POLKA

The POLKA Convex Polyhedra library
Edition 2.1.0c, 2007

by Bertrand Jeannet

POLKA

1 Introduction to POLKA

New Polka is a library to handle convex polyhedra, whose constraints and generators have rational coefficients. It is programmed in ANSI C, so you can use it in any C or C++ programs. An interface to the language OCaml version 3.00 is also provided. This library is currently used in my verification tool NBac, and also by others research teams working on static analysis and abstract interpretation.

It is mainly based on the IRISA library Polylib and the old library used in the Polka tool inside the synchronous team of the laboratory VERIMAG. The main motivation to develop a new library was the need for multi-precision integers and 64 bits integers. The interface and memory management have also been improved (according to the author !), and saturation matrices can be kept in memory, thus saving computation time. There is also an option to handle strict constraints like x>y.

Implemented operations include creation of polyhedra from constraints or generators, intersection, convex hull, image and preimage by linear transformations, widening operator (as used in linear relation analysis, see publications of Nicolas Halbwachs about this technique), and inspection of saturation matrices.

The C interface is simple but also quite rough. The OCaml interface offers however pretty input and output of constraints, matrices and polyhedra, and gives you a polyhedra desk calculator thanks to the OCaml toplevel.

2 POLKA and the other polyhedra libraries

2.1 Comparison with Polylib library

Our objectives to develop our own version was at that time (1999) to use a Polylib-like library with multi-precision integers, and secondary to implement some parts differently. Our library is also more simpler because we don't consider explicit unions of polyhedra as in IRISA's library.

The main differences are sketched below:

1. we don't consider unions of polyhedra;
2. we can use either machine or multi-precision integers;
3. strict inequalities are implemented;
4. conversion and minimization operations can be delayed; IRISA's library always keep both representation, whereas we keep ordinary only one and do a conversion only when it is mandatory, for example when we wants to intersect polyhedra whose constraints are not available; when both representations are available, they are necessarily minimal, butthat's not the case when only one is available;
5. when both representations are available for a polyhedra, we keep also the saturation matrix. This allows to perform intersection and convex hull with another polyhedron in an quite incremental way, because we don't compute it again.
6. in some case of variable assignation or substitution on a polyhedron, when the transformation is inversible, we operate on both representation if they were already available; it is the case too for the embeding or projection into a space with supplementary dimensions.
7. when we do the conversion from constraints to generators, for instance, we obtain a minimal set of the generators, but the set of constraints is still not minimal. The algorithm which perform this minimization seems to me cheaper that the one of IRISA; I must confess however that I have not fully understood the IRISA's algorithm.
8. we implements *widening* operators, as defined in *cousot78,polka:fmsd:97*.

2.2 Inspiration coming from Cdd library

We took from the Cdd library the following ideas: matrices rows are lexicographically sorted (lazily, as for conversion). This presents two advantages: *fukuda96* observed that it speeds conversion of representation. In addition when we merge constraints of two polyhedra in intersection operations, this allows us to remove identical constraints easily.

2.3 Other libraries

The Polylib library has been superseded by the PolyLib library, which now offers multi-precision arithmetic. It provides sophisticated operations used in automatic parallelization of programs.

The Parma library is a C++ library, which used POLKA as a starting point.

3 Installing POLKA

3.1 Requested tools

To compile the C library, you need

- an ANSI C compiler (only GCC with '`-ansi`' option has been really tested);
- GNU Make
- GMP (Gnu Multi-Precision library) if you want the version of POLKA with multi-precision arithmetic.

In addition, if you want the OCaml interface, you need

- The OCaml system, version 3.00 or higher;
- The CamlIDL stub code generator, version 1.04 or higher; actually, only the runtime library is really required;
- Possibly, The SED Stream EDitor (GNU version available at `http://www.gnu.org/software/software.html`) if you want to regenerate the stub source files.

This documentation has been generated with the TEXINFO system, using the executables `texi2dvi`,`makeinfo` and `texi2html`.

3.2 Configuration

Configuration is performed by setting variables in the '`Makefile.config`' file. The requested paths give the *prefix* directories. For instance, the GMP header file will be found in '`$(GMP_INSTALL)/include`'.

