset_Powerset< PSET > Class Template Reference

The powerset construction instantiated on PPL pointset domains.

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

Inheritance diagram for Parma_Polyhedra_Library::Pointset_Powerset< 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 powerset 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 &cg)
*Intersects *this with congruence cg.*
- 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 $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, 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* $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relysym*.
- void **generalized_affine_image** (const **Linear_Expression** &lhs, **Relation_Symbol** relysym, const **Linear_Expression** &rhs)
Assigns to **this* the image of **this* with respect to the *generalized affine relation* $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relysym*.
 - void **generalized_affine_preimage** (const **Linear_Expression** &lhs, **Relation_Symbol** relysym, const **Linear_Expression** &rhs)
Assigns to **this* the preimage of **this* with respect to the *generalized affine relation* $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relysym*.
 - 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* $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - 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* $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - 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** *cs_p=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 widen_fun, unsigned max_disjuncts)
Assigns to **this* the result of applying the *BGP99 extrapolation operator* to **this* and *y*, using the widening function *widen_fun* and the cardinality threshold *max_disjuncts*.
 - template<typename Cert, typename Widening >
void **BHZ03_widening_assign** (const **Pointset_Powerset** &y, Widening widen_fun)
Assigns to **this* the result of computing the *BHZ03-widening* between **this* and *y*, using the widening function *widen_fun* 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 **m_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 >
Widening_Function< PSET > [widen_fun_ref](#) (void(PSET::*wm)(const PSET &, unsigned *))
- template<typename PSET , typename CSYS >
Limited_Widening_Function
< PSET, CSYS > [widen_fun_ref](#) (void(PSET::*lwm)(const PSET &, const CSYS &, unsigned *),
const CSYS &cs)
- template<typename PSET >
void [swap](#) ([Pointset_Powerset](#)< PSET > &x, [Pointset_Powerset](#)< PSET > &y)
Swaps x with y.
- 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 >`
`bool check_containment (const PSET &ph, const Pointset_Powerset< PSET > &ps)`
- `template<>`
`bool check_containment (const C_Polyhedron &ph, const Pointset_Powerset< C_Polyhedron > &ps)`
- `template<typename PSET >`
`void swap (Pointset_Powerset< PSET > &x, Pointset_Powerset< PSET > &y)`
- `template<typename PSET >`
`std::pair< PSET,`
`Pointset_Powerset`
`< NNC_Polyhedron > > linear_partition (const PSET &p, const PSET &q)`

Additional Inherited Members

10.79.1 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>`.

10.79.2 Constructor & Destructor Documentation

`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

<i>num_dimensions</i>	The number of dimensions of the vector space enclosing the powerset;
<i>kind</i>	Specifies whether the universe or the empty powerset has to be built.

`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.

`template<typename PSET > template<typename QH > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Pointset_Powerset< QH > & y, Complexity_Class complexity = ANY_COMPLEXITY) [explicit]` Conversion constructor: the type `QH` of the disjuncts in the source powerset is different from `PSET`.

Parameters

<i>y</i>	The powerset to be used to build the new powerset.
<i>complexity</i>	The maximal complexity of any algorithms used.

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

<i>ph</i>	The closed polyhedron to be used to build the powerset.
<i>complexity</i>	The maximal complexity of any algorithms used.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>ph</i> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>ph</i>	The closed polyhedron to be used to build the powerset.
<i>complexity</i>	The maximal complexity of any algorithms used.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>ph</i> exceeds the maximum allowed space dimension.
--------------------------	---

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

<i>gr</i>	The grid to be used to build the powerset.
<i>complexity</i>	This argument is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>gr</i> exceeds the maximum allowed space dimension.
--------------------------	---

template<typename PSET > template<typename T > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Octagonal_Shape< T > & os, Complexity_Class complexity = ANY_COMPLEXITY) [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

<i>os</i>	The octagonal shape to be used to build the powerset.
<i>complexity</i>	This argument is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>os</i> exceeds the maximum allowed space dimension.
--------------------------	---

template<typename PSET > template<typename T > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const BD_Shape< T > & bds, Complexity_Class complexity = ANY_COMPLEXITY) [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

<i>bds</i>	The bd shape to be used to build the powerset.
<i>complexity</i>	This argument is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>bds</i> exceeds the maximum allowed space dimension.
--------------------------	--

template<typename PSET > template<typename Interval > Parma_Polyhedra_Library::Pointset_Powerset< PSET >::Pointset_Powerset (const Box< Interval > & box, Complexity_Class complexity = ANY_COMPLEXITY) [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

<i>box</i>	The box to be used to build the powerset.
<i>complexity</i>	This argument is ignored.

Exceptions

<i>std::length_error</i>	Thrown if the space dimension of <i>box</i> exceeds the maximum allowed space dimension.
--------------------------	--

10.79.3 Member Function Documentation

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::is_disjoint_from (const Pointset_Powerset< PSET > & y) const Returns `true` if and only if `*this` and `y` are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::constrains (Variable *var*) const Returns `true` if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

$$ps = \{\{x \geq 0\}, \{x \leq 0\}\},$$

even though *ps* is geometrically equal to the whole vector space.

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::bounds_from_above (const Linear_Expression & *expr*) const Returns `true` if and only if *expr* is bounded from above in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::bounds_from_below (const Linear_Expression & *expr*) const Returns `true` if and only if *expr* is bounded from below in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

Parameters

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	<code>true</code> if and only if the supremum is also the maximum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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 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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	true if and only if the supremum is also the maximum value;
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <i>expr</i> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

Parameters

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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
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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value;
<i>g</i>	When minimization succeeds, will be assigned a point or closure point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<code>std::invalid_argument</code>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------------	---

Warning

This may be *really* expensive!

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

<code>std::invalid_argument</code>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------------	---

Warning

This may be *really* expensive!

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::contains (const Pointset_Powerset< PSET > & y) const Returns `true` if and only if each disjunct of `y` is contained in a disjunct of `*this`.
Exceptions

<code>std::invalid_argument</code>	Thrown if <code>*this</code> and <code>y</code> are topology-incompatible or dimension-incompatible.
------------------------------------	--

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::strictly_contains (const Pointset_Powerset< PSET > & y) const Returns `true` if and only if each disjunct of `y` is strictly contained in a disjunct of `*this`.
Exceptions

<code>std::invalid_argument</code>	Thrown if <code>*this</code> and <code>y</code> are topology-incompatible or dimension-incompatible.
------------------------------------	--

template<typename PSET > Poly_Con_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Constraint & c) const Returns the relations holding between the powerset `*this` and the constraint `c`.
Exceptions

<code>std::invalid_argument</code>	Thrown if <code>*this</code> and constraint <code>c</code> are dimension-incompatible.
------------------------------------	--

template<typename PSET > Poly_Gen_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Generator & g) const Returns the relations holding between the powerset `*this` and the generator `g`.
Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible.
------------------------------	---

template<typename PSET > Poly_Con_Relation Parma_Polyhedra_Library::Pointset_Powerset< PSET >::relation_with (const Congruence & *cg*) const Returns the relations holding between the powerset **this* and the congruence *c*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>c</i> are dimension-incompatible.
------------------------------	--

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().

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_disjunct (const PSET & *ph*) Adds to **this* the disjunct *ph*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>ph</i> are dimension-incompatible.
------------------------------	--

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_constraint (const Constraint & *c*) Intersects **this* with constraint *c*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are topology-incompatible or dimension-incompatible.
------------------------------	---

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_constraint (const Constraint & *c*) Use the constraint *c* to refine **this*.

Parameters

<i>c</i>	The constraint to be used for refinement.
----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>c</i> are dimension-incompatible.
------------------------------	---

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_constraints (const Constraint_System & *cs*) Intersects **this* with the constraints in *cs*.

Parameters

<i>cs</i>	The constraints to intersect with.
-----------	------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	---

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_with_constraints (const Constraint_System & *cs*) Use the constraints in *cs* to refine **this*.

Parameters

<i>cs</i>	The constraints to be used for refinement.
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

**template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_
_congruence (const Congruence & *cg*)** Intersects **this* with congruence *cg*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

**template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_
_with_congruence (const Congruence & *cg*)** Use the congruence *cg* to refine **this*.

Parameters

<i>cg</i>	The congruence to be used for refinement.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cg</i> are dimension-incompatible.
------------------------------	--

**template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::add_
_congruences (const Congruence_System & *cgs*)** Intersects **this* with the congruences in *cgs*.

Parameters

<i>cgs</i>	The congruences to intersect with.
------------	------------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

**template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::refine_
_with_congruences (const Congruence_System & *cgs*)** Use the congruences in *cgs* to refine **this*.

Parameters

<i>cgs</i>	The congruences to be used for refinement.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

**template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::unconstrain
(Variable *var*)** Computes the [cylindrification](#) of **this* with respect to space dimension *var*, assigning the result to **this*.

Parameters

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::drop_-some_non_integer_points (Complexity_Class complexity = ANY_COMPLEXITY) Possibly tightens **this* by dropping some points with non-integer coordinates.

Parameters

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

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

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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*.

Parameters

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

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.

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename PSET > bool Parma_Polyhedra_Library::Pointset_Powerset< PSET >::simplify_using_context_assign (const Pointset_Powerset< PSET > & y) Assigns to **this* a [meet-preserving simplification](#) of **this* with respect to *y*. If *false* is returned, then the intersection is empty.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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 ()) Assigns to **this* the [affine image](#) of **this* under the function mapping variable *var* to the affine expression specified by *expr* and *denominator*.

Parameters

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the [affine preimage](#) of **this* under the function mapping variable *var* to the affine expression specified by *expr* and *denominator*.

Parameters

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the image of **this* with respect to the [generalized affine relation](#) $var' \bowtie \frac{expr}{denominator}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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 ()) Assigns to **this* the preimage of **this* with respect to the [generalized affine relation](#) $var' \bowtie \frac{expr}{denominator}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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](#) $lhs' \bowtie rhs$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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](#) $lhs' \bowtie rhs$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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 ()) Assigns to **this* the image of **this* with respect to the [bounded affine relation](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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 ()) Assigns to **this* the preimage of **this* with respect to the [bounded affine relation](#) $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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*.

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 * cs_p = 0, unsigned complexity_threshold = 16, bool wrap_individually = true) Wraps the specified dimensions of the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	Possibly null pointer to a constraint system whose variables are contained in <i>vars</i> . If <i>*cs_p</i> depends on variables not in <i>vars</i> , 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 <i>*cs_p</i> .
<i>complexity_threshold</i>	A precision parameter of the wrapping operator : higher values result in possibly improved precision.
<i>wrap_individually</i>	true if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*cs_p</i> is dimension-incompatible with <i>vars</i> , or if <i>*this</i> is dimension-incompatible <i>vars</i> or with <i>*cs_p</i> .
------------------------------	---

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::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.

On exit, for all the pairs \mathcal{P}, \mathcal{Q} of different disjuncts in **this*, we have $\mathcal{P} \uplus \mathcal{Q} \neq \mathcal{P} \cup \mathcal{Q}$.

template<typename PSET > template<typename Widening > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::BGP99_extrapolation_assign (const Pointset_Powerset< PSET > & y, Widening widen_fun, unsigned max_disjuncts) Assigns to **this* the result of applying the [BG-P99 extrapolation operator](#) to **this* and *y*, using the widening function *widen_fun* and the cardinality threshold *max_disjuncts*.

Parameters

<i>y</i>	A powerset that <i>must</i> definitely entail <i>*this</i> ;
<i>widen_fun</i>	The widening function to be used on polyhedra objects. It is obtained from the corresponding widening method by using the helper function <code>Parma_Polyhedra_Library::widen_fun_ref</code> . Legal values are, e.g., <code>widen_fun_ref(&Polyhedron::H79_widening_assign)</code> and <code>widen_fun_ref(&Polyhedron::limited_H79_extrapolation_assign, cs)</code> ;

<i>max_disjuncts</i>	The maximum number of disjuncts occurring in the powerset <i>*this</i> <i>before</i> starting the computation. If this number is exceeded, some of the disjuncts in <i>*this</i> are collapsed (i.e., joined together).
----------------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

For a description of the extrapolation operator, see [\[BGP99\]](#) and [\[BHZ03b\]](#).

template<typename PSET > template<typename Cert , typename Widening > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::BHZ03_widening_assign (const Pointset_Powerset< PSET > & y, Widening widen_fun) Assigns to **this* the result of computing the [BHZ03-widening](#) between **this* and *y*, using the widening function *widen_fun* certified by the convergence certificate *Cert*.

Parameters

<i>y</i>	The finite powerset computed in the previous iteration step. It <i>must</i> definitely entail <i>*this</i> ;
<i>widen_fun</i>	The widening function to be used on disjuncts. It is obtained from the corresponding widening method by using the helper function <i>widen_fun_ref</i> . Legal values are, e.g., <i>widen_fun_ref(&Polyhedron::H79_widening_assign)</i> and <i>widen_fun_ref(&Polyhedron::limited_H79_extrapolation_assign, cs)</i> .

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> 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 *widen_fun*; otherwise, an extrapolation operator is obtained. For a description of the methods that should be provided by *Cert*, see [BHR-Z03_Certificate](#) or [H79_Certificate](#).

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::concatenate_assign (const Pointset_Powerset< PSET > & y) 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*.

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::remove_space_dimensions (const Variables_Set & vars) Removes all the specified space dimensions.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be removed.
-------------	--

Exceptions

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

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::remove_higher_space_dimensions (dimension_type new_dimension) Removes the higher space dimensions so that the resulting space will have dimension *new_dimension*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>new_dimensions</code> is greater than the space dimension of <code>*this</code> .
------------------------------	---

template<typename PSET > template<typename Partial_Function > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a partial function.

See also [Polyhedron::map_space_dimensions](#).

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::expand_space_dimension (Variable var, dimension_type m) Creates `m` copies of the space dimension corresponding to `var`.

Parameters

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding <code>m</code> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

template<typename PSET > void Parma_Polyhedra_Library::Pointset_Powerset< PSET >::fold_space_dimensions (const Variables_Set & vars, Variable dest) Folds the space dimensions in `vars` into `dest`.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> or with one of the Variable objects contained in <code>vars</code> . Also thrown if <code>dest</code> is contained in <code>vars</code> .
------------------------------	---

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.

10.79.4 Friends And Related Function Documentation

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

<i>wm</i>	The widening method.
-----------	----------------------

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

<i>lwm</i>	The limited widening method.
<i>cs</i>	The constraint system limiting the widening.

template<typename PSET > Widening_Function< PSET > widen_fun_ref (void(PSET::*)(const PSET &, unsigned *) *wm*) [related]

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]

template<typename PSET > void swap (Pointset_Powerset< PSET > & *x*, Pointset_Powerset< PSET > & *y*) [related] Swaps *x* with *y*.

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*).

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* with respect to *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.second*.

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.

template<typename PSET > bool check_containment (const PSET & *ph*, const Pointset_Powerset< PSET > & *ps*) [related]

bool check_containment (const C.Polyhedron & *ph*, const Pointset.Powerset< C.Polyhedron > & *ps*) [**related**]

template<typename PSET > void swap (Pointset.Powerset< PSET > & *x*, Pointset.Powerset< PSET > & *y*) [**related**]

template<typename PSET > std::pair< PSET, Pointset.Powerset< NNC.Polyhedron > > linear_partition (const PSET & *p*, const PSET & *q*) [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.80 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.

Related Functions

(Note that these are not member functions.)

- bool [operator==](#) (const [Poly_Con_Relation](#) &*x*, const [Poly_Con_Relation](#) &*y*)
True if and only if `x` and `y` are logically equivalent.
- bool [operator!=](#) (const [Poly_Con_Relation](#) &*x*, const [Poly_Con_Relation](#) &*y*)
True if and only if `x` and `y` are not logically equivalent.

- `Poly_Con_Relation operator&&` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)
Yields the logical conjunction of x and y.
- `Poly_Con_Relation operator-` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)
Yields the assertion with all the conjuncts of x that are not in y.
- `std::ostream & operator<<` (std::ostream &s, const `Poly_Con_Relation` &r)
Output operator.
- `bool operator==` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)
- `bool operator!=` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)
- `Poly_Con_Relation operator&&` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)
- `Poly_Con_Relation operator-` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y)

10.80.1 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.

10.80.2 Friends And Related Function Documentation

`bool operator==` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**] True if and only if x and y are logically equivalent.

`bool operator!=` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**] 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) [**related**] Yields the logical conjunction of x and y.

`std::ostream & operator<<` (std::ostream &s, const `Poly_Con_Relation` &r) [**related**] Output operator.

`bool operator==` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**]

`bool operator!=` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**]

`Poly_Con_Relation operator&&` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**]

`Poly_Con_Relation operator-` (const `Poly_Con_Relation` &x, const `Poly_Con_Relation` &y) [**related**]
The documentation for this class was generated from the following file:

- `ppl.hh`

10.81 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.

Related Functions

(Note that these are not member functions.)

- bool `operator== (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
True if and only if `x` and `y` are logically equivalent.
- bool `operator!= (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
True if and only if `x` and `y` are not logically equivalent.
- `Poly_Gen_Relation operator&& (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
Yields the logical conjunction of `x` and `y`.
- `Poly_Gen_Relation operator- (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
Yields the assertion with all the conjuncts of `x` that are not in `y`.
- `std::ostream & operator<< (std::ostream &s, const Poly_Gen_Relation &r)`
Output operator.
- bool `operator== (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
- bool `operator!= (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
- `Poly_Gen_Relation operator&& (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`
- `Poly_Gen_Relation operator- (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y)`

10.81.1 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.

10.81.2 Friends And Related Function Documentation

bool operator== (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y) [related] True if and only if `x` and `y` are logically equivalent.

bool operator!= (const Poly_Gen_Relation &x, const Poly_Gen_Relation &y) [related] 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)
[related] Yields the logical conjunction of x and y .

std::ostream & operator<< (std::ostream & s, const Poly_Gen_Relation & r) **[related]** Output operator.

bool operator== (const Poly_Gen_Relation & x, const Poly_Gen_Relation & y) **[related]**

bool operator!= (const Poly_Gen_Relation & x, const Poly_Gen_Relation & y) **[related]**

Poly_Gen_Relation operator&& (const Poly_Gen_Relation & x, const Poly_Gen_Relation & y)
[related]

Poly_Gen_Relation operator- (const Poly_Gen_Relation & x, const Poly_Gen_Relation & y) **[related]**
The documentation for this class was generated from the following file:

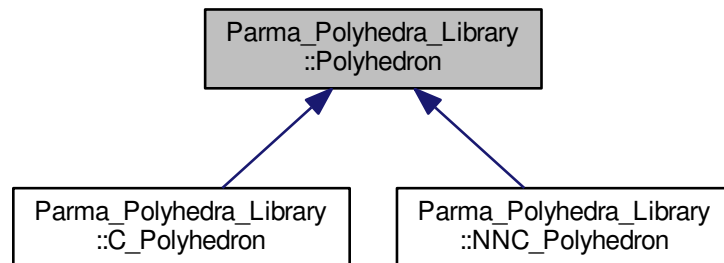
- ppl.hh

10.82 Parma_Polyhedra_Library::Polyhedron Class Reference

The base class for convex polyhedra.

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

Inheritance diagram for Parma_Polyhedra_Library::Polyhedron:



Public Types

- typedef [Coefficient](#) [coefficient_type](#)
The numeric type of coefficients.