3.3 Building the libraries

Building both C library and OCaml interface

'`make alli`'
: Builds the libraries with normal `long int` machine integers;

'`make alll`'
: Builds the libraries with `long long int` machine integers; `long long ints` are recognized by GCC and in the ISO C99 standard;

'`make allg`'
: Builds the libraries with GMP integers;

'`make all`'
: Builds all the version of the library.

Installation is done with '`make install`'.

Using GMP integers prevents any overflow problems. Be cautious, these are not detected ! The observed effect is usually an infinite loop.

Building the C library only

Enter the 'C' directory, and type the above-mentioned commands.

Generated files

'make all%' where $\% = i, l, g$ generates the 'libpolka%.a' C library, and the following public OCaml files:

- 'libpolka%_caml.a': C interface library
- 'polka.cma': bytecode OCaml library
- 'polka.cmxa' and 'polka.a': native OCaml library
- '(polka,vector,matrix,poly,polkaIO).(cmi|cmx)': bytecode/native OCaml interface files

Building the documentation

Type 'make doc'. The generated files 'polka.dvi', 'polka.ps', 'polka.info' and 'html/*' can be found in the 'documentation' directory.

4 Using POLKA

4.1 Convex polyhedra and their representation

4.1.1 Basic facts about polyhedra

Convex polyhedra have two dual possible representations: you can define a polyhedron with by giving either a set of linear constraints or a set of generators:
$P = \{x\backslash inQ\hat{}d|A.x>=b\}$
$P = \{x\backslash inQ\hat{}d|\backslash foralli : A\_i.x>=b\_i\}$
or
$P = \{x\backslash inQ\hat{}d|x = V.\backslash lambda{+}R.\backslash mu, \backslash sum\_i\backslash lambda\_i = 1, \backslash mu>=0\}$
$P = \{x\backslash inQ\hat{}d|x = \backslash sum\_i\backslash lambda\_iV\_i{+}\backslash sum\_j\backslash mu\_jR\_j\}$
A is the constraints matrix, V the vertices matrix and R the rays matrix. $x, b, \backslash lambda, \backslash mu$ are column vectors.

Working in a linear framework instead of an affine one is much more simpler, as a consequence we will embed $Q\hat{}d$ in $Q\hat{}(d{+}1)$ in a classical way with the transformation $x{\mapsto}(\backslash xi = 1, x)$ with $\backslash xi>=0$, as explained for example in *wilde93*. Any polyhedron will then be a cone. The reverse transformation is the intersection of the cone with the hyperplane $\backslash xi = 1$. This allows us to concentrate our attention to cones, dual representations of which are:
$P = \{x\backslash inQ\hat{}(d{+}1)|A.x>=0\}$
$P = \{x\backslash inQ\hat{}(d{+}1)|x = R.\backslash mu, \backslash mu>=0\}$ The double description method is a method to convert from one representation to another and to minimize the size of representation. This allows easily to perform intersection of convex polyhedra, by merging the constraints of the involved polyhedra, or convex hull, by merging generators of the involved polyhedra.

4.1.2 Working with strict inequalities

Using strict inequalities can be done by introducing a second special variable $\backslash epsilon$, which satisfies $\backslash epsilon>=0$. A polyhedron is then empty if its intersection with the half-space $\backslash epsilon{>}0$ is empty, or in other words, if there is no vertex whose coefficient associated to $\backslash epsilon$ is strictly positive. The inclusion test need also a (more complicated) adaptation.

4.1.3 Representation of constraints, generators and affine expressions

Coefficients

Normally we shoud use rational numbers for coefficients of vectors and matrices. *avis98b* observed experimentally in a very similar context that using a common denominator for all coefficients of a vector or of a matrix row is more efficient, probably because vector combination is easier. *wilde93* uses also the same technique.

To avoid overflow problem, multi-precision integers are needed; their drawback is obviously the loss of speed. As a consequence, we have defined a generic interface for integers and the compile-time option allows to deal with either

- multi-precision integers; we use for that the GNU Multi Precision library (GMP), which reveals very efficient according to K. Fukuda's home page;
- normal machine integers (32 or 64 bits long int's), with at this time no overflow check;
- and long long int's, introduced by the new ANSI C 99 standard and allowed by gcc and some others compilers.

The type `pkint_t` is supposed to provide conversion operators from machine `int` to type `pkint_t`. You should look at file '`gint.nw`' for details.

Format of frames, generators and affine expressions

As in *wilde93* we reserve a particular treatment for equality constraints, instead of considering them as two opposed inequalities, because more efficient methods are available for example to minimize them (gauss pivot). Dually, we distinguish for generators *bidirectionnal* lines from normal rays.

The format of objects depends if the library has been opened with the `strict` option or not. We note d the dimension of affine polyhedra, i.e., the number of dimensions different of $\backslash xi$ and $\backslash epsilon$.

If the `strict` option is unset (no strict inequalities):

- `[0,b,a_0,...,a_{d-1}]` represents the equality constraint $a\_0x\_0+...+a\_\{d-1\}x\_\{d-1\}+b=0$;
- `[1,b,a_0,...,a_{d-1}]` represents the inequality constraint $a\_0x\_0+...+a\_\{d-1\}x\_\{d-1\}+b>=0$;
- `[0,0,a_0,...,a_{d-1}]` represents the line of direction $(a\_0,...,a\_\{d-1\})$;
- `[1,0,a_0,...,a_{d-1}]` represents the ray of direction $(a\_0,...,a\_\{d-1\})$;
- `[1,b,a_0,...,a_{d-1}]` represents the vertex $(a\_0/b,...,a\_\{d-1\}/b)$;
- `[d,b,a_0,...,a_{d-1}]` represents the affine expression $x\mapsto a\_0/dx\_0+...+a\_\{d-1\}/dx\_\{d-1\}+b/d$;

The nature of a vector: constraint, generator, or affine expression, is inferred by the context. As you can guess, index 0 is used either to distinguish bidirectional or unidirectional vectors (0 or 1), either to put a denominator; index 1 is the $\backslash xi$-coefficient, used for constants.

If the `strict` option is set (strict inequalities), an additionnal dimension is introduced at index 2 to put $\backslash epsilon$-coefficients:

- `[0,b,0,a_0,...,a_{d-1}]` represents the equality constraint $a\_0x\_0+...+a\_\{d-1\}x\_\{d-1\}+b=0$;
- `[1,b,0,a_0,...,a_{d-1}]` represents the inequality constraint $a\_0x\_0+...+a\_\{d-1\}x\_\{d-1\}+b>=0$;
- `[1,b,-s,a_0,...,a_{d-1}]` where $s>0$ represents the inequality constraint $a\_0x\_0+...+a\_\{d-1\}x\_\{d-1\}+b>0$;
- `[0,0,0,a_0,...,a_{d-1}]` represents the line of direction $(a\_0,...,a\_\{d-1\})$;
- `[1,0,0,a_0,...,a_{d-1}]` represents the ray of direction $(a\_0,...,a\_\{d-1\})$;
- `[1,b,s,a_0,...,a_{d-1}]` where $s>=0$ represents the vertex $(s/b)|(a\_0/b,...,a\_\{d-1\}/b)$;

- `[den,b,0,a_0,...,a_{d-1}]` represents the affine expression
  $x \mapsto a\_0/denx\_0 + ... + a\_\{d-1\}/denx\_\{d-1\} + b/den$;

Don't ask me the intuitive meaning of $s>=0$ in vertices !

If a vector is given without matching a suitable format, depending on its nature, the results are unpredictable. That's come from the fact that the library uses the assumption $\backslash xi>=0$ and possibly $\backslash epsilon>=0$ to decide wether a polyhedron is empty or not.

The constraint $\backslash xi>=0$ ($\backslash xi>=\backslash epsilon$ wit strict option) is called the *positivity constraint*, and the constraint $\backslash epsilon>=0$ the *strictness constraint*.

To make more easy a certain form of genericity, the library offers variables `bool polka_strict` and `int polka_dec` that memorized respectively the operation mode (strict or non strict) and the index of the first "normal" coefficient (2 or 3). The index of the constant coefficient is in any case 1.

4.2 Using C library

First include some of the following files in your source program, say '`test.c`'. A casual user doesn't need the header '`polka/{bit|satmat}.h`'.

```
#include <polka/polka.h>
#include <polka/pkint.h>
#include <polka/vector.h>
#include <polka/bit.h>
#include <polka/satmat.h>
#include <polka/matrix.h>
#include <polka/poly.h>
```

'`polka/config.h`', '`polka/cherni.h`' an '`polka/cherni.h`' should be considered as internal.