Public Member Functions

- template<typename Source_Linear_System , typename Dest_Linear_System >
[dimension_type conversion](#) (Source_Linear_System &source, const [dimension_type](#) start, Dest_Linear_System &dest, Bit_Matrix &sat, [dimension_type](#) num_lines_or_inequalities)
- template<typename Source_Linear_System , typename Dest_Linear_System >
bool [minimize](#) (const bool con_to_gen, Source_Linear_System &source, Dest_Linear_System &dest, Bit_Matrix &sat)

- `template<typename Source_Linear_System1 , typename Source_Linear_System2 , typename Dest_Linear_System >`
`bool add_and_minimize (const bool con_to_gen, Source_Linear_System1 &source1, Dest_Linear_System &dest, Bit_Matrix &sat, const Source_Linear_System2 &source2)`
- `template<typename Source_Linear_System , typename Dest_Linear_System >`
`bool add_and_minimize (const bool con_to_gen, Source_Linear_System &source, Dest_Linear_System &dest, Bit_Matrix &sat)`
- `template<typename Linear_System1 >`
`dimension_type simplify (Linear_System1 &sys, Bit_Matrix &sat)`

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.*
- `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 &c) 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.*
- template<typename FP_Format , typename Interval_Info >
void `refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< FP_Format, Interval_Info > > &left, const `Linear_Form`< `Interval`< FP_Format, Interval_Info > > &right, bool is_strict=false)
*Refines *this with the constraint expressed by $left < right$ if is_strict is set, with the constraint $left \leq right$ otherwise.*
- template<typename FP_Format , typename Interval_Info >
void `generalized_refine_with_linear_form_inequality` (const `Linear_Form`< `Interval`< FP_Format, Interval_Info > > &left, const `Linear_Form`< `Interval`< FP_Format, Interval_Info > > &right, `Relation_Symbol` relsym)
*Refines *this with the constraint expressed by $left \bowtie right$, where \bowtie is the relation symbol specified by relsym.*
- template<typename FP_Format , typename Interval_Info >
void `refine_fp_interval_abstract_store` (`Box`< `Interval`< FP_Format, Interval_Info > > &store)
const
*Refines store with the 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 `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.*
- template<typename FP_Format , typename Interval_Info >
void `affine_form_image` (`Variable` var, const `Linear_Form`< `Interval`< FP_Format, Interval_Info > > &lf)
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 $var' \bowtie \frac{expr}{denominator}$, 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* $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relysym*.
- void **generalized_affine_image** (const **Linear_Expression** &lhs, **Relation_Symbol** relysym, const **Linear_Expression** &rhs)
Assigns to **this* the image of **this* with respect to the *generalized affine relation* $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relysym*.
 - void **generalized_affine_preimage** (const **Linear_Expression** &lhs, **Relation_Symbol** relysym, const **Linear_Expression** &rhs)
Assigns to **this* the preimage of **this* with respect to the *generalized affine relation* $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relysym*.
 - 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* $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - 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* $\text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.
 - void **time_elapse_assign** (const **Polyhedron** &y)
Assigns to **this* the result of computing the *time-elapse* between **this* and *y*.
 - void **positive_time_elapse_assign** (const **Polyhedron** &y)
Assigns to **this* (the best approximation of) the result of computing the *positive 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** *cs, p=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 `m_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** **vars_p*, **Complexity_Class** *complexity*)
*Possibly tightens *this by dropping some points with non-integer coordinates for the space dimensions corresponding to *vars_p.*
- template<typename *FP_Format* , typename *Interval_Info* >
void **overapproximate_linear_form** (const **Linear_Form**< *Interval*< *FP_Format*, *Interval_Info* > > &*lf*, const **dimension_type** *lf_dimension*, **Linear_Form**< *Interval*< *FP_Format*, *Interval_Info* > > &*result*)
Helper function that overapproximates an interval linear form.
- void **positive_time_elapse_assign_impl** (const **Polyhedron** &*y*)
*Assuming *this is NNC, assigns to *this the result of the "positive time-elapse" between *this and y.*

Static Protected Member Functions

- template<typename *FP_Format* , typename *Interval_Info* >
static void **convert_to_integer_expression** (const **Linear_Form**< *Interval*< *FP_Format*, *Interval_Info* > > &*lf*, const **dimension_type** *lf_dimension*, **Linear_Expression** &*result*)
Helper function that makes result become a Linear_Expression obtained by normalizing the denominators in lf.
- template<typename *FP_Format* , typename *Interval_Info* >
static void **convert_to_integer_expressions** (const **Linear_Form**< *Interval*< *FP_Format*, *Interval_Info* > > &*lf*, const **dimension_type** *lf_dimension*, **Linear_Expression** &*res*, **Coefficient** &*res_low_coeff*, **Coefficient** &*res_hi_coeff*, **Coefficient** &*denominator*)
Normalization helper function.

Related Functions

(Note that these are not member functions.)

- **std::ostream** & **operator<<** (**std::ostream** &*s*, const **Polyhedron** &*ph*)
Output operator.
- void **swap** (**Polyhedron** &*x*, **Polyhedron** &*y*)

Swaps x with y .

- `bool operator==(const Polyhedron &x, const Polyhedron &y)`

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

- `bool operator!=(const Polyhedron &x, const Polyhedron &y)`

Returns `true` if and only if x and y are different polyhedra.

- `void swap(Polyhedron &x, Polyhedron &y)`
- `bool operator!=(const Polyhedron &x, const Polyhedron &y)`

10.82.1 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

$$\{(2, y)^T \in \mathbb{R}^2 \mid y \in \mathbb{R}\}.$$

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 $\{(2, 0)^T\} \subseteq \mathbb{R}^2$.

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 $\{(3, 1, 0, 2)^T\} \subseteq \mathbb{R}^4$, while the resulting polyhedron is $\{(3, 2)^T\} \subseteq \mathbb{R}^2$. Be careful when removing space dimensions *incrementally*: since dimensions are automatically renamed after each application of the `remove_space_dimensions` operator, unexpected results can be obtained. For instance, by using the following code we would obtain a different result:

```

set<Variable> 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 $\{(3, 0)^T\} \subseteq \mathbb{R}^2$: 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.

10.82.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Polyhedron::Polyhedron (*Topology* *topol*, *dimension_type* *num_dimensions*, *Degenerate_Element* *kind*) [**protected**] Builds a polyhedron having the specified properties.

Parameters

<i>topol</i>	The topology of the polyhedron;
<i>num_-dimensions</i>	The number of dimensions of the vector space enclosing the polyhedron;
<i>kind</i>	Specifies whether the universe or the empty polyhedron has to be built.

Parma_Polyhedra_Library::Polyhedron::Polyhedron (*const* Polyhedron & *y*, *Complexity_Class* *complexity* = ANY_COMPLEXITY) [**protected**] Ordinary copy constructor.

The complexity argument is ignored.

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

<i>topol</i>	The topology of the polyhedron;
<i>cs</i>	The system of constraints defining the polyhedron.

Exceptions

<i>std::invalid_argument</i>	Thrown if the topology of <i>cs</i> is incompatible with <i>topol</i> .
------------------------------	---

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

<i>topol</i>	The topology of the polyhedron;
<i>cs</i>	The system of constraints defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the topology of <code>cs</code> is incompatible with <code>topol</code> .
------------------------------	---

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

<i>topol</i>	The topology of the polyhedron;
<i>gs</i>	The system of generators defining the polyhedron.

Exceptions

<i>std::invalid_argument</i>	Thrown if the topology of <code>gs</code> is incompatible with <code>topol</code> , or if the system of generators is not empty but has no points.
------------------------------	--

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

<i>topol</i>	The topology of the polyhedron;
<i>gs</i>	The system of generators defining the polyhedron. It is not declared <code>const</code> because its data-structures may be recycled to build the polyhedron.
<i>dummy</i>	A dummy tag to syntactically differentiate this one from the other constructors.

Exceptions

<i>std::invalid_argument</i>	Thrown if the topology of <code>gs</code> is incompatible with <code>topol</code> , or if the system of generators is not empty but has no points.
------------------------------	--

template<typename Interval > Parma_Polyhedra_Library::Polyhedron::Polyhedron (Topology *topol*, const Box< Interval > & *box*, Complexity_Class *complexity* = ANY_COMPLEXITY) [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

<i>topol</i>	The topology of the polyhedron;
<i>box</i>	The box representing the polyhedron to be built;
<i>complexity</i>	This argument is ignored.

10.82.3 Member Function Documentation

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>c</i> are dimension-incompatible.
------------------------------	--

bool Parma_Polyhedra_Library::Polyhedron::is_disjoint_from (const Polyhedron & y) const Returns *true* if and only if **this* and *y* are disjoint.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>x</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

bool Parma_Polyhedra_Library::Polyhedron::constrains (Variable var) const Returns *true* if and only if *var* is constrained in **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	true if and only if the supremum is also the maximum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be maximized subject to <i>*this</i> ;
<i>sup_n</i>	The numerator of the supremum value;
<i>sup_d</i>	The denominator of the supremum value;
<i>maximum</i>	true if and only if the supremum is also the maximum value;
<i>g</i>	When maximization succeeds, will be assigned the point or closure point where <i>expr</i> reaches its supremum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression to be minimized subject to <i>*this</i> ;
<i>inf_n</i>	The numerator of the infimum value;
<i>inf_d</i>	The denominator of the infimum value;
<i>minimum</i>	true if and only if the infimum is also the minimum value;
<i>g</i>	When minimization succeeds, will be assigned a point or closure point where <i>expr</i> reaches its infimum value.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> 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.

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

<i>expr</i>	The linear expression for which the frequency is needed;
<i>freq_n</i>	If <i>true</i> is returned, the value is set to 0; Present for interface compatibility with class Grid , where the frequency can have a non-zero value;
<i>freq_d</i>	If <i>true</i> is returned, the value is set to 1;
<i>val_n</i>	The numerator of <i>val</i> ;
<i>val_d</i>	The denominator of <i>val</i> ;

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>expr</i> and <i>*this</i> are dimension-incompatible.
------------------------------	--

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

bool Parma_Polyhedra_Library::Polyhedron::contains (const Polyhedron & *y*) const Returns *true* if and only if **this* contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

bool Parma_Polyhedra_Library::Polyhedron::strictly_contains (const Polyhedron & *y*) const [inline] Returns *true* if and only if **this* strictly contains *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>check_not_empty</i>	true if and only if, in addition to checking the invariants, <i>*this</i> 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.

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

<i>c</i>	The constraint that will be added to the system of constraints of <i>*this</i> .
----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are topology-incompatible or dimension-incompatible.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and generator <i>g</i> are topology-incompatible or dimension-incompatible, or if <i>*this</i> is an empty polyhedron and <i>g</i> is not a point.
------------------------------	---

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible, or if <i>cg</i> is a proper congruence which is neither a tautology, nor a contradiction.
------------------------------	---

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

<i>cs</i>	Contains the constraints that will be added to the system of constraints of <i>*this</i> .
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	---

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

<i>cs</i>	The constraint system to be added to <i>*this</i> . The constraints in <i>cs</i> may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> 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.

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

<i>gs</i>	Contains the generators that will be added to the system of generators of <i>*this</i> .
-----------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>gs</i> are topology-incompatible or dimension-incompatible, or if <i>*this</i> is empty and the system of generators <i>gs</i> is not empty, but has no points.
------------------------------	---

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

<i>gs</i>	The generator system to be added to <i>*this</i> . The generators in <i>gs</i> may be recycled.
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>gs</i> are topology-incompatible or dimension-incompatible, or if <i>*this</i> is empty and the system of generators <i>gs</i> 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.

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

<i>cgs</i>	The congruences to be added.
------------	------------------------------

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or if there exists in <i>cgs</i> a proper congruence which is neither a tautology, nor a contradiction.
------------------------------	---

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

<i>cgs</i>	The congruences to be added. Its elements may be recycled.
------------	--

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible, or if there exists in <i>cgs</i> 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.

void Parma_Polyhedra_Library::Polyhedron::refine_with_constraint (const Constraint & c) Uses a copy of constraint *c* to refine **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and constraint <i>c</i> are dimension-incompatible.
------------------------------	--

void Parma_Polyhedra_Library::Polyhedron::refine_with_congruence (const Congruence & cg) Uses a copy of congruence *cg* to refine **this*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and congruence <i>cg</i> are dimension-incompatible.
------------------------------	---

void Parma_Polyhedra_Library::Polyhedron::refine_with_constraints (const Constraint_System & cs) Uses a copy of the constraints in *cs* to refine **this*.

Parameters

<i>cs</i>	Contains the constraints used to refine the system of constraints of <i>*this</i> .
-----------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cs</i> are dimension-incompatible.
------------------------------	--

void Parma_Polyhedra_Library::Polyhedron::refine_with_congruences (const Congruence_System & cgs) Uses a copy of the congruences in *cgs* to refine **this*.

Parameters

<i>cgs</i>	Contains the congruences used to refine the system of constraints of <i>*this</i> .
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>cgs</i> are dimension-incompatible.
------------------------------	---

template<typename FP_Format, typename Interval_Info > void Parma_Polyhedra_Library::Polyhedron::refine_with_linear_form_inequality (const Linear_Form< Interval< FP_Format, Interval_Info > > & left, const Linear_Form< Interval< FP_Format, Interval_Info > > & right, bool is_strict = false) Refines **this* with the constraint expressed by *left* < *right* if *is_strict* is set, with the constraint $\text{left} \leq \text{right}$ otherwise.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is on the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is on the right of the comparison operator. All of its coefficients MUST be bounded.
<i>is_strict</i>	True if the comparison is strict.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>left</i> (or <i>right</i>) is dimension-incompatible with <i>*this</i> .
------------------------------	--

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by *left* and *right* respectively.

template<typename FP_Format, typename Interval_Info > void Parma_Polyhedra_Library::Polyhedron::generalized_refine_with_linear_form_inequality (const Linear_Form< Interval< FP_Format, Interval_Info > > & *left*, const Linear_Form< Interval< FP_Format, Interval_Info > > & *right*, Relation_Symbol *relysym*) [inline] Refines **this* with the constraint expressed by *left* \bowtie *right*, where \bowtie is the relation symbol specified by *relysym*.

Parameters

<i>left</i>	The linear form on intervals with floating point boundaries that is on the left of the comparison operator. All of its coefficients MUST be bounded.
<i>right</i>	The linear form on intervals with floating point boundaries that is on the right of the comparison operator. All of its coefficients MUST be bounded.
<i>relysym</i>	The relation symbol.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>left</i> (or <i>right</i>) is dimension-incompatible with <i>*this</i> .
<i>std::runtime_error</i>	Thrown if <i>relysym</i> is not a valid relation symbol.

This function is used in abstract interpretation to model a filter that is generated by a comparison of two expressions that are correctly approximated by *left* and *right* respectively.

template<typename FP_Format, typename Interval_Info > void Parma_Polyhedra_Library::Polyhedron::refine_fp_interval_abstract_store (Box< Interval< FP_Format, Interval_Info > > & *store*) const [inline] Refines *store* with the constraints defining **this*.

Parameters

<i>store</i>	The interval floating point abstract store to refine.
--------------	---

void Parma_Polyhedra_Library::Polyhedron::unconstrain (Variable *var*) Computes the [cylindrifcation](#) of **this* with respect to space dimension *var*, assigning the result to **this*.

Parameters

<i>var</i>	The space dimension that will be unconstrained.
------------	---

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	---

void Parma_Polyhedra_Library::Polyhedron::unconstrain (const Variables_Set & *vars*) Computes the [cylindrifcation](#) of **this* with respect to the set of space dimensions *vars*, assigning the result to **this*.

Parameters

<i>vars</i>	The set of space dimension that will be unconstrained.
-------------	--

Exceptions

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

void Parma_Polyhedra_Library::Polyhedron::intersection_assign (const Polyhedron & y) Assigns to **this* the intersection of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

void Parma_Polyhedra_Library::Polyhedron::poly_hull_assign (const Polyhedron & y) Assigns to **this* the poly-hull of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

void Parma_Polyhedra_Library::Polyhedron::poly_difference_assign (const Polyhedron & y) Assigns to **this* the [poly-difference](#) of **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>var</i>	The variable to which the affine expression is assigned;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

template<typename FP_Format , typename Interval_Info > void Parma_Polyhedra_Library::Polyhedron::affine_form_image (Variable *var*, const Linear_Form< Interval< FP_Format, Interval_Info > > & *lf*) Assigns to `*this` the [affine form image](#) of `*this` under the function mapping variable `var` into the affine expression(s) specified by `lf`.

Parameters

<i>var</i>	The variable to which the affine expression is assigned.
<i>lf</i>	The linear form on intervals with floating point boundaries that defines the affine expression(s). ALL of its coefficients MUST be bounded.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>lf</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

This function is used in abstract interpretation to model an assignment of a value that is correctly overapproximated by `lf` to the floating point variable represented by `var`.

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

<i>var</i>	The variable to which the affine expression is substituted;
<i>expr</i>	The numerator of the affine expression;
<i>denominator</i>	The denominator of the affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> .
------------------------------	---

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](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by `relsym`.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if denominator is zero or if <code>expr</code> and <code>*this</code> are dimension-incompatible or if <code>var</code> is not a space dimension of <code>*this</code> or if <code>*this</code> is a C_Polyhedron and <code>relsym</code> is a strict relation symbol.
------------------------------	---

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](#) $\text{var}' \bowtie \frac{\text{expr}}{\text{denominator}}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>var</i>	The left hand side variable of the generalized affine relation;
<i>relsym</i>	The relation symbol;
<i>expr</i>	The numerator of the right hand side affine expression;
<i>denominator</i>	The denominator of the right hand side affine expression (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>expr</i> and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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](#) $\text{lhs}' \bowtie \text{rhs}$, where \bowtie is the relation symbol encoded by *relsym*.

Parameters

<i>lhs</i>	The left hand side affine expression;
<i>relsym</i>	The relation symbol;
<i>rhs</i>	The right hand side affine expression.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> is dimension-incompatible with <i>lhs</i> or <i>rhs</i> or if <i>*this</i> is a C.Polyhedron and <i>relsym</i> is a strict relation symbol.
------------------------------	--

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](#) $\frac{\text{lb_expr}}{\text{denominator}} \leq \text{var}' \leq \frac{\text{ub_expr}}{\text{denominator}}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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** $\frac{lb_expr}{denominator} \leq var' \leq \frac{ub_expr}{denominator}$.

Parameters

<i>var</i>	The variable updated by the affine relation;
<i>lb_expr</i>	The numerator of the lower bounding affine expression;
<i>ub_expr</i>	The numerator of the upper bounding affine expression;
<i>denominator</i>	The (common) denominator for the lower and upper bounding affine expressions (optional argument with default value 1).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>denominator</i> is zero or if <i>lb_expr</i> (resp., <i>ub_expr</i>) and <i>*this</i> are dimension-incompatible or if <i>var</i> is not a space dimension of <i>*this</i> .
------------------------------	--

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

void Parma_Polyhedra_Library::Polyhedron::positive_time_elapse_assign (const Polyhedron & *y*) Assigns to **this* (the best approximation of) the result of computing the **positive time-elapse** between **this* and *y*.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

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 * *cs*, *p* = 0, unsigned *complexity_threshold* = 16, bool *wrap_individually* = true) Wraps the specified dimensions of the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be wrapped.
<i>w</i>	The width of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>r</i>	The representation of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>o</i>	The overflow behavior of the bounded integer type corresponding to all the dimensions to be wrapped.
<i>cs_p</i>	Possibly null pointer to a constraint system whose variables are contained in <i>vars</i> . If <i>*cs_p</i> depends on variables not in <i>vars</i> , 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 <i>*cs_p</i> .
<i>complexity_-threshold</i>	A precision parameter of the wrapping operator : higher values result in possibly improved precision.
<i>wrap_-individually</i>	<code>true</code> if the dimensions should be wrapped individually (something that results in much greater efficiency to the detriment of precision).

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*cs_p</i> is dimension-incompatible with <i>vars</i> , or if <i>*this</i> is dimension-incompatible <i>vars</i> or with <i>*cs_p</i> .
------------------------------	---

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

<i>complexity</i>	The maximal complexity of any algorithms used.
-------------------	--

Note

Currently there is no optimality guarantee, not even if *complexity* is [ANY_COMPLEXITY](#).

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

<i>vars</i>	Points with non-integer coordinates for these variables/space-dimensions can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

Currently there is no optimality guarantee, not even if *complexity* is [ANY_COMPLEXITY](#).

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>cs</i>	The system of constraints used to improve the widened polyhedron;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>cs</i>	The system of constraints used to improve the widened polyhedron;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>cs</i>	The system of constraints used to improve the widened polyhedron;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>y</i>	A polyhedron that <i>must</i> be contained in <i>*this</i> ;
<i>cs</i>	The system of constraints used to improve the widened polyhedron;
<i>tp</i>	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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> , <i>y</i> and <i>cs</i> are topology-incompatible or dimension-incompatible.
------------------------------	--

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

<i>m</i>	The number of dimensions to add.
----------	----------------------------------

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> 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

$$\{ (x, y, z)^T \in \mathbb{R}^3 \mid (x, y)^T \in \mathcal{P} \}.$$

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

<i>m</i>	The number of space dimensions to add.
----------	--

Exceptions

<i>std::length_error</i>	Thrown if adding <i>m</i> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .
--------------------------	--

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

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

void Parma_Polyhedra_Library::Polyhedron::concatenate_assign (const Polyhedron & y) Assigns to **this* the [concatenation](#) of **this* and *y*, taken in this order.

Exceptions

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are topology-incompatible.
<i>std::length_error</i>	Thrown if the concatenation would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

void Parma_Polyhedra_Library::Polyhedron::remove_space_dimensions (const Variables_Set & vars) Removes all the specified dimensions from the vector space.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be removed.
-------------	--

Exceptions

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

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

<i>std::invalid_argument</i>	Thrown if <i>new_dimensions</i> is greater than the space dimension of <i>*this</i> .
------------------------------	---

template<typename Partial_Function > void Parma_Polyhedra_Library::Polyhedron::map_space_dimensions (const Partial_Function & pfunc) Remaps the dimensions of the vector space according to a [partial function](#).

Parameters

<i>pfunc</i>	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](#).

void Parma_Polyhedra_Library::Polyhedron::expand_space_dimension (Variable var, dimension_type m) Creates `m` copies of the space dimension corresponding to `var`.

Parameters

<i>var</i>	The variable corresponding to the space dimension to be replicated;
<i>m</i>	The number of replicas to be created.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>var</code> does not correspond to a dimension of the vector space.
<i>std::length_error</i>	Thrown if adding <code>m</code> new space dimensions would cause the vector space to exceed dimension <code>max_space_dimension()</code> .

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

void Parma_Polyhedra_Library::Polyhedron::fold_space_dimensions (const Variables_Set & vars, Variable dest) Folds the space dimensions in `vars` into `dest`.

Parameters

<i>vars</i>	The set of Variable objects corresponding to the space dimensions to be folded;
<i>dest</i>	The variable corresponding to the space dimension that is the destination of the folding operation.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> is dimension-incompatible with <code>dest</code> or with one of the Variable objects contained in <code>vars</code> . Also thrown if <code>dest</code> is contained in <code>vars</code> .
------------------------------	---

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.

void Parma_Polyhedra_Library::Polyhedron::m_swap (Polyhedron & y) [inline] Swaps `*this` with polyhedron `y`. (`*this` and `y` can be dimension-incompatible.)

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>x</code> and <code>y</code> are topology-incompatible.
------------------------------	--

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()`.

**void Parma_Polyhedra_Library::Polyhedron::drop_some_non_integer_points (const Variables_Set *
vars_p, Complexity_Class *complexity*) [protected]** Possibly tightens **this* by dropping some
points with non-integer coordinates for the space dimensions corresponding to **vars_p*.

Parameters

<i>vars_p</i>	When nonzero, points with non-integer coordinates for the variables/space-dimensions contained in <i>*vars_p</i> can be discarded.
<i>complexity</i>	The maximal complexity of any algorithms used.

Note

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

template<typename FP_Format , typename Interval_Info > void Parma_Polyhedra_Library::Polyhedron::overapproximate_linear_form (const Linear_Form< Interval< FP_Format, Interval_Info > > & lf, const dimension_type lf_dimension, Linear_Form< Interval< FP_Format, Interval_Info > > & result) [protected] Helper function that overapproximates an interval linear form.

Parameters

<i>lf</i>	The linear form on intervals with floating point boundaries to approximate. ALL of its coefficients MUST be bounded.
<i>lf_dimension</i>	Must be the space dimension of <i>lf</i> .
<i>result</i>	Used to store the result.

This function makes `result` become a linear form that is a correct approximation of `lf` under the constraints specified by `*this`. The resulting linear form has the property that all of its variable coefficients have a non-significant upper bound and can thus be considered as singletons.

template<typename FP_Format , typename Interval_Info > static void Parma_Polyhedra_Library::Polyhedron::convert_to_integer_expression (const Linear_Form< Interval< FP_Format, Interval_Info > > & lf, const dimension_type lf_dimension, Linear_Expression & result) [static], [protected] Helper function that makes `result` become a [Linear_Expression](#) obtained by normalizing the denominators in `lf`.

Parameters

<i>lf</i>	The linear form on intervals with floating point boundaries to normalize. It should be the result of an application of static method <code>overapproximate_linear_form</code> .
<i>lf_dimension</i>	Must be the space dimension of <i>lf</i> .
<i>result</i>	Used to store the result.

This function ignores the upper bound of intervals in `lf`, so that in fact `result` can be seen as `lf` multiplied by a proper normalization constant.

template<typename FP_Format , typename Interval_Info > static void Parma_Polyhedra_Library::Polyhedron::convert_to_integer_expressions (const Linear_Form< Interval< FP_Format, Interval_Info > > & lf, const dimension_type lf_dimension, Linear_Expression & res, Coefficient & res_low_coeff, Coefficient & res_hi_coeff, Coefficient & denominator) [static], [protected] Normalization helper function.

Parameters

<i>lf</i>	The linear form on intervals with floating point boundaries to normalize. It should be the result of an application of static method <code>overapproximate_linear_form</code> .
<i>lf_dimension</i>	Must be the space dimension of <i>lf</i> .
<i>res</i>	Stores the normalized linear form, except its inhomogeneous term.
<i>res_low_coeff</i>	Stores the lower boundary of the inhomogeneous term of the result.
<i>res_hi_coeff</i>	Stores the higher boundary of the inhomogeneous term of the result.
<i>denominator</i>	Becomes the common denominator of <code>res_low_coeff</code> , <code>res_hi_coeff</code> and all coefficients in <code>res</code> .

Results are obtained by normalizing denominators in `lf`, ignoring the upper bounds of variable coefficients in `lf`.

void Parma_Polyhedra_Library::Polyhedron::positive_time_elapse_assign_impl (const Polyhedron & y) [protected] Assuming `*this` is NNC, assigns to `*this` the result of the "positive time-elapse" between `*this` and `y`.

Exceptions

<i>std::invalid_argument</i>	Thrown if <code>*this</code> and <code>y</code> are dimension-incompatible.
------------------------------	---

template<typename Source_Linear_System , typename Dest_Linear_System > dimension_type Parma_Polyhedra_Library::Polyhedron::conversion (Source_Linear_System & source, const dimension_type start, Dest_Linear_System & dest, Bit_Matrix & sat, dimension_type num_lines_or_equalities)

Returns

The number of lines of the polyhedron or the number of equality constraints in the result of conversion.

Parameters

<i>source</i>	The system to use to convert <code>dest</code> : it may be modified;
<i>start</i>	The index of <code>source</code> row from which conversion begin;
<i>dest</i>	The result of the conversion;
<i>sat</i>	The saturation matrix telling us, for each row in <code>source</code> , which are the rows of <code>dest</code> that satisfy but do not saturate it;
<i>num_lines_or_equalities</i>	The number of rows in the system <code>dest</code> that are either lines of the polyhedron (when <code>dest</code> is a system of generators) or equality constraints (when <code>dest</code> is a system of constraints).

template<typename Source_Linear_System , typename Dest_Linear_System > bool Parma_Polyhedra_Library::Polyhedron::minimize (const bool con_to_gen, Source_Linear_System & source, Dest_Linear_System & dest, Bit_Matrix & sat)

Returns

`true` if the polyhedron is empty, `false` otherwise.

Parameters

<i>con_to_gen</i>	<code>true</code> if <code>source</code> represents the constraints, <code>false</code> otherwise;
<i>source</i>	The given system, which is not empty;
<i>dest</i>	The system to build and minimize;
<i>sat</i>	The saturation matrix.

`dest` is not `const` because it will be built (and then modified) during [minimize\(\)](#). Also, `sat` and `source` are not `const` because the former will be built during `dest` creation and the latter will maybe be sorted and modified by `conversion()` and `simplify()`.

`sat` has the generators on its columns and the constraints on its rows if `con_to_gen` is `true`, otherwise it has the generators on its rows and the constraints on its columns.

Given `source`, this function builds (by means of `conversion()`) `dest` and then simplifies (invoking `simplify()`) `source`, erasing redundant rows. For the sequel we assume that `source` is the system of constraints and `dest` is the system of generators. This will simplify the description of the function; the dual case is similar.