Types, variables and functions in each module are prefixed by the name of the module. For instance, we have the functions `matrix_t* matrix_merge(matrix_t*,matrix_t*)` and `void poly_minimize(poly_t*)`.

Coefficients in vectors and matrices are of generic type `pkint_t`.

To compile '`test.c`', type '`gcc <options> -DPOLKA_NUM=<n> -c -o test.o test.c`'
where n is either $1, 2, 3$ and defines the effective type of `pkint_t`.

To link '`test.o`', type '`gcc <options> -o test test.o -lpolka% -lgmp`'
where % is either i, l, g.

n and % should respect the following correspondance:

n	%	Effective type of `pkint_t`
1	i	`long int`
2	l	`long long int`
3	g	`mpz_t`

4.3 Using OCaml library

The OCaml interface defines the modules `Polka`, `Vector`, `Matrix` and `Poly`.

To compile a file '`test.ml`' into bytecode, first generate a suitable custom bytecode interpreter with

- ‘ocaml <options> -make_runtime -o polkarun% polka.cma \
  -cclib "-L<path> -lpolka%_caml -lpolka% -lgmp -lcamlidl"’

You can then compile ‘test.ml’ and link it with:

- ‘ocamlc <options> -c test.ml’
- ‘ocamlc -use_runtime polkarun% -o test polka.cma test.cmo’

To compile ‘test.ml’ into native code:

- ‘ocamlopt <options> -c test.ml’
- ‘ocamlopt -o test polka.cmxa test.cmx \
  -cclib "-L<path> -lpolka%_caml -lpolka% -lgmp -lcamlidl"’

5 C Library

This chapter describes the C API of Polka, which is the native API of the library`<`.

5.1 Organization of the C library

- Files '`config.h`' and '`pkint.h`' respectively define some configuration stuff and the operations on generic integers.
- '`polka.c`' and '`polka.h`' define global variables that set the operation mode and the maximum size of matrices, and the initialization finalization functions.
- '`internal.c`' and '`internal.h`' define the internal global variables.
- '`bit.c`' and '`bit.h`' define types `bitstring_t` and `bitindex_t` that allow to acces and perform operations on bitstrings.
- '`satmat.c`' and '`satmat.h`' define the type `satmat_t` of saturation matrices and the needed operations.
- '`vector.c`' and '`vector.h`' define operations on vector, considered as C arrays;
- '`matrix.c`' and '`matrix.h`' define the type `matrix_t` and needed operations on it, including rows sorting, matrices merging, transformation by assignation or substitution of a variable by an affine transformation;
- '`cherni.c`' and '`cherni.h`' contain the conversion and minimization algorithms;
- last, '`poly.c`' and '`poly.h`' define the `poly_t` and operations on polyhedra.

In principle, a user needs only to know the '`polka`', '`vector`', '`matrix`' and '`poly`' interfaces, and some of '`pkint`'.

5.2 Module pkint

This module is defined by the file 'polka_num.h', included from 'polka.h'. It defines generic operations on integers. The naming scheme and the semantics of these operations comes from the GMP library. They assume in particular that all integer objects are initialized before being used.

pkint_t datatype

```
typedef struct pkint_t {
  ACTUALTYPE rep;
} pkint_t;
```

The generic type of coefficients in vectors and matrices. The actual type is defined by the `configure` command.

void **pkint_init** (pkint_t *integer*) Function

Initialize *integer* and set its value to 0.

void **pkint_clear** (pkint_t *integer*) Function

Free the space possibly used by *integer*. Make sure to call this function for all `pkint_t` variables when you are done with them.

void **pkint_set** (pkint_t *rop*, pkint_t *op*) Function
void **pkint_set_ui** (pkint_t *rop*, unsigned int *op*) Function
void **pkint_set_si** (pkint_t *rop*, signed int *op*) Function

Set the value of *rop* from *op*.

void **pkint_init_set** (pkint_t *rop*, pkint_t *op*) Function
void **pkint_init_set_ui** (pkint_t *rop*, unsigned int *op*) Function
void **pkint_init_set_si** (pkint_t *rop*, signed int *op*) Function

Initialize *rop* and set its value from *op*.

void **pkint_add** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $op1 + op2$.