```
template<typename Source_Linear_System1 , typename Source_Linear_System2 , typename Dest_Linear_System > bool Parma_Polyhedra_Library::Polyhedron::add_and_minimize ( const bool con_to_gen, Source_Linear_System1 & source1, Dest_Linear_System & dest, Bit_Matrix & sat, const Source_Linear_System2 & source2 )
```

Returns

true if the obtained polyhedron is empty, false otherwise.

Parameters

<i>con_to_gen</i>	true if <i>source1</i> and <i>source2</i> are system of constraints, false otherwise;
<i>source1</i>	The first element of the given DD pair;
<i>dest</i>	The second element of the given DD pair;
<i>sat</i>	The saturation matrix that bind <i>source1</i> to <i>dest</i> ;
<i>source2</i>	The new system of generators or constraints.

It is assumed that *source1* and *source2* are sorted and have no pending rows. It is also assumed that *dest* has no pending rows. On entry, the rows of *sat* are indexed by the rows of *dest* and its columns are indexed by the rows of *source1*. On exit, the rows of *sat* are indexed by the rows of *dest* and its columns are indexed by the rows of the system obtained by merging *source1* and *source2*.

Let us suppose we want to add some constraints to a given system of constraints *source1*. This method, given a minimized double description pair (*source1*, *dest*) and a system of new constraints *source2*, modifies *source1* by adding to it the constraints of *source2* that are not in *source1*. Then, by invoking `add_and_minimize(bool, Linear_System_Class&, Linear_System_Class&, Bit_Matrix&)`, processes the added constraints obtaining a new DD pair.

This method treats also the dual case, i.e., adding new generators to a previous system of generators. In this case *source1* contains the old generators, *source2* the new ones and *dest* is the system of constraints in the given minimized DD pair.

Since *source2* contains the constraints (or the generators) that will be added to *source1*, it is constant: it will not be modified.

```
template<typename Source_Linear_System , typename Dest_Linear_System > bool Parma_Polyhedra_Library::Polyhedron::add_and_minimize ( const bool con_to_gen, Source_Linear_System & source, Dest_Linear_System & dest, Bit_Matrix & sat )
```

Returns

true if the obtained polyhedron is empty, false otherwise.

Parameters

<i>con_to_gen</i>	true if <i>source</i> is a system of constraints, false otherwise;
<i>source</i>	The first element of the given DD pair. It also contains the pending rows to be processed;
<i>dest</i>	The second element of the given DD pair. It cannot have pending rows;
<i>sat</i>	The saturation matrix that bind the upper part of <i>source</i> to <i>dest</i> .

On entry, the rows of *sat* are indexed by the rows of *dest* and its columns are indexed by the non-pending rows of *source*. On exit, the rows of *sat* are indexed by the rows of *dest* and its columns are indexed by the rows of *source*.

Let us suppose that *source* is a system of constraints. This method assumes that the non-pending part of *source* and system *dest* form a double description pair in minimal form and will build a new DD pair in minimal form by processing the pending constraints in *source*. To this end, it will call `conversion()` and `simplify`.

This method treats also the dual case, i.e., processing pending generators. In this case *source* contains generators and *dest* is the system of constraints corresponding to the non-pending part of *source*.

template<typename Linear_System1 > dimension_type Parma_Polyhedra_Library::Polyhedron::simplify
(Linear_System1 & sys, Bit_Matrix & sat)

Returns

The rank of `sys`.

Parameters

<code>sys</code>	The system to simplify: it will be modified;
<code>sat</code>	The saturation matrix corresponding to <code>sys</code> .

`sys` may be modified by swapping some of its rows and by possibly removing some of them, if they turn out to be redundant.

If `sys` is a system of constraints, then the rows of `sat` are indexed by constraints and its columns are indexed by generators; otherwise, if `sys` is a system of generators, then the rows of `sat` are indexed by generators and its columns by constraints.

Given a system of constraints or a system of generators, this function simplifies it using Gauss' elimination method (to remove redundant equalities/lines), deleting redundant inequalities/rays/points and making back-substitution. The explanation that follows assumes that `sys` is a system of constraints. For the case when `sys` is a system of generators, a similar explanation can be obtain by applying duality.

The explanation relies on the notion of *redundancy*. (See the Introduction.)

First we make some observations that can help the reader in understanding the function:

Proposition: An inequality that is saturated by all the generators can be transformed to an equality.

In fact, by combining any number of generators that saturate the constraints, we obtain a generator that saturates the constraints too:

$$\langle c, r_1 \rangle = 0 \wedge \langle c, r_2 \rangle = 0 \Rightarrow \langle c, (\lambda_1 r_1 + \lambda_2 r_2) \rangle = \lambda_1 \langle c, r_1 \rangle + \lambda_2 \langle c, r_2 \rangle = 0,$$

where λ_1, λ_2 can be any real number.

10.82.4 Friends And Related Function Documentation

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 ", ".

void swap (Polyhedron & x, Polyhedron & y) [related] Swaps `x` with `y`.

bool operator== (const Polyhedron & x, const Polyhedron & y) [related] Returns `true` if and only if `x` and `y` are the same polyhedron.

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

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.

void swap (Polyhedron & x, Polyhedron & y) [related]

bool operator!= (const Polyhedron & x, const Polyhedron & y) [related] The documentation for this class was generated from the following file:

- `ppl.hh`

10.83 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 `m_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 >
void [swap](#) (Powerset< D > &x, Powerset< D > &y)
Swaps x with y.
- 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 >
bool [operator!=](#) (const Powerset< D > &x, const Powerset< D > &y)
- template<typename D >
void [swap](#) (Powerset< D > &x, Powerset< D > &y)

- `template<typename D >`
`bool operator== (const Powerset< D > &x, const Powerset< D > &y)`
- `template<typename D >`
`std::ostream & operator<< (std::ostream &s, const Powerset< D > &x)`

10.83.1 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.

10.83.2 Member Typedef Documentation

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.

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.

10.83.3 Member Function Documentation

template<typename D> void Parma_Polyhedra_Library::Powerset< D >::omega_reduce () const Drops from the sequence of disjuncts in **this* all the non-maximal elements so that **this* is non-redundant.

This method is declared `const` because, even though Omega-reduction may change the syntactic representation of **this*, its semantics will be unchanged.

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*.

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

<i>std::invalid_argument</i>	Thrown if <i>*this</i> and <i>y</i> are dimension-incompatible.
------------------------------	---

template<typename D> Powerset< D >::iterator Parma_Polyhedra_Library::Powerset< D >::add_non_bottom_disjunct_preserve_reduction (const D & d, iterator first, iterator last) [protected] Adds to **this* the disjunct *d*, assuming *d* is not the bottom element and ensuring partial Omega-reduction.

If *d* is not the bottom element and is not Omega-redundant with respect to elements in positions between *first* and *last*, all elements in these positions that would be made Omega-redundant by the addition of *d* are dropped and *d* is added to the reduced sequence. If **this* is reduced before an invocation of this method, it will be reduced upon successful return from the method.

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.

template<typename D> template<typename Binary.Operator.Assign> void Parma_Polyhedra_Library::Powerset< D >::pairwise_apply_assign (const Powerset< D > & y, Binary.Operator.Assign op_assign) [protected] Assigns to **this* the result of applying *op_assign* pairwise to the elements in **this* and *y*.

The elements of the powerset result are obtained by applying *op_assign* to each pair of elements whose components are drawn from **this* and *y*, respectively.

10.83.4 Friends And Related Function Documentation

template<typename D > void swap (Powerset< D > & x, Powerset< D > & y) [related]
Swaps x with y.

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.

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.

template<typename D > std::ostream & operator<< (std::ostream & s, const Powerset< D > & x) [related] Output operator.

template<typename D > bool operator!= (const Powerset< D > & x, const Powerset< D > & y) [related]

template<typename D > void swap (Powerset< D > & x, Powerset< D > & y) [related]

template<typename D > bool operator== (const Powerset< D > & x, const Powerset< D > & y) [related]

template<typename D > std::ostream & operator<< (std::ostream & s, const Powerset< D > & x) [related] The documentation for this class was generated from the following file:

- ppl.hh

10.84 Parma Polyhedra Library::Recycle Input Struct Reference

A tag class.

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

10.84.1 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

10.85 Parma Polyhedra Library::Select_Temp_Boundary_Type< Interval_Boundary_Type > Struct Template Reference

Helper class to select the appropriate numerical type to perform boundary computations so as to reduce the chances of overflow without incurring too much overhead.

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

10.85.1 Detailed Description

```
template<typename Interval_Boundary_Type>struct Parma_Polyhedra_Library::Select_Temp_Boundary-  
_Type< Interval_Boundary_Type >
```

Helper class to select the appropriate numerical type to perform boundary computations so as to reduce the chances of overflow without incurring too much overhead.

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

- ppl.hh

10.86 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.

10.86.1 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 equalities, the shapes of the domains are unaltered.

10.86.2 Member Function Documentation

```
template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Shape_Preserving_Reduction<  
D1, D2 >::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.

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

<i>d1</i>	A pointset domain element;
<i>d2</i>	A pointset domain element;

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

- ppl.hh

10.87 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.

10.87.1 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.

10.87.2 Member Function Documentation

template<typename D1 , typename D2 > void Parma_Polyhedra_Library::Smash_Reduction< D1, D2 >::product_reduce (D1 &d1, D2 &d2) 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

<i>d1</i>	A pointset domain element;
<i>d2</i>	A pointset domain element;

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

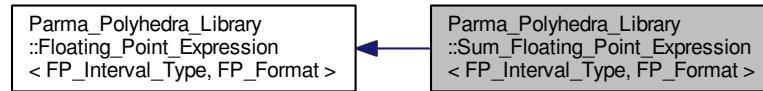
- ppl.hh

10.88 Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic Sum Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form](#) [FP_Linear_Form](#)
Alias for the Linear_Form<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Interval_Abstract_Store](#) [FP_Interval_Abstract_Store](#)
Alias for the Box<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[FP_Linear_Form_Abstract_Store](#) [FP_Linear_Form_Abstract_Store](#)
Alias for the std::map<dimension_type, FP_Linear_Form> from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[boundary_type](#) [boundary_type](#)
Alias for the FP_Interval_Type::boundary_type from [Floating_Point_Expression](#).
- typedef
[Floating_Point_Expression](#)
 < FP_Interval_Type, FP_Format >
 ::[info_type](#) [info_type](#)
Alias for the FP_Interval_Type::info_type from [Floating_Point_Expression](#).

Public Member Functions

- bool [linearize](#) (const [FP_Interval_Abstract_Store](#) &int_store, const [FP_Linear_Form_Abstract_Store](#) &lf_store, [FP_Linear_Form](#) &result) const
Linearizes the expression in a given astract store.
- void [m_swap](#) ([Sum_Floating_Point_Expression](#)< FP_Interval_Type, FP_Format > &y)
*Swaps *this with y.*

Constructors and Destructor

- [Sum_Floating_Point_Expression](#) ([Floating_Point_Expression](#)< FP_Interval_Type, FP_Format > *const x, [Floating_Point_Expression](#)< FP_Interval_Type, FP_Format > *const y)

- *Constructor with two parameters: builds the sum floating point expression corresponding to $x \oplus y$.*
`~Sum_Floating_Point_Expression ()`
Destructor.

Related Functions

(Note that these are not member functions.)

- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap (Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`
Swaps x with y .
- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap (Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x, Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y)`

Additional Inherited Members

10.88.1 Detailed Description

template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >

A generic Sum Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of sum floating-point expressions

Let $i + \sum_{v \in \mathcal{V}} i_v v$ and $i' + \sum_{v \in \mathcal{V}} i'_v v$ be two linear forms and $\boxplus^\#$ a sound abstract operator on linear forms such that:

$$\left(i + \sum_{v \in \mathcal{V}} i_v v \right) \boxplus^\# \left(i' + \sum_{v \in \mathcal{V}} i'_v v \right) = (i \oplus^\# i') + \sum_{v \in \mathcal{V}} (i_v \oplus^\# i'_v) v.$$

Given an expression $e_1 \oplus e_2$ and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket e_1 \oplus e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as follows:

$$\llbracket e_1 \oplus e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket = \llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \boxplus^\# \varepsilon_f \left(\llbracket e_1 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# \varepsilon_f \left(\llbracket e_2 \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket \right) \boxplus^\# m_{f_f}[-1, 1]$$

where $\varepsilon_f(l)$ is the linear form computed by calling method `Floating_Point_Expression::relative_error` on l and m_{f_f} is a rounding error defined in `Floating_Point_Expression::absolute_error`.

10.88.2 Member Function Documentation

template<typename FP_Interval_Type, typename FP_Format > bool Parma_Polyhedra_Library::Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const **[virtual]** Linearizes the expression in a given astract store.

Makes `result` become the linearization of `*this` in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

Note that all variables occurring in the expressions represented by `first_operand` and `second_operand` MUST have an associated value in `int_store`. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how `result` is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

10.88.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] Swaps `x` with `y`.

template<typename FP_Interval_Type , typename FP_Format > void swap (Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Sum_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y) [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.89 Parma_Polyhedra_Library::Threshold_Watcher< Traits > Class Template Reference

A class of watchdogs controlling the exceeding of a threshold.

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

10.89.1 Detailed Description

template<typename Traits>class Parma_Polyhedra_Library::Threshold_Watcher< Traits >

A class of watchdogs controlling the exceeding of a threshold.

Template Parameters

<i>Traits</i>	A class to set data types and functions for the threshold handling. See <code>Parma_Polyhedra_Library::Weightwatch_Traits</code> for an example.
---------------	--

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

- `ppl.hh`

10.90 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.

10.90.1 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

10.91 Parma_Polyhedra_Library::Implementation::Watchdog::Time Class Reference

A class for representing and manipulating positive time intervals.

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

Public Member Functions

- [Time](#) ()
Zero seconds.
- [Time](#) (long centisecs)
Constructor taking a number of centiseconds.
- [Time](#) (long s, long m)
Constructor with seconds and microseconds.
- long [seconds](#) () const
Returns the number of whole seconds contained in the represented time interval.
- long [microseconds](#) () const
Returns the number of microseconds that, when added to the number of seconds returned by [seconds\(\)](#), give the represent time interval.
- [Time](#) & [operator+=](#) (const [Time](#) &y)
*Adds y to *this.*
- [Time](#) & [operator-=](#) (const [Time](#) &y)
*Subtracts y from *this; if *this is shorter than y, *this is set to the null interval.*
- bool [OK](#) () const
Checks if all the invariants are satisfied.

10.91.1 Detailed Description

A class for representing and manipulating positive time intervals.

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

- ppl.hh

10.92 Parma_Polyhedra_Library::Unary_Operator< Target > Class Template Reference

A unary operator applied to one concrete expression.

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


10.92.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Unary_Operator< Target >

A unary operator applied to one concrete expression.

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

- ppl.hh

10.93 Parma_Polyhedra_Library::Unary_Operator_Common< Target > Class Template Reference

Base class for unary operator applied to one concrete expression.

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

Public Member Functions

- [Concrete_Expression_UOP unary_operator](#) () const
*Returns a constant identifying the operator of *this.*
- const [Concrete_Expression](#)
< Target > * [argument](#) () const
*Returns the argument *this.*

10.93.1 Detailed Description

template<typename Target>class Parma_Polyhedra_Library::Unary_Operator_Common< Target >

Base class for unary operator applied to one concrete expression.

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

- ppl.hh

10.94 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.
- `void m_swap (Variable &v)`
*Swaps `*this` and `v`.*

Static Public Member Functions

- `static dimension_type max_space_dimension ()`
Returns the maximum space dimension a `Variable` can handle.
- `static void default_output_function (std::ostream &s, const Variable v)`
The default output function.
- `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.
- `void swap (Variable &x, Variable &y)`
- `bool less (const Variable v, const Variable w)`

10.94.1 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 misled 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;
```

10.94.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Variable::Variable (dimension_type *i*) [inline], [explicit]

Builds the variable corresponding to the Cartesian axis of index *i*.

Exceptions

<code>std::length_error</code>	Thrown if <code>i+1</code> exceeds <code>Variable::max_space_dimension()</code> .
--------------------------------	---

10.94.3 Member Function Documentation

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.

10.94.4 Friends And Related Function Documentation

`std::ostream & operator<< (std::ostream & s, const Variable v)` [related] Output operator.

`bool less (Variable v, Variable w)` [related] Defines a total ordering on variables.

`void swap (Variable & x, Variable & y)` [related]

`bool less (const Variable v, const Variable w)` [related] The documentation for this class was generated from the following file:

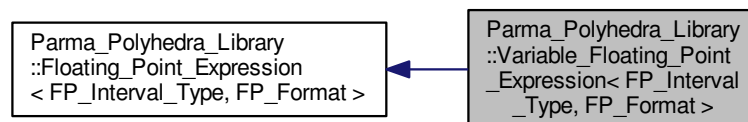
- ppl.hh

10.95 Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > Class Template Reference

A generic [Variable](#) Floating Point Expression.

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

Inheritance diagram for Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >:



Public Types

- typedef [Floating_Point_Expression](#) < FP_Interval_Type, FP_Format > ::FP_Linear_Form [FP_Linear_Form](#)
Alias for the Linear_Form<FP_Interval_Type> from [Floating_Point_Expression](#).
- typedef [Floating_Point_Expression](#) < FP_Interval_Type, FP_Format > ::FP_Interval_Abstract_Store [FP_Interval_Abstract_Store](#)

- *Alias for the `Box<FP_Interval_Type>` from `Floating_Point_Expression`.*
- `typedef`
`Floating_Point_Expression`
`< FP_Interval_Type, FP_Format >`
`::FP_Linear_Form_Abstract_Store FP_Linear_Form_Abstract_Store`
Alias for the `std::map<dimension_type, FP_Linear_Form>` from `Floating_Point_Expression`.
- `typedef`
`Floating_Point_Expression`
`< FP_Interval_Type, FP_Format >`
`::boundary_type boundary_type`
Alias for the `FP_Interval_Type::boundary_type` from `Floating_Point_Expression`.
- `typedef`
`Floating_Point_Expression`
`< FP_Interval_Type, FP_Format >`
`::info_type info_type`
Alias for the `FP_Interval_Type::info_type` from `Floating_Point_Expression`.

Public Member Functions

- `bool linearize` (`const FP_Interval_Abstract_Store &int_store`, `const FP_Linear_Form_Abstract_Store &lf_store`, `FP_Linear_Form &result`) `const`
Linearizes the expression in a given abstract store.
- `void linear_form_assign` (`const FP_Linear_Form &lf`, `FP_Linear_Form_Abstract_Store &lf_store`) `const`
*Assigns a linear form to the variable with the same index of `*this` in a given linear form abstract store.*
- `void m_swap` (`Variable_Floating_Point_Expression &y`)
*Swaps `*this` with `y`.*

Constructors and Destructor

- `Variable_Floating_Point_Expression` (`const dimension_type v_index`)
Constructor with a parameter: builds the variable floating point expression corresponding to the variable having `v_index` as its index.
- `~Variable_Floating_Point_Expression` ()
Destructor.

Related Functions

(Note that these are not member functions.)

- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap` (`Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x`, `Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y`)
Swaps `x` with `y`.
- `template<typename FP_Interval_Type, typename FP_Format >`
`void swap` (`Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > &x`, `Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > &y`)

Additional Inherited Members

10.95.1 Detailed Description

`template<typename FP_Interval_Type, typename FP_Format>class Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >`

A generic `Variable` Floating Point Expression.

Template type parameters

- The class template type parameter `FP_Interval_Type` represents the type of the intervals used in the abstract domain.
- The class template type parameter `FP_Format` represents the floating point format used in the concrete domain.

Linearization of floating-point variable expressions

Given a variable expression v and a composite abstract store $\llbracket \rho^\#, \rho_l^\# \rrbracket$, we construct the interval linear form $\llbracket v \rrbracket \llbracket \rho^\#, \rho_l^\# \rrbracket$ as $\rho_l^\#(v)$ if it is defined; otherwise we construct it as $[-1, 1]v$.

10.95.2 Member Function Documentation

template<typename FP_Interval_Type , typename FP_Format > bool Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linearize (const FP_Interval_Abstract_Store & *int_store*, const FP_Linear_Form_Abstract_Store & *lf_store*, FP_Linear_Form & *result*) const **[inline], [virtual]** Linearizes the expression in a given abstract store.

Makes *result* become the linearization of **this* in the given composite abstract store.

Parameters

<i>int_store</i>	The interval abstract store.
<i>lf_store</i>	The linear form abstract store.
<i>result</i>	The modified linear form.

Returns

`true` if the linearization succeeded, `false` otherwise.

Note that the variable in the expression MUST have an associated value in *int_store*. If this precondition is not met, calling the method causes an undefined behavior.

See the class description for a detailed explanation of how *result* is computed.

Implements [Parma_Polyhedra_Library::Floating_Point_Expression< FP_Interval_Type, FP_Format >](#).

template<typename FP_Interval_Type , typename FP_Format > void Parma_Polyhedra_Library::Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format >::linear_form_assign (const FP_Linear_Form & *lf*, FP_Linear_Form_Abstract_Store & *lf_store*) const **[inline]** Assigns a linear form to the variable with the same index of **this* in a given linear form abstract store.

Parameters

<i>lf</i>	The linear form assigned to the variable.
<i>lf_store</i>	The linear form abstract store.

Note that once *lf* is assigned to a variable, all the other entries of *lf_store* which contain that variable are discarded.

10.95.3 Friends And Related Function Documentation

template<typename FP_Interval_Type , typename FP_Format > void swap (Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *x*, Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > & *y*) **[related]** Swaps *x* with *y*.

`template<typename FP_Interval_Type , typename FP_Format > void swap (Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > & x, Variable_Floating_Point_Expression< FP_Interval_Type, FP_Format > & y)` [**related**] The documentation for this class was generated from the following file:

- `ppl.hh`

10.96 Parma_Polyhedra_Library::Variables_Set Class Reference

An `std::set` of variables' indexes.

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

Inherits `set< dimension_type >`.

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](#) &vs)
Output operator.

10.96.1 Detailed Description

An `std::set` of variables' indexes.

10.96.2 Constructor & Destructor Documentation

Parma_Polyhedra_Library::Variables_Set::Variables_Set (const Variable *v*, const Variable *w*)

Builds the set of variables' 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.

10.96.3 Friends And Related Function Documentation

`std::ostream & operator<< (std::ostream & s, const Variables_Set & vs)` **[related]** Output operator.

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

- `ppl.hh`

10.97 Parma_Polyhedra_Library::Watchdog Class Reference

A watchdog timer.

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

Public Member Functions

- [Watchdog](#) (long *csecs*, void(*const function)())
Constructor: if not reset, the watchdog will trigger after csecs centiseconds, invoking handler function.
- [~Watchdog](#) ()
Destructor.

Static Public Member Functions

- static void [initialize](#) ()
Static class initialization.
- static void [finalize](#) ()
Static class finalization.

10.97.1 Detailed Description

A watchdog timer.

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

- `ppl.hh`

Index

- ANY_COMPLEXITY
 - C++ Language Interface, [69](#)
- abandon_expensive_computations
 - C++ Language Interface, [77](#)
- abs_assign
 - Parma_Polyhedra_Library::Checked_Number, [170](#)
 - Parma_Polyhedra_Library::GMP_Integer, [261](#)
- absolute_error
 - Parma_Polyhedra_Library::Floating_Point_Expression, [240](#)
- add_and_minimize
 - Parma_Polyhedra_Library::Polyhedron, [489](#), [490](#)
- add_congruence
 - Parma_Polyhedra_Library::BD_Shape, [104](#)
 - Parma_Polyhedra_Library::Box, [136](#)
 - Parma_Polyhedra_Library::Grid, [277](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [366](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [400](#)
 - Parma_Polyhedra_Library::Pointset_Powerset, [446](#)
 - Parma_Polyhedra_Library::Polyhedron, [473](#)
- add_congruences
 - Parma_Polyhedra_Library::BD_Shape, [105](#)
 - Parma_Polyhedra_Library::Box, [136](#)
 - Parma_Polyhedra_Library::Grid, [277](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [366](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [401](#)
 - Parma_Polyhedra_Library::Pointset_Powerset, [446](#)
 - Parma_Polyhedra_Library::Polyhedron, [474](#)
- add_constraint
 - Parma_Polyhedra_Library::BD_Shape, [104](#)
 - Parma_Polyhedra_Library::Box, [135](#)
 - Parma_Polyhedra_Library::Grid, [278](#)
 - Parma_Polyhedra_Library::MIP_Problem, [340](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [365](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [400](#)
 - Parma_Polyhedra_Library::PIP_Problem, [425](#)
 - Parma_Polyhedra_Library::Pointset_Powerset, [445](#)
 - Parma_Polyhedra_Library::Polyhedron, [472](#)
- add_constraints
 - Parma_Polyhedra_Library::BD_Shape, [104](#)
 - Parma_Polyhedra_Library::Box, [135](#)
 - Parma_Polyhedra_Library::Grid, [278](#)
 - Parma_Polyhedra_Library::MIP_Problem, [340](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [365](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [401](#)
 - Parma_Polyhedra_Library::PIP_Problem, [425](#)
 - Parma_Polyhedra_Library::Pointset_Powerset, [445](#)
 - Parma_Polyhedra_Library::Polyhedron, [473](#)
- add_disjunct
 - Parma_Polyhedra_Library::Pointset_Powerset, [445](#)
- add_generator
 - Parma_Polyhedra_Library::Polyhedron, [473](#)
- add_generators
 - Parma_Polyhedra_Library::Polyhedron, [473](#)
- add_grid_generator
 - Parma_Polyhedra_Library::Grid, [277](#)
- add_grid_generators
 - Parma_Polyhedra_Library::Grid, [279](#)
- add_linearize
 - Parma_Polyhedra_Library::Concrete_Expression, [177](#)
- add_mul_assign
 - Parma_Polyhedra_Library::Checked_Number, [170](#)
 - Parma_Polyhedra_Library::GMP_Integer, [261](#)
 - Parma_Polyhedra_Library::Linear_Expression, [324](#), [325](#)
- add_non_bottom_disjunct_preserve_reduction
 - Parma_Polyhedra_Library::Powerset, [496](#)
- add_recycled_congruences
 - Parma_Polyhedra_Library::BD_Shape, [105](#)
 - Parma_Polyhedra_Library::Box, [136](#)
 - Parma_Polyhedra_Library::Grid, [277](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [366](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [401](#)
 - Parma_Polyhedra_Library::Polyhedron, [474](#)
- add_recycled_constraints
 - Parma_Polyhedra_Library::BD_Shape, [104](#)
 - Parma_Polyhedra_Library::Box, [136](#)
 - Parma_Polyhedra_Library::Grid, [278](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [365](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [402](#)
 - Parma_Polyhedra_Library::Polyhedron, [473](#)
- add_recycled_generators
 - Parma_Polyhedra_Library::Polyhedron, [474](#)
- add_recycled_grid_generators
 - Parma_Polyhedra_Library::Grid, [279](#)
- add_space_dimensions_and_embed
 - Parma_Polyhedra_Library::BD_Shape, [115](#)
 - Parma_Polyhedra_Library::Box, [145](#)
 - Parma_Polyhedra_Library::Grid, [288](#)
 - Parma_Polyhedra_Library::MIP_Problem, [340](#)
 - Parma_Polyhedra_Library::Octagonal_Shape, [377](#)
 - Parma_Polyhedra_Library::Partially_Reduced_Product, [409](#)
 - Parma_Polyhedra_Library::PIP_Problem, [424](#)
 - Parma_Polyhedra_Library::Polyhedron, [484](#)

add_space_dimensions_and_project
 Parma_Polyhedra_Library::BD_Shape, 115
 Parma_Polyhedra_Library::Box, 145
 Parma_Polyhedra_Library::Grid, 288
 Parma_Polyhedra_Library::Octagonal_Shape, 377
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 409
 Parma_Polyhedra_Library::Polyhedron, 484
 add_to_integer_space_dimensions
 Parma_Polyhedra_Library::MIP_Problem, 340
 add_to_parameter_space_dimensions
 Parma_Polyhedra_Library::PIP_Problem, 425
 add_unit_rows_and_space_dimensions
 Parma_Polyhedra_Library::Congruence_System, 196
 affine_form_image
 Parma_Polyhedra_Library::BD_Shape, 109
 Parma_Polyhedra_Library::Box, 140
 Parma_Polyhedra_Library::Linear_Form, 331
 Parma_Polyhedra_Library::Octagonal_Shape, 372
 Parma_Polyhedra_Library::Polyhedron, 478
 affine_image
 Parma_Polyhedra_Library::BD_Shape, 109
 Parma_Polyhedra_Library::Box, 139
 Parma_Polyhedra_Library::Grid, 282
 Parma_Polyhedra_Library::Octagonal_Shape, 370
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 403
 Parma_Polyhedra_Library::Pointset_Powerset, 448
 Parma_Polyhedra_Library::Polyhedron, 478
 affine_preimage
 Parma_Polyhedra_Library::BD_Shape, 109
 Parma_Polyhedra_Library::Box, 140
 Parma_Polyhedra_Library::Grid, 282
 Parma_Polyhedra_Library::Octagonal_Shape, 372
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 403
 Parma_Polyhedra_Library::Pointset_Powerset, 448
 Parma_Polyhedra_Library::Polyhedron, 478
 all_affine_quasi_ranking_functions_MS
 C++ Language Interface, 75
 all_affine_quasi_ranking_functions_MS_2
 C++ Language Interface, 76
 all_affine_ranking_functions_MS
 C++ Language Interface, 74
 all_affine_ranking_functions_MS_2
 C++ Language Interface, 74
 all_affine_ranking_functions_PR
 C++ Language Interface, 77
 all_affine_ranking_functions_PR_2
 C++ Language Interface, 77
 approximate_partition
 Parma_Polyhedra_Library::Pointset_Powerset, 454
 Artificial_Parameter
 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial-
 Parameter, 90
 ascii_dump
 Parma_Polyhedra_Library::Checked_Number, 172
 ascii_load
 Parma_Polyhedra_Library::Checked_Number, 174
 Parma_Polyhedra_Library::Generator_System, 258
 Parma_Polyhedra_Library::Grid_Generator_System, 304
 assign_r
 Parma_Polyhedra_Library::Checked_Number, 169, 174
 BITS_128
 C++ Language Interface, 69
 BITS_16
 C++ Language Interface, 69
 BITS_32
 C++ Language Interface, 69
 BITS_64
 C++ Language Interface, 69
 BITS_8
 C++ Language Interface, 69
 BD_Shape
 Parma_Polyhedra_Library::BD_Shape, 99, 100
 BGP99_extrapolation_assign
 Parma_Polyhedra_Library::Pointset_Powerset, 451
 BHMZ05_widening_assign
 Parma_Polyhedra_Library::BD_Shape, 113
 Parma_Polyhedra_Library::Octagonal_Shape, 376
 BHRZ03_widening_assign
 Parma_Polyhedra_Library::Polyhedron, 482
 BHZ03_widening_assign
 Parma_Polyhedra_Library::Pointset_Powerset, 452
 banner
 Parma_Polyhedra_Library, 86
 begin
 Parma_Polyhedra_Library::Linear_Expression, 321
 bounded_BHRZ03_extrapolation_assign
 Parma_Polyhedra_Library::Polyhedron, 483
 bounded_H79_extrapolation_assign
 Parma_Polyhedra_Library::Polyhedron, 484
 Bounded_Integer_Type_Overflow
 C++ Language Interface, 69
 Bounded_Integer_Type_Representation
 C++ Language Interface, 69
 Bounded_Integer_Type_Width
 C++ Language Interface, 69
 bounded_affine_image
 Parma_Polyhedra_Library::BD_Shape, 111
 Parma_Polyhedra_Library::Box, 141
 Parma_Polyhedra_Library::Grid, 284
 Parma_Polyhedra_Library::Octagonal_Shape, 373

- Parma.Polyhedra.Library::Partially_Reduced_Product, 407
- Parma.Polyhedra.Library::Pointset_Powerset, 450
- Parma.Polyhedra.Library::Polyhedron, 480
- bounded_affine_preimage
 - Parma.Polyhedra.Library::BD_Shape, 111
 - Parma.Polyhedra.Library::Box, 142
 - Parma.Polyhedra.Library::Grid, 284
 - Parma.Polyhedra.Library::Octagonal_Shape, 374
 - Parma.Polyhedra.Library::Partially_Reduced_Product, 407
 - Parma.Polyhedra.Library::Pointset_Powerset, 450
 - Parma.Polyhedra.Library::Polyhedron, 480
- bounded_integer_type_overflow
 - Parma.Polyhedra.Library::Concrete_Expression_Type, 183
- bounded_integer_type_representation
 - Parma.Polyhedra.Library::Concrete_Expression_Type, 183
- bounded_integer_type_width
 - Parma.Polyhedra.Library::Concrete_Expression_Type, 183
- bounds_from_above
 - Parma.Polyhedra.Library::BD_Shape, 101
 - Parma.Polyhedra.Library::Box, 133
 - Parma.Polyhedra.Library::Grid, 274
 - Parma.Polyhedra.Library::Octagonal_Shape, 363
 - Parma.Polyhedra.Library::Partially_Reduced_Product, 398
 - Parma.Polyhedra.Library::Pointset_Powerset, 442
 - Parma.Polyhedra.Library::Polyhedron, 470
- bounds_from_below
 - Parma.Polyhedra.Library::BD_Shape, 101