void **pkint_sub** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $op1 - op2$.

void **pkint_neg** (pkint_t *rop*, pkint_t *op*) Function

Set *rop* to $-op$.

void **pkint_mul** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $op1 * op2$.

void **pkint_addmul** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $rop + op1 * op2$.

void **pkint_submul** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $rop - op1 * op2$.

void **pkint_div** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $op1/op2$.

void **pkint_mod** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to *op1* modulo *op2*.

void **pkint_abs** (pkint_t *rop*, pkint_t *op*) Function

Set *rop* to the absolute value of *op*.

int **pkint_sgn** (pkint_t *integer*) Function

Return the sign of *integer*: 0 if null, > 0 if positive, < 0 if negative.

void **pkint_gcd** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to Greatest Common Divisor of *op1* and *op2*.

void **pkint_divexact** (pkint_t *rop*, pkint_t *op1*, pkint_t *op2*) Function

Set *rop* to $op1/op2$, **assuming that** *op2* **is a divisor of** *op1*.

int **pkint_cmp** (pkint_t *op1*, pkint_t *op2*) Function
int **pkint_cmp_ui** (pkint_t *op1*, unsigned long int *op2*) Function
int **pkint_cmp_si** (pkint_t *op1*, signed long int *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

unsigned long int **pkint_get_ui** (pkint_t *integer*) Function

Return the least significant part from *integer*.

signed long int **pkint_get_si** (pkint_t *integer*) Function

If *integer* fits into a `signed long int` return the value of *integer*. Otherwise return the least significant part of *integer*, with the same sign as *integer*.

If *integer* is too large to fit in a `signed long int`, the returned result is probably not very useful. To find out if the value will fit, use the function `pkint_fits_slong_p`.

void **pkint_set_str10** (pkint_t *integer*, char* *str*) Function

Put in *integer* the string representation in base 10 *str* of an integer. Behavior unspecified if *str* is not a correct representation.

void **pkint_sizeinbase10** (pkint_t *integer*) Function

Return the size of *integer* measured in number of digits in base 10. The sign of *integer* is ignored. The result may be too big than the exact value.

This function is useful in order to allocate the right amount of space before converting OP to a string. The right amount of allocation is normally two more than the value returned by `pkint_sizeinbase10`.

`void` **`pkint_get_str10`** (`char*` *str*, `pkint_t` *integer*) Function

Convert *integer* to a string of digits in base 10.

If *str* is `NULL`, the result string is allocated using `malloc`. The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator.

If *str* is not `NULL`, it should point to a block of storage large enough for the result, that being `pkint_sizeinbase10(`*integer*`) + 2`. The two extra bytes are for a possible minus sign, and the null-terminator.

A pointer to the result string is returned, being either the allocated block, or the given *str*.

`void` **`pkint_print`** (`pkint_t` *integer*) Function

Prints *integer* on the standard output.

5.3 Module Polka

This modules defines datatypes, some global read-only variables, and library initialization and finalization functions.

5.3.1 Datatypes

bool datatype

```
typedef enum bool {
  false = 0,
  true  = 1
} bool;
```

Boolean type.

tbool datatype

```
typedef enum tbool {
  tbool_bottom = 0,
  tbool_true   = 1,
  tbool_false  = 2,
  tbool_top    = 3
} tbool;
```

This the data-type for Boolean lattice, ordered as $bottom < \{true, false\} < top$.

dimsup_t datatype

```
typedef struct dimsup_t {
  int pos;
  int nbdims;
} dimsup_t;
```

Data-type for insertion and deletion of columns in vectors, matrices, and polyhedra.

equation_t datatype

```
typedef struct equation_t {
  int var;
  pkint* expr;
} equation_t_t;
```

Data-type for performing parallel assignements and substitutions on matrices and polyhedra.