 - Parma.Polyhedra.Library::Box, 133
 - Parma.Polyhedra.Library::Grid, 275
 - Parma.Polyhedra.Library::Octagonal_Shape, 363
 - Parma.Polyhedra.Library::Partially_Reduced_Product, 398
 - Parma.Polyhedra.Library::Pointset_Powerset, 442
 - Parma.Polyhedra.Library::Polyhedron, 470
- Box
 - Parma.Polyhedra.Library::Box, 129, 131, 132
- C++ Language Interface, 61
 - ANY_COMPLEXITY, 69
 - abandon_expensive_computations, 77
 - all_affine_quasi_ranking_functions_MS, 75
 - all_affine_quasi_ranking_functions_MS_2, 76
 - all_affine_ranking_functions_MS, 74
 - all_affine_ranking_functions_MS_2, 74
 - all_affine_ranking_functions_PR, 77
 - all_affine_ranking_functions_PR_2, 77
 - BITS_128, 69
 - BITS_16, 69
 - BITS_32, 69
 - BITS_64, 69
 - BITS_8, 69
 - Bounded_Integer_Type_Overflow, 69
 - Bounded_Integer_Type_Representation, 69
 - Bounded_Integer_Type_Width, 69
 - Coefficient, 67
 - Complexity_Class, 68
 - DENSE, 70
 - Degenerate_Element, 68
 - dimension_type, 67
 - EMPTY, 68
 - EQUAL, 68
 - Floating_Point_Format, 70
 - GREATER_OR_EQUAL, 68
 - GREATER_THAN, 68
 - IBM_DOUBLE, 70
 - IBM_SINGLE, 70
 - IEEE754_DOUBLE, 70
 - IEEE754_HALF, 70
 - IEEE754_QUAD, 70
 - IEEE754_SINGLE, 70
 - INTEL_DOUBLE_EXTENDED, 70
 - inverse, 71
 - LESS_OR_EQUAL, 68
 - LESS_THAN, 68
 - MAXIMIZATION, 69
 - MINIMIZATION, 69
 - MIP_Problem_Status, 70
 - memory_size_type, 67
 - NOT_EQUAL, 68
 - OPTIMIZED_MIP_PROBLEM, 70
 - OPTIMIZED_PIP_PROBLEM, 70
 - OVERFLOW_IMPOSSIBLE, 69
 - OVERFLOW_UNDEFINED, 69
 - OVERFLOW_WRAPS, 69
 - one_affine_ranking_function_MS, 72
 - one_affine_ranking_function_MS_2, 73
 - one_affine_ranking_function_PR, 77
 - one_affine_ranking_function_PR_2, 77
 - operator-, 70
 - operator&, 70, 71
 - Optimization_Mode, 69
 - POLYNOMIAL_COMPLEXITY, 69
 - PIP_Problem_Status, 70
 - PPL_VERSION, 66
 - PPL_VERSION_MAJOR, 66
 - PPL_VERSION_MINOR, 66
 - ROUND_DOWN, 68
 - ROUND_IGNORE, 68
 - ROUND_NOT_NEEDED, 68
 - ROUND_STRICT_RELATION, 68
 - ROUND_UP, 68
 - Relation_Symbol, 68

- Representation, 69
- Result, 67
- result_class, 70
- result_relation, 71
- result_relation_class, 71
- round_dir, 71
- round_direct, 71
- round_down, 71
- round_fpu_dir, 71
- round_ignore, 71
- round_inverse, 71
- round_not_needed, 71
- round_not_requested, 71
- round_strict_relation, 71
- round_up, 71
- Rounding_Dir, 68
- SIGNED_2_COMPLEMENT, 69
- SIMPLEX_COMPLEXITY, 69
- SPARSE, 70
- termination_test_MS, 71
- termination_test_MS_2, 72
- termination_test_PR, 77
- termination_test_PR_2, 77
- UNBOUNDED_MIP_PROBLEM, 70
- UNFEASIBLE_MIP_PROBLEM, 70
- UNFEASIBLE_PIP_PROBLEM, 70
- UNIVERSE, 68
- UNSIGNED, 69
- V_CVT_STR_UNK, 68
- V_DIV_ZERO, 68
- V_EMPTY, 67
- V_EQ, 67
- V_EQ_MINUS_INFINITY, 67
- V_EQ_PLUS_INFINITY, 67
- V_GE, 67
- V_GT, 67
- V_GT_MINUS_INFINITY, 67
- V_GT_SUP, 67
- V_INF_ADD_INF, 68
- V_INF_DIV_INF, 68
- V_INF_MOD, 68
- V_INF_MUL_ZERO, 68
- V_INF_SUB_INF, 68
- V_LE, 67
- V_LGE, 67
- V_LT, 67
- V_LT_INF, 67
- V_LT_PLUS_INFINITY, 67
- V_MOD_ZERO, 68
- V_NAN, 68
- V_NE, 67
- V_OVERFLOW, 67
- V_SQRT_NEG, 68
- V_UNKNOWN_NEG_OVERFLOW, 68

- V_UNKNOWN_POS_OVERFLOW, 68
- V_UNREPRESENTABLE, 68
- CLOSURE_POINT
 - Parma_Polyhedra_Library::Generator, 249
- CUTTING_STRATEGY
 - Parma_Polyhedra_Library::PIP_Problem, 423
- CUTTING_STRATEGY_ALL
 - Parma_Polyhedra_Library::PIP_Problem, 424
- CUTTING_STRATEGY_DEEPEST
 - Parma_Polyhedra_Library::PIP_Problem, 424
- CUTTING_STRATEGY_FIRST
 - Parma_Polyhedra_Library::PIP_Problem, 424
- C_Polyhedron
 - Parma_Polyhedra_Library::C_Polyhedron, 153–156
- CC76_extrapolation_assign
 - Parma_Polyhedra_Library::BD_Shape, 112, 113
 - Parma_Polyhedra_Library::Octagonal_Shape, 375, 376
- CC76_narrowing_assign
 - Parma_Polyhedra_Library::BD_Shape, 113
 - Parma_Polyhedra_Library::Box, 144
 - Parma_Polyhedra_Library::Octagonal_Shape, 376
- CC76_widening_assign
 - Parma_Polyhedra_Library::Box, 143, 144
- cast_linearize
 - Parma_Polyhedra_Library::Concrete_Expression, 181
- ceil_assign
 - Parma_Polyhedra_Library::Checked_Number, 170
- check_containment
 - Parma_Polyhedra_Library::Pointset_Powerset, 454
- classify
 - Parma_Polyhedra_Library::Checked_Number, 169
- clear
 - Parma_Polyhedra_Library::MIP_Problem, 340
 - Parma_Polyhedra_Library::PIP_Problem, 424
- clone
 - Parma_Polyhedra_Library::Linear_Expression_-Impl::const_iterator, 201
 - Parma_Polyhedra_Library::Linear_Expression_-Interface::const_iterator_interface, 207
- closure_point
 - Parma_Polyhedra_Library::Generator, 250, 251, 254
- cmp
 - Parma_Polyhedra_Library::Checked_Number, 174
- Coefficient
 - C++ Language Interface, 67
- coefficient
 - Parma_Polyhedra_Library::Congruence, 189
 - Parma_Polyhedra_Library::Constraint, 216
 - Parma_Polyhedra_Library::Generator, 251
 - Parma_Polyhedra_Library::Grid_Generator, 298

compare
 Parma_Polyhedra_Library::BHRZ03_Certificate, 121
 Parma_Polyhedra_Library::Grid_Certificate, 292
 Parma_Polyhedra_Library::H79_Certificate, 305
 compatibility_check
 Parma_Polyhedra_Library::PIP_Tree_Node, 433
 Complexity_Class
 C++ Language Interface, 68
 concatenate_assign
 Parma_Polyhedra_Library::BD_Shape, 115
 Parma_Polyhedra_Library::Box, 145
 Parma_Polyhedra_Library::Grid, 289
 Parma_Polyhedra_Library::Octagonal_Shape, 377
 Parma_Polyhedra_Library::Partially_Reduced_Product, 409
 Parma_Polyhedra_Library::Pointset_Powerset, 452
 Parma_Polyhedra_Library::Polyhedron, 484
 Concrete_Expression_BOP
 Parma_Polyhedra_Library, 85
 Concrete_Expression_Kind
 Parma_Polyhedra_Library, 85
 Concrete_Expression_UOP
 Parma_Polyhedra_Library, 85
 Congruence
 Parma_Polyhedra_Library::Congruence, 188
 Congruence_System
 Parma_Polyhedra_Library::Congruence_System, 194
 congruence_widening_assign
 Parma_Polyhedra_Library::Grid, 285
 const_iterator
 Parma_Polyhedra_Library::CO_Tree::const_iterator, 198, 199
 Parma_Polyhedra_Library::Linear_Expression::const_iterator, 202
 constrains
 Parma_Polyhedra_Library::BD_Shape, 104
 Parma_Polyhedra_Library::Box, 132
 Parma_Polyhedra_Library::Grid, 274
 Parma_Polyhedra_Library::Octagonal_Shape, 363
 Parma_Polyhedra_Library::Partially_Reduced_Product, 398
 Parma_Polyhedra_Library::Pointset_Powerset, 442
 Parma_Polyhedra_Library::Polyhedron, 470
 Constraint
 Parma_Polyhedra_Library::Constraint, 215, 216
 Constraint_System
 Parma_Polyhedra_Library::Constraint_System, 222
 constraints
 Parma_Polyhedra_Library::PIP_Tree_Node, 431
 construct
 Parma_Polyhedra_Library::Checked_Number, 169
 contains
 Parma_Polyhedra_Library::BD_Shape, 103
 Parma_Polyhedra_Library::Box, 135
 Parma_Polyhedra_Library::Grid, 276
 Parma_Polyhedra_Library::Octagonal_Shape, 362
 Parma_Polyhedra_Library::Partially_Reduced_Product, 400
 Parma_Polyhedra_Library::Pointset_Powerset, 444
 Parma_Polyhedra_Library::Polyhedron, 472
 Control_Parameter_Name
 Parma_Polyhedra_Library::MIP_Problem, 338
 Parma_Polyhedra_Library::PIP_Problem, 423
 Control_Parameter_Value
 Parma_Polyhedra_Library::MIP_Problem, 338
 Parma_Polyhedra_Library::PIP_Problem, 423
 conversion
 Parma_Polyhedra_Library::Polyhedron, 489
 convert_to_integer_expression
 Parma_Polyhedra_Library::Polyhedron, 488
 convert_to_integer_expressions
 Parma_Polyhedra_Library::Polyhedron, 488
 DENSE
 C++ Language Interface, 70
 default_representation
 Parma_Polyhedra_Library::Congruence, 191
 Parma_Polyhedra_Library::Constraint, 219
 Parma_Polyhedra_Library::Generator, 255
 Parma_Polyhedra_Library::Grid_Generator, 300
 Degenerate_Element
 C++ Language Interface, 68
 difference_assign
 Parma_Polyhedra_Library::BD_Shape, 108
 Parma_Polyhedra_Library::Box, 139
 Parma_Polyhedra_Library::Grid, 280
 Parma_Polyhedra_Library::Octagonal_Shape, 370
 Parma_Polyhedra_Library::Partially_Reduced_Product, 403
 Parma_Polyhedra_Library::Pointset_Powerset, 447
 dimension_type
 C++ Language Interface, 67
 discard_occurrences
 Parma_Polyhedra_Library::Linear_Form, 331
 div_2exp_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 262
 div_assign
 Parma_Polyhedra_Library::Interval, 311
 div_linearize
 Parma_Polyhedra_Library::Concrete_Expression, 180
 divisor
 Parma_Polyhedra_Library::Generator, 251
 Parma_Polyhedra_Library::Grid_Generator, 299
 drop_some_non_integer_points

Parma_Polyhedra_Library::BD_Shape, 112
 Parma_Polyhedra_Library::Box, 143
 Parma_Polyhedra_Library::Grid, 285
 Parma_Polyhedra_Library::Octagonal_Shape, 375
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 408
 Parma_Polyhedra_Library::Pointset_Powerset, 447
 Parma_Polyhedra_Library::Polyhedron, 482, 487
 EMPTY
 C++ Language Interface, 68
 EQUAL
 C++ Language Interface, 68
 EQUALITY
 Parma_Polyhedra_Library::Constraint, 215
 end
 Parma_Polyhedra_Library::Linear_Expression, 321
 equal
 Parma_Polyhedra_Library::Checked_Number, 171
 euclidean_distance_assign
 Parma_Polyhedra_Library::BD_Shape, 118, 119
 Parma_Polyhedra_Library::Box, 150, 151
 Parma_Polyhedra_Library::Generator, 252–255
 Parma_Polyhedra_Library::Octagonal_Shape, 383, 384
 evaluate_objective_function
 Parma_Polyhedra_Library::MIP_Problem, 342
 exact_div_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 262
 expand_space_dimension
 Parma_Polyhedra_Library::BD_Shape, 116
 Parma_Polyhedra_Library::Box, 146
 Parma_Polyhedra_Library::Grid, 290
 Parma_Polyhedra_Library::Octagonal_Shape, 379
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 410
 Parma_Polyhedra_Library::Pointset_Powerset, 453
 Parma_Polyhedra_Library::Polyhedron, 486
 export_interval_constraints
 Parma_Polyhedra_Library::BD_Shape, 107
 Parma_Polyhedra_Library::Octagonal_Shape, 380
 external_memory_in_bytes
 Parma_Polyhedra_Library::Checked_Number, 174
 FP_Interval_Abstract_Store
 Parma_Polyhedra_Library::Floating_Point_Expression, 239
 feasible_point
 Parma_Polyhedra_Library::MIP_Problem, 342
 Floating_Point.Format
 C++ Language Interface, 70
 floating_point_format
 Parma_Polyhedra_Library::Concrete_Expression_-
 _Type, 183
 floor_assign
 Parma_Polyhedra_Library::Checked_Number, 170
 fold_space_dimensions
 Parma_Polyhedra_Library::BD_Shape, 116
 Parma_Polyhedra_Library::Box, 146
 Parma_Polyhedra_Library::Grid, 290
 Parma_Polyhedra_Library::Octagonal_Shape, 380
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 411
 Parma_Polyhedra_Library::Pointset_Powerset, 453
 Parma_Polyhedra_Library::Polyhedron, 487
 fpu_check_inexact
 Parma_Polyhedra_Library, 86
 frequency
 Parma_Polyhedra_Library::BD_Shape, 102
 Parma_Polyhedra_Library::Box, 134
 Parma_Polyhedra_Library::Grid, 276
 Parma_Polyhedra_Library::Octagonal_Shape, 365
 Parma_Polyhedra_Library::Polyhedron, 472
 GREATER_OR_EQUAL
 C++ Language Interface, 68
 GREATER_THAN
 C++ Language Interface, 68
 gcd_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 261
 gcdext_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 261
 generalized_affine_image
 Parma_Polyhedra_Library::BD_Shape, 109, 110
 Parma_Polyhedra_Library::Box, 140, 141
 Parma_Polyhedra_Library::Grid, 282, 283
 Parma_Polyhedra_Library::Octagonal_Shape, 372, 373
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 405
 Parma_Polyhedra_Library::Pointset_Powerset, 448, 449
 Parma_Polyhedra_Library::Polyhedron, 479, 480
 generalized_affine_preimage
 Parma_Polyhedra_Library::BD_Shape, 110
 Parma_Polyhedra_Library::Box, 140, 141
 Parma_Polyhedra_Library::Grid, 283
 Parma_Polyhedra_Library::Octagonal_Shape, 373, 374
 Parma_Polyhedra_Library::Partially_Reduced_-
 Product, 405, 407
 Parma_Polyhedra_Library::Pointset_Powerset, 449
 Parma_Polyhedra_Library::Polyhedron, 479, 480
 generalized_refine_with_linear_form_inequality

Parma.Polyhedra.Library::BD_Shape, 106	hash_code
Parma.Polyhedra.Library::Octagonal_Shape, 369	Parma.Polyhedra.Library::BD_Shape, 117
Parma.Polyhedra.Library::Polyhedron, 476	Parma.Polyhedra.Library::Box, 149
generate_cut	Parma.Polyhedra.Library::Grid, 290
Parma.Polyhedra.Library::PIP_Solution_Node, 428	Parma.Polyhedra.Library::Octagonal_Shape, 382
Generator	Parma.Polyhedra.Library::Partially_Reduced_Product, 411
Parma.Polyhedra.Library::Generator, 249	Parma.Polyhedra.Library::Pointset_Powerset, 445
Generator_System	Parma.Polyhedra.Library::Polyhedron, 487
Parma.Polyhedra.Library::Generator_System, 258	
generator_widening_assign	IBM.DOUBLE
Parma.Polyhedra.Library::Grid, 286	C++ Language Interface, 70
geometrically_covers	IBM.SINGLE
Parma.Polyhedra.Library::Pointset_Powerset, 443	C++ Language Interface, 70
geometrically_equals	IEEE754.DOUBLE
Parma.Polyhedra.Library::Pointset_Powerset, 444	C++ Language Interface, 70
get_associated_dimensions	IEEE754.HALF
Parma.Polyhedra.Library::FP_Oracle, 242	C++ Language Interface, 70
get_big_parameter_dimension	IEEE754.QUAD
Parma.Polyhedra.Library::PIP_Problem, 426	C++ Language Interface, 70
get_fp_constant_value	IEEE754.SINGLE
Parma.Polyhedra.Library::FP_Oracle, 241	C++ Language Interface, 70
get_integer_expr_value	INTEL.DOUBLE.EXTENDED
Parma.Polyhedra.Library::FP_Oracle, 242	C++ Language Interface, 70
get_interval	index
Parma.Polyhedra.Library::Box, 148	Parma.Polyhedra.Library::CO_Tree::const_iterator, 200
Parma.Polyhedra.Library::FP_Oracle, 241	Parma.Polyhedra.Library::CO_Tree::iterator, 314
greater_or_equal	infinity_sign
Parma.Polyhedra.Library::Checked_Number, 171	Parma.Polyhedra.Library::Checked_Number, 169
greater_than	input
Parma.Polyhedra.Library::Checked_Number, 172	Parma.Polyhedra.Library::Checked_Number, 172, 175
Grid	insert
Parma.Polyhedra.Library::Grid, 271–274	Parma.Polyhedra.Library::Congruence_System, 194, 196
Grid_Generator	Parma.Polyhedra.Library::Grid_Generator_System, 304
Parma.Polyhedra.Library::Grid_Generator, 298	
Grid_Generator_System	
Parma.Polyhedra.Library::Grid_Generator_System, 304	
grid_line	integer_upper_bound_assign_if_exact
Parma.Polyhedra.Library::Grid_Generator, 298, 300	Parma.Polyhedra.Library::BD_Shape, 108
grid_point	Parma.Polyhedra.Library::Octagonal_Shape, 370
Parma.Polyhedra.Library::Grid_Generator, 298–300	intersection_assign
	Parma.Polyhedra.Library::BD_Shape, 108
	Parma.Polyhedra.Library::Box, 139
	Parma.Polyhedra.Library::Grid, 280
H79_widening_assign	Parma.Polyhedra.Library::Octagonal_Shape, 369
Parma.Polyhedra.Library::BD_Shape, 114	Parma.Polyhedra.Library::Partially_Reduced_Product, 402
Parma.Polyhedra.Library::Polyhedron, 483	Parma.Polyhedra.Library::Pointset_Powerset, 447
has_lower_bound	Parma.Polyhedra.Library::Polyhedron, 477
Parma.Polyhedra.Library::Box, 148	
has_nontrivial_weakening	intervalize
Parma.Polyhedra.Library::Determinate, 226	Parma.Polyhedra.Library::Floating_Point_Expression, 240
has_upper_bound	Parma.Polyhedra.Library::Linear_Form, 331
Parma.Polyhedra.Library::Box, 148	

inverse
 C++ Language Interface, 71
 is_discrete
 Parma_Polyhedra_Library::Grid, 274
 is_disjoint_from
 Parma_Polyhedra_Library::BD_Shape, 103
 Parma_Polyhedra_Library::Box, 135
 Parma_Polyhedra_Library::Grid, 274
 Parma_Polyhedra_Library::Octagonal_Shape, 362
 Parma_Polyhedra_Library::Partially_Reduced_Product, 398
 Parma_Polyhedra_Library::Pointset_Powerset, 441
 Parma_Polyhedra_Library::Polyhedron, 470
 is_equal_to
 Parma_Polyhedra_Library::Constraint, 217
 Parma_Polyhedra_Library::Generator, 251
 Parma_Polyhedra_Library::Grid_Generator, 299
 Parma_Polyhedra_Library::Linear_Expression, 321
 is_equality
 Parma_Polyhedra_Library::Congruence, 189
 is_equivalent_to
 Parma_Polyhedra_Library::Constraint, 216
 Parma_Polyhedra_Library::Generator, 251
 Parma_Polyhedra_Library::Grid_Generator, 299
 is_inconsistent
 Parma_Polyhedra_Library::Congruence, 189
 Parma_Polyhedra_Library::Constraint, 216
 is_integer
 Parma_Polyhedra_Library::Checked_Number, 169
 is_minus_infinity
 Parma_Polyhedra_Library::Checked_Number, 169
 is_not_a_number
 Parma_Polyhedra_Library::Checked_Number, 169
 is_plus_infinity
 Parma_Polyhedra_Library::Checked_Number, 169
 is_proper_congruence
 Parma_Polyhedra_Library::Congruence, 189
 is_satisfiable
 Parma_Polyhedra_Library::MIP_Problem, 342
 Parma_Polyhedra_Library::PIP_Problem, 425
 is_tautological
 Parma_Polyhedra_Library::Congruence, 189
 Parma_Polyhedra_Library::Constraint, 216
 is_topologically_closed
 Parma_Polyhedra_Library::Grid, 274
 iterator
 Parma_Polyhedra_Library::CO_Tree::iterator, 313
 Parma_Polyhedra_Library::Powerset, 496
 LESS_OR_EQUAL
 C++ Language Interface, 68
 LESS_THAN
 C++ Language Interface, 68
 LINE
 Parma_Polyhedra_Library::Generator, 249
 Parma_Polyhedra_Library::Grid_Generator, 297
 l_infinity_distance_assign
 Parma_Polyhedra_Library::BD_Shape, 118–120
 Parma_Polyhedra_Library::Box, 150, 151
 Parma_Polyhedra_Library::Generator, 253, 255
 Parma_Polyhedra_Library::Octagonal_Shape, 383, 384
 lcm_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 261
 Parma_Polyhedra_Library::Variable, 506
 less_or_equal
 Parma_Polyhedra_Library::Checked_Number, 172
 less_than
 Parma_Polyhedra_Library::Checked_Number, 172
 limited_BHMZ05_extrapolation_assign
 Parma_Polyhedra_Library::BD_Shape, 113
 Parma_Polyhedra_Library::Octagonal_Shape, 376
 limited_BHRZ03_extrapolation_assign
 Parma_Polyhedra_Library::Polyhedron, 482
 limited_CC76_extrapolation_assign
 Parma_Polyhedra_Library::BD_Shape, 114
 Parma_Polyhedra_Library::Box, 144
 Parma_Polyhedra_Library::Octagonal_Shape, 377
 limited_H79_extrapolation_assign
 Parma_Polyhedra_Library::BD_Shape, 114
 Parma_Polyhedra_Library::Polyhedron, 483
 limited_congruence_extrapolation_assign
 Parma_Polyhedra_Library::Grid, 286
 limited_extrapolation_assign
 Parma_Polyhedra_Library::Grid, 288
 limited_generator_extrapolation_assign
 Parma_Polyhedra_Library::Grid, 286
 line
 Parma_Polyhedra_Library::Generator, 249, 251, 254
 Linear_Expression
 Parma_Polyhedra_Library::Linear_Expression, 320, 321
 Linear_Form
 Parma_Polyhedra_Library::Linear_Form, 330
 linear_combine
 Parma_Polyhedra_Library::Linear_Expression, 321
 linear_combine_lax
 Parma_Polyhedra_Library::Linear_Expression, 321
 linear_form_assign
 Parma_Polyhedra_Library::Variable_Floating_Point_Expression, 508
 linear_partition
 Parma_Polyhedra_Library::Pointset_Powerset, 454, 455
 linearize

Parma_Polyhedra_Library::Cast_Floating_Point_Expression, 158
 Parma_Polyhedra_Library::Concrete_Expression, 181
 Parma_Polyhedra_Library::Constant_Floating_Point_Expression, 209
 Parma_Polyhedra_Library::Difference_Floating_Point_Expression, 229
 Parma_Polyhedra_Library::Division_Floating_Point_Expression, 231
 Parma_Polyhedra_Library::Floating_Point_Expression, 239
 Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression, 345
 Parma_Polyhedra_Library::Opposite_Floating_Point_Expression, 387
 Parma_Polyhedra_Library::Sum_Floating_Point_Expression, 501
 Parma_Polyhedra_Library::Variable_Floating_Point_Expression, 508
 lower_bound
 Parma_Polyhedra_Library::Linear_Expression, 321
 MAXIMIZATION
 C++ Language Interface, 69
 MINIMIZATION
 C++ Language Interface, 69
 m_swap
 Parma_Polyhedra_Library::CO_Tree::const_iterator, 199
 Parma_Polyhedra_Library::CO_Tree::iterator, 314
 Parma_Polyhedra_Library::Linear_Expression::const_iterator, 202
 Parma_Polyhedra_Library::Polyhedron, 487
 MIP_Problem
 Parma_Polyhedra_Library::MIP_Problem, 338, 339
 MIP_Problem_Status
 C++ Language Interface, 70
 map_space_dimensions
 Parma_Polyhedra_Library::BD_Shape, 116
 Parma_Polyhedra_Library::Box, 146
 Parma_Polyhedra_Library::Grid, 289
 Parma_Polyhedra_Library::Octagonal_Shape, 379
 Parma_Polyhedra_Library::Partially_Reduced_Product, 410
 Parma_Polyhedra_Library::Pointset_Powerset, 453
 Parma_Polyhedra_Library::Polyhedron, 486
 maximize
 Parma_Polyhedra_Library::BD_Shape, 101
 Parma_Polyhedra_Library::Box, 133
 Parma_Polyhedra_Library::Grid, 275
 Parma_Polyhedra_Library::Octagonal_Shape, 363, 364
 Parma_Polyhedra_Library::Partially_Reduced_Product, 398
 Parma_Polyhedra_Library::Pointset_Powerset, 442
 Parma_Polyhedra_Library::Polyhedron, 470, 471
 memory_size_type
 C++ Language Interface, 67
 minimize
 Parma_Polyhedra_Library::BD_Shape, 102
 Parma_Polyhedra_Library::Box, 134
 Parma_Polyhedra_Library::Grid, 275, 276
 Parma_Polyhedra_Library::Octagonal_Shape, 364
 Parma_Polyhedra_Library::Partially_Reduced_Product, 399
 Parma_Polyhedra_Library::Pointset_Powerset, 443
 Parma_Polyhedra_Library::Polyhedron, 471, 489
 mul_2exp_assign
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::GMP_Integer, 261
 mul_assign
 Parma_Polyhedra_Library::Interval, 311
 mul_linearize
 Parma_Polyhedra_Library::Concrete_Expression, 179
 NONSTRICT_INEQUALITY
 Parma_Polyhedra_Library::Constraint, 215
 NOT_EQUAL
 C++ Language Interface, 68
 ONC_Polyhedron
 Parma_Polyhedra_Library::NNC_Polyhedron, 348–350
 onst_assign
 Parma_Polyhedra_Library::Checked_Number, 170
 Parma_Polyhedra_Library::GMP_Integer, 261
 Parma_Polyhedra_Library::Linear_Expression, 324, 325
 normalize
 Parma_Polyhedra_Library::Congruence, 190
 Parma_Polyhedra_Library::Linear_Expression, 322
 not_equal
 Parma_Polyhedra_Library::Checked_Number, 171
 OPTIMIZED_MIP_PROBLEM
 C++ Language Interface, 70
 OPTIMIZED_PIP_PROBLEM
 C++ Language Interface, 70
 OVERFLOW_IMPOSSIBLE
 C++ Language Interface, 69
 OVERFLOW_UNDEFINED
 C++ Language Interface, 69
 OVERFLOW_WRAPS
 C++ Language Interface, 69
 OK
 Parma_Polyhedra_Library::Grid, 277

Parma_Polyhedra_Library::Polyhedron, 472
 Octagonal_Shape
 Parma_Polyhedra_Library::Octagonal_Shape, 360
 362
 omega_reduce
 Parma_Polyhedra_Library::Powerset, 496
 one_affine_ranking_function_MS
 C++ Language Interface, 72
 one_affine_ranking_function_MS_2
 C++ Language Interface, 73
 one_affine_ranking_function_PR
 C++ Language Interface, 77
 one_affine_ranking_function_PR_2
 C++ Language Interface, 77
 operator<
 Parma_Polyhedra_Library::Checked_Number, 172
 Parma_Polyhedra_Library::Constraint, 217, 219
 operator<<
 Parma_Polyhedra_Library::BD_Shape, 117, 120
 Parma_Polyhedra_Library::Box, 149, 151
 Parma_Polyhedra_Library::Checked_Number, 172, 175
 Parma_Polyhedra_Library::Congruence, 190
 Parma_Polyhedra_Library::Congruence_System, 196
 Parma_Polyhedra_Library::Constraint, 218
 Parma_Polyhedra_Library::Constraint_System, 222
 Parma_Polyhedra_Library::Determinate, 226
 Parma_Polyhedra_Library::Generator, 251, 254
 Parma_Polyhedra_Library::Generator_System, 259
 Parma_Polyhedra_Library::Grid, 290
 Parma_Polyhedra_Library::Grid_Generator, 299
 Parma_Polyhedra_Library::Grid_Generator_System, 304
 Parma_Polyhedra_Library::Linear_Expression, 324, 325
 Parma_Polyhedra_Library::Linear_Form, 333, 335
 Parma_Polyhedra_Library::MIP_Problem, 343
 Parma_Polyhedra_Library::Octagonal_Shape, 382, 385
 Parma_Polyhedra_Library::Partially_Reduced_Product, 411, 412
 Parma_Polyhedra_Library::PIP_Problem, 426
 Parma_Polyhedra_Library::PIP_Tree_Node, 433
 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter, 90
 Parma_Polyhedra_Library::Poly_Con_Relation, 456
 Parma_Polyhedra_Library::Poly_Gen_Relation, 458
 Parma_Polyhedra_Library::Polyhedron, 491
 Parma_Polyhedra_Library::Powerset, 497
 Parma_Polyhedra_Library::Variable, 506
 Parma_Polyhedra_Library::Variables_Set, 510
 operator<=
 Parma_Polyhedra_Library::Checked_Number, 172
 Parma_Polyhedra_Library::Constraint, 217–219
 operator>
 Parma_Polyhedra_Library::Checked_Number, 172
 Parma_Polyhedra_Library::Constraint, 217–219
 operator>>
 Parma_Polyhedra_Library::Checked_Number, 174, 175
 operator>=
 Parma_Polyhedra_Library::Checked_Number, 171
 Parma_Polyhedra_Library::Constraint, 218, 219
 operator*
 Parma_Polyhedra_Library::Linear_Expression, 323–325
 Parma_Polyhedra_Library::Linear_Form, 332, 334
 operator**=
 Parma_Polyhedra_Library::Linear_Expression, 323, 325
 Parma_Polyhedra_Library::Linear_Form, 333, 335
 operator+
 Parma_Polyhedra_Library::Checked_Number, 170, 174
 Parma_Polyhedra_Library::Linear_Expression, 322, 324, 325
 Parma_Polyhedra_Library::Linear_Form, 332–334
 operator++
 Parma_Polyhedra_Library::CO_Tree::const_iterator, 199, 200
 Parma_Polyhedra_Library::CO_Tree::iterator, 314
 Parma_Polyhedra_Library::Linear_Expression::const_iterator, 203
 Parma_Polyhedra_Library::Linear_Expression_-Impl::const_iterator, 201
 Parma_Polyhedra_Library::Linear_Expression_-Interface::const_iterator_interface, 207
 operator+=
 Parma_Polyhedra_Library::Linear_Expression, 323–325
 Parma_Polyhedra_Library::Linear_Form, 333–335
 operator-
 C++ Language Interface, 70
 Parma_Polyhedra_Library::Checked_Number, 170, 174
 Parma_Polyhedra_Library::Linear_Expression, 322–325
 Parma_Polyhedra_Library::Linear_Form, 332, 334
 Parma_Polyhedra_Library::Poly_Con_Relation, 456
 Parma_Polyhedra_Library::Poly_Gen_Relation, 458
 operator--
 Parma_Polyhedra_Library::CO_Tree::const_iterator, 199, 200
 Parma_Polyhedra_Library::CO_Tree::iterator, 314

Parma.Polyhedra.Library::Linear.Expression::const_iterator, 203	Parma.Polyhedra.Library::PIP_Tree_Node::Artificial_Parameter, 90
Parma.Polyhedra.Library::Linear.Expression_-Impl::const_iterator, 201	Parma.Polyhedra.Library::Poly_Con_Relation, 456
Parma.Polyhedra.Library::Linear.Expression_-Interface::const_iterator_interface, 207	Parma.Polyhedra.Library::Poly_Gen_Relation, 457, 458
operator-=	Parma.Polyhedra.Library::Polyhedron, 491
Parma.Polyhedra.Library::Linear.Expression, 323–325	Parma.Polyhedra.Library::Powerset, 497
Parma.Polyhedra.Library::Linear.Form, 333–335	operator%=
operator/	Parma.Polyhedra.Library::Congruence, 190
Parma.Polyhedra.Library::Congruence, 190	operator&
operator/=	C++ Language Interface, 70, 71
Parma.Polyhedra.Library::Congruence, 189	operator&&
Parma.Polyhedra.Library::Linear.Expression, 324, 325	Parma.Polyhedra.Library::Poly_Con_Relation, 456
Parma.Polyhedra.Library::Linear.Form, 333, 335	Parma.Polyhedra.Library::Poly_Gen_Relation, 457, 458
operator=	optimal_value
Parma.Polyhedra.Library::CO_Tree::const_iterator, 199	Parma.Polyhedra.Library::MIP_Problem, 342
Parma.Polyhedra.Library::CO_Tree::iterator, 314	Optimization_Mode
Parma.Polyhedra.Library::Linear.Expression::const_iterator, 202	C++ Language Interface, 69
operator==	optimizing_point
Parma.Polyhedra.Library::BD_Shape, 117, 119	Parma.Polyhedra.Library::MIP_Problem, 342
Parma.Polyhedra.Library::Box, 149	optimizing_solution
Parma.Polyhedra.Library::Checked_Number, 171	Parma.Polyhedra.Library::PIP_Problem, 425
Parma.Polyhedra.Library::CO_Tree::const_iterator, 200	output
Parma.Polyhedra.Library::CO_Tree::iterator, 314	Parma.Polyhedra.Library::Checked_Number, 172, 174
Parma.Polyhedra.Library::Congruence, 190	overapproximate_linear_form
Parma.Polyhedra.Library::Congruence_System, 196	Parma.Polyhedra.Library::Polyhedron, 487
Parma.Polyhedra.Library::Constraint, 217–219	overflows
Parma.Polyhedra.Library::Determinate, 226	Parma.Polyhedra.Library::Floating_Point_Expression, 239
Parma.Polyhedra.Library::Generator, 251, 254	Parma.Polyhedra.Library::Linear_Form, 331
Parma.Polyhedra.Library::Grid, 290	PARAMETER
Parma.Polyhedra.Library::Grid_Generator, 299, 300	Parma.Polyhedra.Library::Grid_Generator, 297
Parma.Polyhedra.Library::Grid_Generator_System, 304	PIVOT_ROW_STRATEGY
Parma.Polyhedra.Library::Linear.Expression::const_iterator, 203	Parma.Polyhedra.Library::PIP_Problem, 423
Parma.Polyhedra.Library::Linear.Expression_-Impl::const_iterator, 201	PIVOT_ROW_STRATEGY_FIRST
Parma.Polyhedra.Library::Linear.Expression_-Interface::const_iterator_interface, 207	Parma.Polyhedra.Library::PIP_Problem, 424
Parma.Polyhedra.Library::Linear.Form, 333, 335	PIVOT_ROW_STRATEGY_MAX.COLUMN
Parma.Polyhedra.Library::MIP_Problem::const_iterator, 205	Parma.Polyhedra.Library::PIP_Problem, 424
Parma.Polyhedra.Library::Octagonal_Shape, 382, 384	POINT
Parma.Polyhedra.Library::Partially_Reduced_Product, 411	Parma.Polyhedra.Library::Generator, 249
	Parma.Polyhedra.Library::Grid_Generator, 297
	POLYNOMIAL_COMPLEXITY
	C++ Language Interface, 69
	PRICING
	Parma.Polyhedra.Library::MIP_Problem, 338
	PRICING_STEEPEST_EDGE.EXACT
	Parma.Polyhedra.Library::MIP_Problem, 338
	PRICING_STEEPEST_EDGE.FLOAT
	Parma.Polyhedra.Library::MIP_Problem, 338

PRICING_TEXTBOOK
 Parma.Polyhedra.Library::MIP_Problem, 338
 PIP_Problem
 Parma.Polyhedra.Library::PIP_Problem, 424
 PIP_Problem_Status
 C++ Language Interface, 70
 PIP_Solution_Node
 Parma.Polyhedra.Library::PIP_Solution_Node, 428
 PPL_VERSION
 C++ Language Interface, 66
 PPL_VERSION_MAJOR
 C++ Language Interface, 66
 PPL_VERSION_MINOR
 C++ Language Interface, 66
 pairwise_apply_assign
 Parma.Polyhedra.Library::Powerset, 496
 pairwise_reduce
 Parma.Polyhedra.Library::Pointset_Powerset, 451
 parameter
 Parma.Polyhedra.Library::Grid_Generator, 298–300
 parametric_values
 Parma.Polyhedra.Library::PIP_Solution_Node, 428
 Parma.Polyhedra.Library
 VC_MINUS_INFINITY, 85
 VC_NAN, 85
 VC_NORMAL, 85
 VC_PLUS_INFINITY, 85
 VR_EMPTY, 86
 VR_EQ, 86
 VR_GE, 86
 VR_GT, 86
 VR_LE, 86
 VR_LGE, 86
 VR_LT, 86
 VR_NE, 86
 Parma.Polyhedra.Library::Constraint
 EQUALITY, 215
 NONSTRICT_INEQUALITY, 215
 STRICT_INEQUALITY, 215
 Parma.Polyhedra.Library::Generator
 CLOSURE_POINT, 249
 LINE, 249
 POINT, 249
 RAY, 249
 Parma.Polyhedra.Library::Grid_Generator
 LINE, 297
 PARAMETER, 297
 POINT, 297
 Parma.Polyhedra.Library::MIP_Problem
 PRICING, 338
 PRICING_STEEPEST_EDGE_EXACT, 338
 PRICING_STEEPEST_EDGE_FLOAT, 338
 PRICING_TEXTBOOK, 338
 Parma.Polyhedra.Library::PIP_Problem
 CUTTING_STRATEGY, 423
 CUTTING_STRATEGY_ALL, 424
 CUTTING_STRATEGY_DEEPEST, 424
 CUTTING_STRATEGY_FIRST, 424
 PIVOT_ROW_STRATEGY, 423
 PIVOT_ROW_STRATEGY_FIRST, 424
 PIVOT_ROW_STRATEGY_MAX_COLUMN, 424
 Parma.Polyhedra.Library, 78
 banner, 86
 Concrete_Expression_BOP, 85
 Concrete_Expression_Kind, 85
 Concrete_Expression_UOP, 85
 fpu_check_inexact, 86
 restore_pre_PPL_rounding, 86
 Result_Class, 85
 Result_Relation, 85
 set_irrational_precision, 86
 set_rounding_for_PPL, 86
 Parma.Polyhedra.Library::Approximable_Reference<Target >, 88
 Parma.Polyhedra.Library::Approximable_Reference_Common<Target >, 88
 Parma.Polyhedra.Library::BD_Shape
 add_congruence, 104
 add_congruences, 105
 add_constraint, 104
 add_constraints, 104
 add_recycled_congruences, 105
 add_recycled_constraints, 104
 affine_form_image, 109
 affine_image, 109
 affine_preimage, 109
 BD_Shape, 99, 100
 BHMZ05_widening_assign, 113
 bounded_affine_image, 111
 bounded_affine_preimage, 111
 bounds_from_above, 101
 bounds_from_below, 101
 CC76_extrapolation_assign, 112, 113
 CC76_narrowing_assign, 113
 concatenate_assign, 115
 constrains, 104
 contains, 103
 difference_assign, 108
 euclidean_distance_assign, 118, 119
 expand_space_dimension, 116
 export_interval_constraints, 107
 fold_space_dimensions, 116
 frequency, 102
 generalized_affine_image, 109, 110

generalized_affine_preimage, 110
 H79_widening_assign, 114
 hash_code, 117
 intersection_assign, 108
 is_disjoint_from, 103
 l_infinity_distance_assign, 118–120
 limited_H79_extrapolation_assign, 114
 map_space_dimensions, 116
 maximize, 101
 minimize, 102
 operator<=, 117, 120
 operator==, 117, 119
 rectilinear_distance_assign, 117–119
 refine_with_congruence, 105
 refine_with_congruences, 106
 refine_with_constraint, 105
 refine_with_constraints, 106
 relation_with, 103
 remove_higher_space_dimensions, 116
 remove_space_dimensions, 115
 simplify_using_context_assign, 108
 strictly_contains, 103
 swap, 117, 120
 time_elapse_assign, 111
 unconstrain, 107
 upper_bound_assign, 108
 wrap_assign, 111
 Parma_Polyhedra_Library::BD_Shape< T >, 90
 Parma_Polyhedra_Library::BHRZ03_Certificate, 120
 compare, 121
 Parma_Polyhedra_Library::BHRZ03_Certificate::Compare, 175
 Parma_Polyhedra_Library::Binary_Operator< Target
 >, 121
 Parma_Polyhedra_Library::Binary_Operator_Common<
 Target >, 121
 Parma_Polyhedra_Library::Box
 add_congruence, 136
 add_congruences, 136
 add_constraint, 135
 add_constraints, 135
 add_recycled_congruences, 136
 add_recycled_constraints, 136
 add_space_dimensions_and_embed, 145
 add_space_dimensions_and_project, 145
 affine_form_image, 140
 affine_image, 139
 affine_preimage, 140
 bounded_affine_image, 141
 bounded_affine_preimage, 142
 bounds_from_above, 133
 bounds_from_below, 133
 Box, 129, 131, 132
 CC76_narrowing_assign, 144
 CC76_widening_assign, 143, 144
 concatenate_assign, 145
 constrains, 132
 contains, 135
 difference_assign, 139
 drop_some_non_integer_points, 143
 euclidean_distance_assign, 150, 151
 expand_space_dimension, 146
 fold_space_dimensions, 146
 frequency, 134
 generalized_affine_image, 140, 141
 generalized_affine_preimage, 140, 141
 get_interval, 148
 has_lower_bound, 148
 has_upper_bound, 148
 hash_code, 149
 intersection_assign, 139
 is_disjoint_from, 135
 l_infinity_distance_assign, 150, 151
 limited_CC76_extrapolation_assign, 144
 map_space_dimensions, 146
 maximize, 133
 minimize, 134
 operator<=, 149, 151
 operator==, 149
 propagate_constraint, 138
 propagate_constraints, 138
 rectilinear_distance_assign, 149, 151
 refine_with_congruence, 137
 refine_with_congruences, 137
 refine_with_constraint, 137
 refine_with_constraints, 137
 relation_with, 132, 133
 remove_higher_space_dimensions, 146
 remove_space_dimensions, 145
 set_interval, 148
 simplify_using_context_assign, 139
 strictly_contains, 135
 swap, 149, 151
 time_elapse_assign, 142
 unconstrain, 138
 upper_bound_assign, 139
 upper_bound_assign_if_exact, 139
 wrap_assign, 142
 Parma_Polyhedra_Library::Box< Interval >, 122
 Parma_Polyhedra_Library::C_Polyhedron, 152
 C_Polyhedron, 153–156
 poly_hull_assign_if_exact, 156
 positive_time_elapse_assign, 156
 Parma_Polyhedra_Library::CO_Tree::const_iterator, 197
 const_iterator, 198, 199
 index, 200
 m_swap, 199
 operator++, 199, 200

- operator--, 199, 200
- operator=, 199
- operator==, 200
- Parma_Polyhedra_Library::CO_Tree::iterator, 312
- index, 314
- iterator, 313
- m_swap, 314
- operator++, 314
- operator--, 314
- operator=, 314
- operator==, 314
- Parma_Polyhedra_Library::Cast_Floating_Point_Expression trunc_assign, 170
- linearize, 158
- swap, 159
- Parma_Polyhedra_Library::Cast_Operator< Target >, 159
- Parma_Polyhedra_Library::Cast_Operator_Common< Target >, 159
- Parma_Polyhedra_Library::Checked_Number
 - abs_assign, 170
 - add_mul_assign, 170
 - ascii_dump, 172
 - ascii_load, 174
 - assign_r, 169, 174
 - ceil_assign, 170
 - classify, 169
 - cmp, 174
 - construct, 169
 - div_2exp_assign, 171
 - equal, 171
 - exact_div_assign, 171
 - external_memory_in_bytes, 170, 174
 - floor_assign, 170
 - gcd_assign, 171
 - gcdext_assign, 171
 - greater_or_equal, 171
 - greater_than, 172
 - infinity_sign, 169
 - input, 172, 175
 - is_integer, 169
 - is_minus_infinity, 169
 - is_not_a_number, 169
 - is_plus_infinity, 169
 - lcm_assign, 171
 - less_or_equal, 172
 - less_than, 172
 - mul_2exp_assign, 171
 - neg_assign, 170
 - not_equal, 171
 - operator<, 172
 - operator<=, 172, 175
 - operator<=, 172
 - operator>, 172
 - operator>=, 174, 175
 - operator>=, 171
 - operator+, 170, 174
 - operator-, 170, 174
 - operator==, 171
 - output, 172, 174
 - raw_value, 174
 - sgn, 174
 - sqrt_assign, 171