5.3.2 Initialization and finalization functions

`void` **polka_initialize** (`bool` *strict*, `int` *maxdims*, `int` *maxrows*) Function

Initialize the global variables of the library. *strict* indicates the operation mode of polyhedra and allows to set global variables `polka_strict` and `polka_dec` properly. *maxdims* and *maxrows* are respectively the maximum dimension of polyhedra and the maximum number of constraints and/or generators in polyhedra. This also means that matrices can have at most *maxrows* rows and that vectors and matrices have at most *polka_dec+maxdims* columns.

void **polka_finalize** () Function

Frees all the memory allocated for global variables in module 'polka'.

void **polka_set_widening_affine** () Function

Select the "affine mode" for the widening operation on polyhedra (see Section 5.8.8 [Widening operators on polyhedra lattice], page 35). In the affine mode, widening is performed on affine polyhedra instead of on underlying linear cones. Typically, `x=1` widened by `1<=x<=2` will give `x>=1` in affine mode, whereas it will give `1<=x<=2` in linear mode.

void **polka_set_widening_linear** () Function

Select the "linear mode" for the widening operation on polyhedra (see Section 5.8.8 [Widening operators on polyhedra lattice], page 35). In the linear mode, widening is performed on underlying linear cones, and not on the affine polyhedra they represent. This is the default mode.

5.3.3 Global variables

The public read-only global variables initialized by the function `polka_initialize` are the following ones:

bool **polka_strict** Variable

True iff. strict inequalities are enable. This requires an additional dimension in vectors and matrices, and modifies emptiness and universality tests.

const **int polka_cst** Variable

Indicates the index of the constant coefficient. Should be always 1, weather strict inequalities are enabled or not.

const **int polka_eps** Variable

Indicates the index of the epsilon coefficient. Should be 2.

int **polka_dec** Variable

Indicates the index of the first "normal" coefficient; 2 if `polka_strict` is false, 3 otherwise.

int **polka_maxnbdims** Variable

Maximum number of dimensions allowed in polyhedra

int **polka_maxnbrows** Variable

Maximum number of rows allowed in matrices.

int **polka_maxcolumns** Variable

Maximum number of columns allowed in vectors and matrices.

5.3.4 Functions on equation

`void` **`translate_equations`** (`equation_t*` *eqn*, `size_t` *size*, `int` *offset*) Function

Add *offset* to the field `.var` of equations belonging to the array *eqn* of size *size*. *offset* may be negative or positive.

`int` **`cmp_equations`** (`const equation_t*` *eqna*, `const equation_t*` *eqnb*) Function

Compare the field *.var* of the two equations.

`int` **`sort_equations`** (`equation_t*` *eqn*, `size_t` *size*) Function

Sort the array *eqn* of size *size* according to the field *var* in ascending order.

5.4 Module Vector

Vectors are implemented in file '`vector.c`'.

5.4.1 The type vector_t

`vector_t` datatype

```
typedef pkint_t* vector_t;
```

A vector is a C array of elements of type `pkint_t`. You access to the kth element of the array `q` with `q[k]`.

5.4.2 Basic Operations on vectors

`pkint_t*` **`vector_alloc`** (`size_t` *size*) Function

Allocates an array of *size* elements of type `pkint_t` and initializes them to zero.

`void` **`vector_free`** (`pkint_t*` *q*, `size_t` *size*) Function

Frees the array *q* of size *size*. The *size* parameter is needed because when the type `pkint_t` is a multi-precision integer, elements have to be finalized.

`void` **`vector_clear`** (`pkint_t*` *q*,`size_t` *size*) Function

This function set all elements of the array *q* to 0.

`pkint_t*` **`vector_copy`** (`const pkint_t*` *q*, `size_t` *size*) Function

Creates a copy of the array *q* of size *size*.

`void` **`vector_print`** (`const pkint_t*` *q*, `size_t` *size*) Function

Prints the array.

5.4.3 Comparison & Hashing on vectors

`int` **`vector_compare`** (`const pkint_t*` *q1*, `const pkint_t*` *q2*, `size_t` *size*) Function

Compares the two arrays *q1* and *q2* of same size *size*, considered as constraints or generators, lexicographically on the order $0, polka\_dec, ..., polka\_dec{+}size{-}1, 1[, 2]$. The returned `int` has the following meaning:

- <0 *r1* is smaller than *r2*
- = 0 they are equal;
- >0 *r1* is greater than *r2*;
- *abs*() = 1 in addition they are parallel.