 - sub_mul_assign, 170
 - swap, 174, 175
 - total_memory_in_bytes, 169, 174
- Parma_Polyhedra_Library::Checked_Number< T, Policy >, 160
- Parma_Polyhedra_Library::Concrete_Expression
 - add_linearize, 177
 - cast_linearize, 181
 - div_linearize, 180
 - linearize, 181
 - mul_linearize, 179
 - sub_linearize, 178
- Parma_Polyhedra_Library::Concrete_Expression< Target >, 176
- Parma_Polyhedra_Library::Concrete_Expression_Common< Target >, 182
- Parma_Polyhedra_Library::Concrete_Expression_Type, 182
 - floating_point_format, 183
- Parma_Polyhedra_Library::Congruence, 184
 - coefficient, 189
 - Congruence, 188
 - default_representation, 191
 - is_equality, 189
 - is_inconsistent, 189
 - is_proper_congruence, 189
 - is_tautological, 189
 - normalize, 190
 - operator<=, 190
 - operator/, 190
 - operator=, 189
 - operator==, 190
 - operator%=, 190
 - set_modulus, 189
 - set_space_dimension, 189
 - shift_space_dimensions, 189
 - sign_normalize, 189
 - strong_normalize, 190
 - swap, 190, 191
- Parma_Polyhedra_Library::Congruence_System, 191
 - Congruence_System, 194
 - insert, 194, 196
 - operator<=, 196
 - operator==, 196
 - permute_space_dimensions, 196

set_space_dimension, 196
 swap, 196
 Parma_Polyhedra_Library::Congruence_System::const_iterator, 203
 Parma_Polyhedra_Library::Congruences_Reduction product_reduce, 197
 Parma_Polyhedra_Library::Congruences_Reduction< D1, D2 >, 196
 Parma_Polyhedra_Library::Constant_Floating_Point_Expression
 linearize, 209
 swap, 210
 Parma_Polyhedra_Library::Constraint, 210
 coefficient, 216
 Constraint, 215, 216
 default_representation, 219
 is_equal_to, 217
 is_equivalent_to, 216
 is_inconsistent, 216
 is_tautological, 216
 operator<, 217, 219
 operator<=, 218
 operator<=, 217–219
 operator>, 217–219
 operator>=, 218, 219
 operator==, 217–219
 remove_space_dimensions, 216
 set_space_dimension, 216
 shift_space_dimensions, 216
 swap, 218, 219
 Type, 215
 Parma_Polyhedra_Library::Constraint_System, 219
 Constraint_System, 222
 operator<=, 222
 swap, 222
 Parma_Polyhedra_Library::Constraint_System_const_iterator, 222
 Parma_Polyhedra_Library::Constraints_Reduction product_reduce, 224
 Parma_Polyhedra_Library::Constraints_Reduction< D1, D2 >, 223
 Parma_Polyhedra_Library::Determinate
 has_nontrivial_weakening, 226
 operator<=, 226
 operator==, 226
 swap, 226
 Parma_Polyhedra_Library::Determinate< PSET >, 224
 Parma_Polyhedra_Library::Difference_Floating_Point_Expression
 linearize, 229
 swap, 229
 Parma_Polyhedra_Library::Division_Floating_Point_Expression
 linearize, 231
 swap, 232
 Parma_Polyhedra_Library::Domain_Product< D1, D2 >, 232
 Parma_Polyhedra_Library::FP_Oracle
 get_associated_dimensions, 242
 get_fp_constant_value, 241
 get_integer_expr_value, 242
 get_interval, 241
 Parma_Polyhedra_Library::FP_Oracle< Target, FP_Interval_Type >, 241
 Parma_Polyhedra_Library::Floating_Point_Constant< Target >, 236
 Parma_Polyhedra_Library::Floating_Point_Constant_Common< Target >, 236
 Parma_Polyhedra_Library::Floating_Point_Expression
 absolute_error, 240
 intervalize, 240
 linearize, 239
 overflows, 239
 relative_error, 240
 Parma_Polyhedra_Library::GMP_Integer, 260
 abs_assign, 261
 add_mul_assign, 261
 div_2exp_assign, 262
 exact_div_assign, 262
 gcd_assign, 261
 gcdext_assign, 261
 lcm_assign, 261
 mul_2exp_assign, 261
 neg_assign, 261
 raw_value, 261
 rem_assign, 261
 sqrt_assign, 262
 sub_mul_assign, 261
 Parma_Polyhedra_Library::Generator, 242
 closure_point, 250, 251, 254
 coefficient, 251
 default_representation, 255
 divisor, 251
 euclidean_distance_assign, 252–255
 Generator, 249
 is_equal_to, 251
 is_equivalent_to, 251
 l_infinity_distance_assign, 253, 255
 line, 249, 251, 254
 operator<=, 251, 254
 operator==, 251, 254
 permute_space_dimensions, 250
 point, 250, 251, 254
 ray, 250, 251, 254
 rectilinear_distance_assign, 251, 252, 254
 remove_space_dimensions, 250
 set_space_dimension, 250

- shift_space_dimensions, 250
 - swap, 251, 255
 - Type, 249
- Parma_Polyhedra_Library::Generator_System, 255
 - ascii_load, 258
 - Generator_System, 258
 - operator<=, 259
 - swap, 259
- Parma_Polyhedra_Library::Generator_System_const_iterator, 259
- Parma_Polyhedra_Library::Grid, 262
 - add_congruence, 277
 - add_congruences, 277
 - add_constraint, 278
 - add_constraints, 278
 - add_grid_generator, 277
 - add_grid_generators, 279
 - add_recycled_congruences, 277
 - add_recycled_constraints, 278
 - add_recycled_grid_generators, 279
 - add_space_dimensions_and_embed, 288
 - add_space_dimensions_and_project, 288
 - affine_image, 282
 - affine_preimage, 282
 - bounded_affine_image, 284
 - bounded_affine_preimage, 284
 - bounds_from_above, 274
 - bounds_from_below, 275
 - concatenate_assign, 289
 - congruence_widening_assign, 285
 - constrains, 274
 - contains, 276
 - difference_assign, 280
 - drop_some_non_integer_points, 285
 - expand_space_dimension, 290
 - fold_space_dimensions, 290
 - frequency, 276
 - generalized_affine_image, 282, 283
 - generalized_affine_preimage, 283
 - generator_widening_assign, 286
 - Grid, 271–274
 - hash_code, 290
 - intersection_assign, 280
 - is_discrete, 274
 - is_disjoint_from, 274
 - is_topologically_closed, 274
 - limited_congruence_extrapolation_assign, 286
 - limited_extrapolation_assign, 288
 - limited_generator_extrapolation_assign, 286
 - map_space_dimensions, 289
 - maximize, 275
 - minimize, 275, 276
 - OK, 277
 - operator<=, 290
 - operator==, 290
 - refine_with_congruence, 278
 - refine_with_congruences, 279
 - refine_with_constraint, 279
 - refine_with_constraints, 279
 - remove_higher_space_dimensions, 289
 - remove_space_dimensions, 289
 - simplify_using_context_assign, 282
 - strictly_contains, 277
 - swap, 290, 291
 - time_elapse_assign, 284
 - unconstrain, 280
 - upper_bound_assign, 280
 - upper_bound_assign_if_exact, 280
 - widening_assign, 286
 - wrap_assign, 285
- Parma_Polyhedra_Library::Grid_Certificate, 291
 - compare, 292
- Parma_Polyhedra_Library::Grid_Certificate::Compare, 176
- Parma_Polyhedra_Library::Grid_Generator, 292
 - coefficient, 298
 - default_representation, 300
 - divisor, 299
 - Grid_Generator, 298
 - grid_line, 298, 300
 - grid_point, 298–300
 - is_equal_to, 299
 - is_equivalent_to, 299
 - operator<=, 299
 - operator==, 299, 300
 - parameter, 298–300
 - remove_space_dimensions, 298
 - scale_to_divisor, 299
 - set_divisor, 299
 - set_space_dimension, 298
 - shift_space_dimensions, 298
 - swap, 299, 300
 - Type, 297
- Parma_Polyhedra_Library::Grid_Generator_System, 300
 - ascii_load, 304
 - insert, 304
 - operator<=, 304
 - operator==, 304
 - swap, 304
- Parma_Polyhedra_Library::Grid_Generator_System::const_iterator, 205
- Parma_Polyhedra_Library::H79_Certificate, 304
 - compare, 305
- Parma_Polyhedra_Library::H79_Certificate::Compare, 176
- Parma_Polyhedra_Library::IO_Operators, 86
 - wrap_string, 87

Parma_Polyhedra_Library::Implementation::Doubly-Linked_Object, 232
 Parma_Polyhedra_Library::Implementation::EList< T Parma_Polyhedra_Library::Linear_Expression::const_iterator, 201
 Parma_Polyhedra_Library::Implementation::EList_Iterator<const_iterator, 202
 T >, 235
 Parma_Polyhedra_Library::Implementation::Watchdog::Handler, 305
 Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Flag< Flag_Base, Flag >, 306
 Parma_Polyhedra_Library::Implementation::Watchdog::Handler_Function, 307
 Parma_Polyhedra_Library::Implementation::WatchdogParma_Polyhedra_Library::Linear_Expression_Impl<::Pending_Element< Threshold >, 412
 Parma_Polyhedra_Library::Implementation::WatchdogParma_Polyhedra_Library::Linear_Expression_Impl::Pending_List< Traits >, 413
 Parma_Polyhedra_Library::Implementation::Watchdog::Time, 503
 Parma_Polyhedra_Library::Integer_Constant< Target >, 308
 Parma_Polyhedra_Library::Integer_Constant_Common< Target >, 308
 Parma_Polyhedra_Library::Interval
 div_assign, 311
 mul_assign, 311
 refine_existential, 311
 refine_universal, 311
 simplify_using_context_assign, 311
 swap, 312
 Parma_Polyhedra_Library::Interval< Boundary, Info >, 309
 Parma_Polyhedra_Library::Linear_Expression, 315
 add_mul_assign, 324, 325
 begin, 321
 end, 321
 is_equal_to, 322
 Linear_Expression, 320, 321
 linear_combine, 321
 linear_combine_lax, 321
 lower_bound, 321
 neg_assign, 324, 325
 normalize, 322
 operator<=, 324, 325
 operator*, 323–325
 operator*=, 323, 325
 operator+, 322, 324, 325
 operator+=, 323–325
 operator-, 322–325
 operator-=, 323–325
 operator/=: 324, 325
 permute_space_dimensions, 322
 remove_space_dimensions, 321
 shift_space_dimensions, 321
 sign_normalize, 322
 sub_mul_assign, 324, 325
 swap, 324, 325
 operator++, 203
 operator--, 203
 operator=, 202
 operator==, 203
 swap, 203
 variable, 203
 Row >::const_iterator, 200
 ::const_iterator
 clone, 201
 operator++, 201
 operator--, 201
 operator==, 201
 variable, 201
 Parma_Polyhedra_Library::Linear_Expression_Interface::const_iterator_interface, 206
 Parma_Polyhedra_Library::Linear_Form
 affine_form_image, 331
 discard_occurrences, 331
 intervalize, 331
 Linear_Form, 330
 operator<=, 333, 335
 operator*, 332, 334
 operator*=, 333, 335
 operator+, 332–334
 operator+=, 333–335
 operator-, 332, 334
 operator-=, 333–335
 operator/=: 333, 335
 operator==, 333, 335
 overflows, 331
 relative_error, 331
 swap, 332, 334
 upper_bound_assign, 332
 Parma_Polyhedra_Library::Linear_Form< C >, 326
 Parma_Polyhedra_Library::MIP_Problem, 335
 add_constraint, 340
 add_constraints, 340
 clear, 340
 Control_Parameter_Name, 338
 Control_Parameter_Value, 338
 evaluate_objective_function, 342
 feasible_point, 342
 is_satisfiable, 342
 MIP_Problem, 338, 339
 operator<=, 343
 optimal_value, 342

- optimizing_point, 342
- set_objective_function, 340
- solve, 342
- swap, 343
- Parma_Polyhedra_Library::MIP_Problem::const_iterator, 204
- operator==, 205
- Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression
 - linearize, 345
 - swap, 346
- Parma_Polyhedra_Library::NNC_Polyhedron, 346
 - NNC_Polyhedron, 348–350
 - positive_time_elapse_assign, 350
- Parma_Polyhedra_Library::No_Reduction
 - product_reduce, 351
- Parma_Polyhedra_Library::No_Reduction< D1, D2 >, 351
- Parma_Polyhedra_Library::Octagonal_Shape
 - add_congruence, 366
 - add_congruences, 366
 - add_constraint, 365
 - add_constraints, 365
 - add_recycled_congruences, 366
 - add_recycled_constraints, 365
 - add_space_dimensions_and_embed, 377
 - add_space_dimensions_and_project, 377
 - affine_form_image, 372
 - affine_image, 370
 - affine_preimage, 372
 - BHMZ05_widening_assign, 376
 - bounded_affine_image, 373
 - bounded_affine_preimage, 374
 - bounds_from_above, 363
 - bounds_from_below, 363
 - CC76_extrapolation_assign, 375, 376
 - CC76_narrowing_assign, 376
 - concatenate_assign, 377
 - constrains, 363
 - contains, 362
 - difference_assign, 370
 - drop_some_non_integer_points, 375
 - euclidean_distance_assign, 383, 384
 - expand_space_dimension, 379
 - export_interval_constraints, 380
 - fold_space_dimensions, 380
 - frequency, 365
 - generalized_affine_image, 372, 373
 - generalized_affine_preimage, 373, 374
 - hash_code, 382
 - intersection_assign, 369
 - is_disjoint_from, 362
 - l_infinity_distance_assign, 383, 384
 - limited_CC76_extrapolation_assign, 377
 - map_space_dimensions, 379
 - maximize, 363, 364
 - minimize, 364
 - Octagonal_Shape, 360–362
 - operator<<, 382, 385
 - operator==, 382, 384
 - rectilinear_distance_assign, 382, 384
 - refine_fp_interval_abstract_store, 380
 - refine_with_congruence, 368
 - refine_with_congruences, 368
 - refine_with_constraint, 366
 - refine_with_constraints, 368
 - refine_with_linear_form_inequality, 368
 - relation_with, 362, 363
 - remove_higher_space_dimensions, 379
 - remove_space_dimensions, 379
 - simplify_using_context_assign, 370
 - strictly_contains, 362
 - swap, 382, 385
 - time_elapse_assign, 374
 - unconstrain, 369
 - upper_bound_assign, 370
 - upper_bound_assign_if_exact, 370
 - wrap_assign, 374
- Parma_Polyhedra_Library::Octagonal_Shape< T >, 351
- Parma_Polyhedra_Library::Opposite_Floating_Point_Expression
 - linearize, 387
 - swap, 387
- Parma_Polyhedra_Library::PIP_Decision_Node, 414
- Parma_Polyhedra_Library::PIP_Problem, 415
 - add_constraint, 425
 - add_constraints, 425
 - clear, 424
 - Control_Parameter_Name, 423
 - Control_Parameter_Value, 423
 - get_big_parameter_dimension, 426
 - is_satisfiable, 425
 - operator<<, 426
 - optimizing_solution, 425
 - PIP_Problem, 424
 - print_solution, 426
 - solution, 425
 - solve, 425
 - swap, 426
- Parma_Polyhedra_Library::PIP_Solution_Node, 426
 - generate_cut, 428
 - parametric_values, 428
 - update_solution, 428
- Parma_Polyhedra_Library::PIP_Tree_Node, 429
 - compatibility_check, 433
 - constraints, 431
 - operator<<, 433

- print, [431](#)
- print_tree, [432](#)
- solve, [432](#)
- update_tableau, [432](#)
- Parma_Polyhedra_Library::Partially_Reduced_Product
 - add_congruence, [400](#)
 - add_congruences, [401](#)
 - add_constraint, [400](#)
 - add_constraints, [401](#)
 - add_recycled_congruences, [401](#)
 - add_recycled_constraints, [402](#)
 - affine_image, [403](#)
 - affine_preimage, [403](#)
 - bounded_affine_image, [407](#)
 - bounded_affine_preimage, [407](#)
 - bounds_from_above, [398](#)
 - bounds_from_below, [398](#)
 - concatenate_assign, [409](#)
 - constrains, [398](#)
 - contains, [400](#)
 - difference_assign, [403](#)
 - expand_space_dimension, [410](#)
 - fold_space_dimensions, [411](#)
 - generalized_affine_image, [405](#)
 - generalized_affine_preimage, [405](#), [407](#)
 - hash_code, [411](#)
 - intersection_assign, [402](#)
 - is_disjoint_from, [398](#)
 - map_space_dimensions, [410](#)
 - maximize, [398](#)
 - minimize, [399](#)
 - operator<=, [411](#), [412](#)
 - operator==, [411](#)
 - refine_with_congruence, [400](#)
 - refine_with_congruences, [401](#)
 - refine_with_constraint, [400](#)
 - refine_with_constraints, [402](#)
 - remove_space_dimensions, [409](#)
 - strictly_contains, [400](#)
 - swap, [411](#), [412](#)
 - time_elapse_assign, [408](#)
 - unconstrain, [402](#)
 - upper_bound_assign, [403](#)
 - widening_assign, [408](#)
- Parma_Polyhedra_Library::Partially_Reduced_Product
 - D1, D2, R >, [387](#)
- Parma_Polyhedra_Library::Pointset_Powerset
 - add_congruence, [446](#)
 - add_congruences, [446](#)
 - add_constraint, [445](#)
 - add_constraints, [445](#)
 - add_disjunct, [445](#)
 - affine_image, [448](#)
 - affine_preimage, [448](#)
 - approximate_partition, [454](#)
 - BGP99_extrapolation_assign, [451](#)
 - BHZ03_widening_assign, [452](#)
 - bounded_affine_image, [450](#)
 - bounded_affine_preimage, [450](#)
 - bounds_from_above, [442](#)
 - bounds_from_below, [442](#)
 - check_containment, [454](#)
 - concatenate_assign, [452](#)
 - constrains, [442](#)
 - contains, [444](#)
 - difference_assign, [447](#)
 - drop_some_non_integer_points, [447](#)
 - expand_space_dimension, [453](#)
 - fold_space_dimensions, [453](#)
 - generalized_affine_image, [448](#), [449](#)
 - generalized_affine_preimage, [449](#)
 - geometrically_covers, [443](#)
 - geometrically_equals, [444](#)
 - hash_code, [445](#)
 - intersection_assign, [447](#)
 - is_disjoint_from, [441](#)
 - linear_partition, [454](#), [455](#)
 - map_space_dimensions, [453](#)
 - maximize, [442](#)
 - minimize, [443](#)
 - pairwise_reduce, [451](#)
 - Pointset_Powerset, [439](#)–[441](#)
 - refine_with_congruence, [446](#)
 - refine_with_congruences, [446](#)
 - refine_with_constraint, [445](#)
 - refine_with_constraints, [445](#)
 - relation_with, [444](#), [445](#)
 - remove_higher_space_dimensions, [452](#)
 - remove_space_dimensions, [452](#)
 - simplify_using_context_assign, [448](#)
 - strictly_contains, [444](#)
 - swap, [454](#), [455](#)
 - time_elapse_assign, [450](#)
 - unconstrain, [446](#), [447](#)
 - widen_fun_ref, [453](#), [454](#)
 - wrap_assign, [450](#)
- Parma_Polyhedra_Library::Pointset_Powerset < PS-ET >, [433](#)
- Parma_Polyhedra_Library::Poly_Con_Relation, [455](#)
 - operator<=, [456](#)
 - operator-, [456](#)
 - operator==, [456](#)
 - operator&&, [456](#)
- Parma_Polyhedra_Library::Poly_Gen_Relation, [456](#)
 - operator<=, [458](#)
 - operator-, [458](#)
 - operator==, [457](#), [458](#)
 - operator&&, [457](#), [458](#)

Parma_Polyhedra_Library::Polyhedron, 458
 add_and_minimize, 489, 490
 add_congruence, 473
 add_congruences, 474
 add_constraint, 472
 add_constraints, 473
 add_generator, 473
 add_generators, 473
 add_recycled_congruences, 474
 add_recycled_constraints, 473
 add_recycled_generators, 474
 add_space_dimensions_and_embed, 484
 add_space_dimensions_and_project, 484
 affine_form_image, 478
 affine_image, 478
 affine_preimage, 478
 BHRZ03_widening_assign, 482
 bounded_BHRZ03_extrapolation_assign, 483
 bounded_H79_extrapolation_assign, 484
 bounded_affine_image, 480
 bounded_affine_preimage, 480
 bounds_from_above, 470
 bounds_from_below, 470
 concatenate_assign, 484
 constrains, 470
 contains, 472
 conversion, 489
 convert_to_integer_expression, 488
 convert_to_integer_expressions, 488
 drop_some_non_integer_points, 482, 487
 expand_space_dimension, 486
 fold_space_dimensions, 487
 frequency, 472
 generalized_affine_image, 479, 480
 generalized_affine_preimage, 479, 480
 generalized_refine_with_linear_form_inequality, 476
 H79_widening_assign, 483
 hash_code, 487
 intersection_assign, 477
 is_disjoint_from, 470
 limited_BHRZ03_extrapolation_assign, 482
 limited_H79_extrapolation_assign, 483
 m_swap, 487
 map_space_dimensions, 486
 maximize, 470, 471
 minimize, 471, 489
 OK, 472
 operator<=, 491
 operator==, 491
 overapproximate_linear_form, 487
 poly_difference_assign, 478
 poly_hull_assign, 477
 Polyhedron, 468, 469
 positive_time_elapse_assign, 481
 positive_time_elapse_assign_impl, 488
 refine_fp_interval_abstract_store, 477
 refine_with_congruence, 476
 refine_with_congruences, 476
 refine_with_constraint, 474
 refine_with_constraints, 476
 refine_with_linear_form_inequality, 476
 relation_with, 469, 470
 remove_higher_space_dimensions, 486
 remove_space_dimensions, 486
 simplify, 490
 simplify_using_context_assign, 478
 strictly_contains, 472
 swap, 491
 time_elapse_assign, 481
 unconstrain, 477
 wrap_assign, 481
 Parma_Polyhedra_Library::Powerset
 add_non_bottom_disjunct_preserve_reduction, 496
 iterator, 496
 omega_reduce, 496
 operator<=, 497
 operator==, 497
 pairwise_apply_assign, 496
 Sequence, 496
 swap, 497
 upper_bound_assign, 496
 upper_bound_assign_if_exact, 496
 Parma_Polyhedra_Library::Powerset< D >, 491
 Parma_Polyhedra_Library::Recycle_Input, 497
 Parma_Polyhedra_Library::Shape_Preserving_Reduction
 product_reduce, 498
 Parma_Polyhedra_Library::Shape_Preserving_Reduction<
 D1, D2 >, 498
 Parma_Polyhedra_Library::Smash_Reduction
 product_reduce, 499
 Parma_Polyhedra_Library::Smash_Reduction< D1, D2
 >, 499
 Parma_Polyhedra_Library::Sum_Floating_Point_Expression
 linearize, 501
 swap, 502
 Parma_Polyhedra_Library::Threshold_Watcher< Traits
 >, 502
 Parma_Polyhedra_Library::Throwable, 502
 Parma_Polyhedra_Library::Unary_Operator< Target
 >, 503
 Parma_Polyhedra_Library::Unary_Operator_Common<
 Target >, 504
 Parma_Polyhedra_Library::Variable, 504
 less, 506
 operator<=, 506
 space_dimension, 506
 swap, 506

Variable, 505
 Parma_Polyhedra_Library::Variable::Compare, 175
 Parma_Polyhedra_Library::Variable_Floating_Point_Expression
 linearize, 508
 swap, 508
 Parma_Polyhedra_Library::Variables_Set, 509
 operator<=, 510
 Variables_Set, 510
 Parma_Polyhedra_Library::Watchdog, 510
 Partially_Reduced_Product
 Parma_Polyhedra_Library::Partially_Reduced_Product, 394–397
 permute_space_dimensions
 Parma_Polyhedra_Library::Congruence_System, 196
 Parma_Polyhedra_Library::Generator, 250
 Parma_Polyhedra_Library::Linear_Expression, 328
 point
 Parma_Polyhedra_Library::Generator, 250, 251, 254
 Pointset_Powerset
 Parma_Polyhedra_Library::Pointset_Powerset, 439–441
 poly_difference_assign
 Parma_Polyhedra_Library::Polyhedron, 478
 poly_hull_assign
 Parma_Polyhedra_Library::Polyhedron, 477
 poly_hull_assign_if_exact
 Parma_Polyhedra_Library::C_Polyhedron, 156
 Parma_Polyhedra_Library::NNC_Polyhedron, 350
 Polyhedron
 Parma_Polyhedra_Library::Polyhedron, 468, 469
 positive_time_elapse_assign
 Parma_Polyhedra_Library::C_Polyhedron, 156
 Parma_Polyhedra_Library::NNC_Polyhedron, 350
 Parma_Polyhedra_Library::Polyhedron, 481
 positive_time_elapse_assign_impl
 Parma_Polyhedra_Library::Polyhedron, 488
 print
 Parma_Polyhedra_Library::PIP_Tree_Node, 431
 print_solution
 Parma_Polyhedra_Library::PIP_Problem, 426
 print_tree
 Parma_Polyhedra_Library::PIP_Tree_Node, 432
 product_reduce
 Parma_Polyhedra_Library::Congruences_Reduction, 197
 Parma_Polyhedra_Library::Constraints_Reduction, 224
 Parma_Polyhedra_Library::No_Reduction, 351
 Parma_Polyhedra_Library::Shape_Preserving_Reduction, 498
 Parma_Polyhedra_Library::Smash_Reduction, 499
 propagate_constraint
 Parma_Polyhedra_Library::Box, 138
 propagate_constraints
 Parma_Polyhedra_Library::Box, 138
 RAY
 Parma_Polyhedra_Library::Generator, 249
 ROUND_DOWN
 C++ Language Interface, 68
 ROUND_IGNORE
 C++ Language Interface, 68
 ROUND_NOT_NEEDED
 C++ Language Interface, 68
 ROUND_STRICT_RELATION
 C++ Language Interface, 68
 ROUND_UP
 C++ Language Interface, 68
 raw_value
 Parma_Polyhedra_Library::Checked_Number, 174
 Parma_Polyhedra_Library::GMP_Integer, 261
 ray
 Parma_Polyhedra_Library::Generator, 250, 251, 254
 rectilinear_distance_assign
 Parma_Polyhedra_Library::BD_Shape, 117–119
 Parma_Polyhedra_Library::Box, 149, 151
 Parma_Polyhedra_Library::Generator, 251, 252, 254
 Parma_Polyhedra_Library::Octagonal_Shape, 382, 384
 refine_existential
 Parma_Polyhedra_Library::Interval, 311
 refine_fp_interval_abstract_store
 Parma_Polyhedra_Library::BD_Shape, 117
 Parma_Polyhedra_Library::Octagonal_Shape, 380
 Parma_Polyhedra_Library::Polyhedron, 477
 refine_universal
 Parma_Polyhedra_Library::Interval, 311
 refine_with_congruence
 Parma_Polyhedra_Library::BD_Shape, 105
 Parma_Polyhedra_Library::Box, 137
 Parma_Polyhedra_Library::Grid, 278
 Parma_Polyhedra_Library::Octagonal_Shape, 368
 Parma_Polyhedra_Library::Partially_Reduced_Product, 400
 Parma_Polyhedra_Library::Pointset_Powerset, 446
 Parma_Polyhedra_Library::Polyhedron, 476
 refine_with_congruences
 Parma_Polyhedra_Library::BD_Shape, 106
 Parma_Polyhedra_Library::Box, 137
 Parma_Polyhedra_Library::Grid, 279
 Parma_Polyhedra_Library::Octagonal_Shape, 368
 Parma_Polyhedra_Library::Partially_Reduced_Product, 401

Parma_Polyhedra_Library::Pointset_Powerset, 446
 Parma_Polyhedra_Library::Polyhedron, 476
 refine_with_constraint
 Parma_Polyhedra_Library::BD_Shape, 105
 Parma_Polyhedra_Library::Box, 137
 Parma_Polyhedra_Library::Grid, 279
 Parma_Polyhedra_Library::Octagonal_Shape, 366
 Parma_Polyhedra_Library::Partially_Reduced_Product, 400
 Parma_Polyhedra_Library::Pointset_Powerset, 445
 Parma_Polyhedra_Library::Polyhedron, 474
 refine_with_constraints
 Parma_Polyhedra_Library::BD_Shape, 106
 Parma_Polyhedra_Library::Box, 137
 Parma_Polyhedra_Library::Grid, 279
 Parma_Polyhedra_Library::Octagonal_Shape, 368
 Parma_Polyhedra_Library::Partially_Reduced_Product, 402
 Parma_Polyhedra_Library::Pointset_Powerset, 445
 Parma_Polyhedra_Library::Polyhedron, 476
 refine_with_linear_form_inequality
 Parma_Polyhedra_Library::BD_Shape, 106
 Parma_Polyhedra_Library::Octagonal_Shape, 368
 Parma_Polyhedra_Library::Polyhedron, 476
 Relation_Symbol
 C++ Language Interface, 68
 relation_with
 Parma_Polyhedra_Library::BD_Shape, 103
 Parma_Polyhedra_Library::Box, 132, 133
 Parma_Polyhedra_Library::Octagonal_Shape, 362, 363
 Parma_Polyhedra_Library::Pointset_Powerset, 444, 445
 Parma_Polyhedra_Library::Polyhedron, 469, 470
 relative_error
 Parma_Polyhedra_Library::Floating_Point_Expression, 240
 Parma_Polyhedra_Library::Linear_Form, 331
 rem_assign
 Parma_Polyhedra_Library::GMP_Integer, 261
 remove_higher_space_dimensions
 Parma_Polyhedra_Library::BD_Shape, 116
 Parma_Polyhedra_Library::Box, 146
 Parma_Polyhedra_Library::Grid, 289
 Parma_Polyhedra_Library::Octagonal_Shape, 379
 Parma_Polyhedra_Library::Partially_Reduced_Product, 410
 Parma_Polyhedra_Library::Pointset_Powerset, 452
 Parma_Polyhedra_Library::Polyhedron, 486
 remove_space_dimensions
 Parma_Polyhedra_Library::BD_Shape, 115
 Parma_Polyhedra_Library::Box, 145
 Parma_Polyhedra_Library::Constraint, 216
 Parma_Polyhedra_Library::Generator, 250
 Parma_Polyhedra_Library::Grid, 289
 Parma_Polyhedra_Library::Grid_Generator, 298
 Parma_Polyhedra_Library::Linear_Expression, 321
 Parma_Polyhedra_Library::Octagonal_Shape, 379
 Parma_Polyhedra_Library::Partially_Reduced_Product, 409
 Parma_Polyhedra_Library::Pointset_Powerset, 452
 Parma_Polyhedra_Library::Polyhedron, 486
 Representation
 C++ Language Interface, 69
 restore_pre_PPL_rounding
 Parma_Polyhedra_Library, 86
 Result
 C++ Language Interface, 67
 Result_Class
 Parma_Polyhedra_Library, 85
 Result_Relation
 Parma_Polyhedra_Library, 85
 result_class
 C++ Language Interface, 70
 result_relation
 C++ Language Interface, 71
 result_relation_class
 C++ Language Interface, 71
 round_dir
 C++ Language Interface, 71
 round_direct
 C++ Language Interface, 71
 round_down
 C++ Language Interface, 71
 round_fpu_dir
 C++ Language Interface, 71
 round_ignore
 C++ Language Interface, 71
 round_inverse
 C++ Language Interface, 71
 round_not_needed
 C++ Language Interface, 71
 round_not_requested
 C++ Language Interface, 71
 round_strict_relation
 C++ Language Interface, 71
 round_up
 C++ Language Interface, 71
 Rounding_Dir
 C++ Language Interface, 68
 SIGNED_2_COMPLEMENT
 C++ Language Interface, 69
 SIMPLEX_COMPLEXITY
 C++ Language Interface, 69
 SPARSE
 C++ Language Interface, 70
 STRICT_INEQUALITY

Parma.Polyhedra.Library::Constraint, 215
 scale_to_divisor
 Parma.Polyhedra.Library::Grid_Generator, 299
 Sequence
 Parma.Polyhedra.Library::Powerset, 496
 set_divisor
 Parma.Polyhedra.Library::Grid_Generator, 299
 set_interval
 Parma.Polyhedra.Library::Box, 148
 set_irrational_precision
 Parma.Polyhedra.Library, 86
 set_modulus
 Parma.Polyhedra.Library::Congruence, 189
 set_objective_function
 Parma.Polyhedra.Library::MIP_Problem, 340
 set_rounding_for_PPL
 Parma.Polyhedra.Library, 86
 set_space_dimension
 Parma.Polyhedra.Library::Congruence, 189
 Parma.Polyhedra.Library::Congruence_System, 196
 Parma.Polyhedra.Library::Constraint, 216
 Parma.Polyhedra.Library::Generator, 250
 Parma.Polyhedra.Library::Grid_Generator, 298
 sgn
 Parma.Polyhedra.Library::Checked_Number, 174
 shift_space_dimensions
 Parma.Polyhedra.Library::Congruence, 189
 Parma.Polyhedra.Library::Constraint, 216
 Parma.Polyhedra.Library::Generator, 250
 Parma.Polyhedra.Library::Grid_Generator, 298
 Parma.Polyhedra.Library::Linear_Expression, 321
 sign_normalize
 Parma.Polyhedra.Library::Congruence, 189
 Parma.Polyhedra.Library::Linear_Expression, 322
 simplify
 Parma.Polyhedra.Library::Polyhedron, 490
 simplify_using_context_assign
 Parma.Polyhedra.Library::BD_Shape, 108
 Parma.Polyhedra.Library::Box, 139
 Parma.Polyhedra.Library::Grid, 282
 Parma.Polyhedra.Library::Interval, 311
 Parma.Polyhedra.Library::Octagonal_Shape, 370
 Parma.Polyhedra.Library::Pointset_Powerset, 448
 Parma.Polyhedra.Library::Polyhedron, 478
 solution
 Parma.Polyhedra.Library::PIP_Problem, 425
 solve
 Parma.Polyhedra.Library::MIP_Problem, 342
 Parma.Polyhedra.Library::PIP_Problem, 425
 Parma.Polyhedra.Library::PIP_Tree_Node, 432
 space_dimension
 Parma.Polyhedra.Library::Variable, 506
 sqrt_assign
 Parma.Polyhedra.Library::Checked_Number, 171
 Parma.Polyhedra.Library::GMP_Integer, 262
 std, 87
 strictly_contains
 Parma.Polyhedra.Library::BD_Shape, 103
 Parma.Polyhedra.Library::Box, 135
 Parma.Polyhedra.Library::Grid, 277
 Parma.Polyhedra.Library::Octagonal_Shape, 362
 Parma.Polyhedra.Library::Partially_Reduced_Product, 400
 Parma.Polyhedra.Library::Pointset_Powerset, 444
 Parma.Polyhedra.Library::Polyhedron, 472
 strong_normalize
 Parma.Polyhedra.Library::Congruence, 190
 sub_linearize
 Parma.Polyhedra.Library::Concrete_Expression, 178
 sub_mul_assign
 Parma.Polyhedra.Library::Checked_Number, 170
 Parma.Polyhedra.Library::GMP_Integer, 261
 Parma.Polyhedra.Library::Linear_Expression, 324, 325
 swap
 Parma.Polyhedra.Library::BD_Shape, 117, 120
 Parma.Polyhedra.Library::Box, 149, 151
 Parma.Polyhedra.Library::Cast_Floating_Point_Expression, 159
 Parma.Polyhedra.Library::Checked_Number, 174, 175
 Parma.Polyhedra.Library::Congruence, 190, 191
 Parma.Polyhedra.Library::Congruence_System, 196
 Parma.Polyhedra.Library::Constant_Floating_Point_Expression, 210
 Parma.Polyhedra.Library::Constraint, 218, 219
 Parma.Polyhedra.Library::Constraint_System, 222
 Parma.Polyhedra.Library::Determinate, 226
 Parma.Polyhedra.Library::Difference_Floating_Point_Expression, 229
 Parma.Polyhedra.Library::Division_Floating_Point_Expression, 232
 Parma.Polyhedra.Library::Generator, 251, 255
 Parma.Polyhedra.Library::Generator_System, 259
 Parma.Polyhedra.Library::Grid, 290, 291
 Parma.Polyhedra.Library::Grid_Generator, 299, 300
 Parma.Polyhedra.Library::Grid_Generator_System, 304
 Parma.Polyhedra.Library::Interval, 312
 Parma.Polyhedra.Library::Linear_Expression, 324, 325
 Parma.Polyhedra.Library::Linear_Expression::const_iterator, 203
 Parma.Polyhedra.Library::Linear_Form, 332, 334

Parma_Polyhedra_Library::MIP_Problem, 343
 Parma_Polyhedra_Library::Multiplication_Floating_Point_Expression, 346
 Parma_Polyhedra_Library::Octagonal_Shape, 382
 Parma_Polyhedra_Library::Opposite_Floating_Point_Expression, 387
 Parma_Polyhedra_Library::Partially_Reduced_Product, 411, 412
 Parma_Polyhedra_Library::PIP_Problem, 426
 Parma_Polyhedra_Library::PIP_Tree_Node::Artificial_Parameter, 90
 Parma_Polyhedra_Library::Pointset_Powerset, 454, 455
 Parma_Polyhedra_Library::Polyhedron, 491
 Parma_Polyhedra_Library::Powerset, 497
 Parma_Polyhedra_Library::Sum_Floating_Point_Expression, 502
 Parma_Polyhedra_Library::Variable, 506
 Parma_Polyhedra_Library::Variable_Floating_Point_Expression, 508
 termination_test_MS
 C++ Language Interface, 71
 termination_test_MS_2
 C++ Language Interface, 72
 termination_test_PR
 C++ Language Interface, 77
 termination_test_PR_2
 C++ Language Interface, 77
 time_elapse_assign
 Parma_Polyhedra_Library::BD_Shape, 111
 Parma_Polyhedra_Library::Box, 142
 Parma_Polyhedra_Library::Grid, 284
 Parma_Polyhedra_Library::Octagonal_Shape, 374
 Parma_Polyhedra_Library::Partially_Reduced_Product, 408
 Parma_Polyhedra_Library::Pointset_Powerset, 450
 Parma_Polyhedra_Library::Polyhedron, 481
 total_memory_in_bytes
 Parma_Polyhedra_Library::Checked_Number, 169, 174
 trunc_assign
 Parma_Polyhedra_Library::Checked_Number, 170
 Type
 Parma_Polyhedra_Library::Constraint, 215
 Parma_Polyhedra_Library::Generator, 249
 Parma_Polyhedra_Library::Grid_Generator, 297
 UNBOUNDED_MIP_PROBLEM
 C++ Language Interface, 70
 UNFEASIBLE_MIP_PROBLEM
 C++ Language Interface, 70
 UNFEASIBLE_PIP_PROBLEM
 C++ Language Interface, 70
 C++ Language Interface, 68
 UNDEFINED
 C++ Language Interface, 69
 unconstrain
 Parma_Polyhedra_Library::BD_Shape, 107
 Parma_Polyhedra_Library::Box, 138
 Parma_Polyhedra_Library::Grid, 280
 Parma_Polyhedra_Library::Octagonal_Shape, 369
 Parma_Polyhedra_Library::Partially_Reduced_Product, 402
 Parma_Polyhedra_Library::Pointset_Powerset, 446, 447
 Parma_Polyhedra_Library::Polyhedron, 477
 update_solution
 Parma_Polyhedra_Library::PIP_Solution_Node, 428
 update_tableau
 Parma_Polyhedra_Library::PIP_Tree_Node, 432
 upper_bound_assign
 Parma_Polyhedra_Library::BD_Shape, 108
 Parma_Polyhedra_Library::Box, 139
 Parma_Polyhedra_Library::Grid, 280
 Parma_Polyhedra_Library::Linear_Form, 332
 Parma_Polyhedra_Library::Octagonal_Shape, 370
 Parma_Polyhedra_Library::Partially_Reduced_Product, 403
 Parma_Polyhedra_Library::Powerset, 496
 upper_bound_assign_if_exact
 Parma_Polyhedra_Library::BD_Shape, 108
 Parma_Polyhedra_Library::Box, 139
 Parma_Polyhedra_Library::Grid, 280
 Parma_Polyhedra_Library::Octagonal_Shape, 370
 Parma_Polyhedra_Library::Partially_Reduced_Product, 403
 Parma_Polyhedra_Library::Powerset, 496
 V_CVT_STR_UNK
 C++ Language Interface, 68
 V_DIV_ZERO
 C++ Language Interface, 68
 V_EMPTY
 C++ Language Interface, 67
 V_EQ
 C++ Language Interface, 67
 V_EQ_MINUS_INFINITY
 C++ Language Interface, 67
 V_EQ_PLUS_INFINITY
 C++ Language Interface, 67
 V_GE
 C++ Language Interface, 67
 V_GT
 C++ Language Interface, 67

V_GT_MINUS_INFINITY
C++ Language Interface, [67](#)
V_GT_SUP
C++ Language Interface, [67](#)
V_INF_ADD_INF
C++ Language Interface, [68](#)
V_INF_DIV_INF
C++ Language Interface, [68](#)
V_INF_MOD
C++ Language Interface, [68](#)
V_INF_MUL_ZERO
C++ Language Interface, [68](#)
V_INF_SUB_INF
C++ Language Interface, [68](#)
V_LE
C++ Language Interface, [67](#)
V_LGE
C++ Language Interface, [67](#)
V_LT
C++ Language Interface, [67](#)
V_LT_INF
C++ Language Interface, [67](#)
V_LT_PLUS_INFINITY
C++ Language Interface, [67](#)
V_MOD_ZERO
C++ Language Interface, [68](#)
V_NAN
C++ Language Interface, [68](#)
V_NE
C++ Language Interface, [67](#)
V_OVERFLOW
C++ Language Interface, [67](#)
V_SQRT_NEG
C++ Language Interface, [68](#)
V_UNKNOWN_NEG_OVERFLOW
C++ Language Interface, [68](#)
V_UNKNOWN_POS_OVERFLOW
C++ Language Interface, [68](#)
V_UNREPRESENTABLE
C++ Language Interface, [68](#)
VC_MINUS_INFINITY
Parma_Polyhedra_Library, [85](#)
VC_NAN
Parma_Polyhedra_Library, [85](#)
VC_NORMAL
Parma_Polyhedra_Library, [85](#)
VC_PLUS_INFINITY
Parma_Polyhedra_Library, [85](#)
VR_EMPTY
Parma_Polyhedra_Library, [86](#)
VR_EQ
Parma_Polyhedra_Library, [86](#)
VR_GE
Parma_Polyhedra_Library, [86](#)
VR_GT
Parma_Polyhedra_Library, [86](#)
VR_LE
Parma_Polyhedra_Library, [86](#)
VR_LGE
Parma_Polyhedra_Library, [86](#)
VR_LT
Parma_Polyhedra_Library, [86](#)
VR_NE
Parma_Polyhedra_Library, [86](#)
Variable
Parma_Polyhedra_Library::Variable, [505](#)
variable
Parma_Polyhedra_Library::Linear_Expression::const_iterator, [203](#)
Parma_Polyhedra_Library::Linear_Expression_-Impl::const_iterator, [201](#)
Parma_Polyhedra_Library::Linear_Expression_-Interface::const_iterator_interface, [207](#)
Variables_Set
Parma_Polyhedra_Library::Variables_Set, [510](#)
widen_fun_ref
Parma_Polyhedra_Library::Pointset_Powerset, [453](#), [454](#)
widening_assign
Parma_Polyhedra_Library::Grid, [286](#)
Parma_Polyhedra_Library::Partially_Reduced_Product, [408](#)
wrap_assign
Parma_Polyhedra_Library::BD_Shape, [111](#)
Parma_Polyhedra_Library::Box, [142](#)
Parma_Polyhedra_Library::Grid, [285](#)
Parma_Polyhedra_Library::Octagonal_Shape, [374](#)
Parma_Polyhedra_Library::Pointset_Powerset, [450](#)
Parma_Polyhedra_Library::Polyhedron, [481](#)
wrap_string
Parma_Polyhedra_Library::IO_Operators, [87](#)