For two *parallel* inequalities (equal coefficients apart for the *xi* and *epsilon* dimensions), the defined order corresponds to the entailment of constraints (if either `polka_strict` is true or not): if *c_1* is less or equal than *c_2*, then *c_1* == >*c_2*.

int **vector_compare_expr** (const pkint_t* *q1*, const pkint_t* *q2*, size_t *size*) Function

Idem as previous functions for vectros considered as affine expressions.

unsigned int **vector_hash** (const pkint_t* *q*, size_t *size*) Function

This function computes a key associated to an array. The function is compatible with the previous comparison function, that is, two identical vectors give the same key.

5.4.4 Normalization of vectors

void **vector_normalize** (pkint_t* *q*, size_t *size*) Function

Normalizes the array considered as a constraint or a generator. The first coefficient is ignored.

void **vector_normalize_expr** (pkint_t* *q*, size_t *size*) Function

Normalizes the array considered as an affine expression. The first coefficient is updated.

5.4.5 Algebraic Operations on vectors

void **vector_combine** (const pkint_t* *q1*, const pkint_t* *q2*, pkint_t* *q3*, int *k*, size_t *size*) Function

This function computes a linear combination of *q1* and *q2* and puts it array *q3* such that `q3[k]=0`. The first coefficient is never considered for computations, except when `k==0`. The result is normalized.

void **vector_product** (pkint_t* *prod*, const pkint_t* *q1*, const pkint_t* *q2*, size_t *size*) Function

Computes the scalar product of arrays *q1* and *q2* of common size *size* and stores it in **prod*. The first coefficient is ignored.

void **vector_product_strict** (pkint_t* *prod*, const pkint_t* *q1*, const pkint_t* *q2*, size_t *size*) Function

As the previous function, but the *epsilon*-coefficients are ignored.

void **vector_add** (pkint_t* *q3*, const pkint_t* *q1*, const pkint_t* *q2*, size_t *size*) Function

This function computes the addition of *q1* and *q2* considered as affine expressions. The result is normalized and put in *q3*.

void **vector_sub** (pkint_t* *q3*, const pkint_t* *q1*, const pkint_t* *q2*, size_t *size*) Function

This function computes the substraction of *q1* by *q2* considered as affine expressions. The result is normalized and put in *q3*.

void **vector_scale** (pkint_t* *q2*, pkint_t num, pkint_t den, const pkint_t* *q1*, size_t *size*) Function

This function scales *q1* considered as an affine expression by the fraction *num/den* and put the results in *q2*. *den* is supposed to be positive.

5.4.6 Change of dimension of vectors

At the end

void **vector_add_dimensions** (pkint_t* *q2*, const pkint_t* *q1*, size_t *size*, int *dimsup*) Function

If *dimsup* is positive, adds `dimsup` dimensions at the end of *q1* and puts the result in *q2*; If *dimsup* is negative, deletes the `-dimsup` last dimensions of *q1* and puts the result in *q2*. *q2* is supposed to have a sufficient size.

Anywhere

void **vector_add_dimensions_multi** (pkint_t* *q2*, const pkint_t* *q1*, size_t *size*, const dimsup_t* *tab*, int *multi*) Function

Fills the vector *q2* by inserting new columns with null values in vector *q1*, according to the array *tab* of size *multi*.

For each element `dimsup` of that array, `dimsup.nbdims` dimensions are inserted starting from rank `dimsup.pos` (of the initial vector). The coefficient `q1[polka_dec+dimsup.pos]` is pushed on the right. For instance, if the array *tab* is `[(0,2);(1,1);(2,3)]`,

```
the vector  [d,b,    a_0,  a_1,      a_2,a_3,...,a_{d-1}]
becomes     [d,b,0,0,a_0,0,a_1,0,0,0,a_2,a_3,...,a_{d-1}].
```

If the strict option is set, `[d,b,s,a_0,...,a_{d-1}]` becomes `[d,b,s,0,0,a_0,0,a_1,0,0,0,,a_2,a_3,...,a_{d-1}]`.

The array *tab* is supposed to be sorted in ascending order w.r.t. `tab[i].pos`. *q2* is supposed to have a sufficient size.

void **vector_remove_dimensions_multi** (pkint_t* *q2*, const pkint_t* *q1*, size_t *size*, const dimsup_t* *tab*, int *multi*) Function

Fills the vector *q2* by deleting some columns in vector *q1*, according to the array *tab* of size *multi*.

For each element `dimsup` of that array, `dimsup.nbdims` dimensions are deleted starting from rank `dimsup.pos` (of the initial vector). For instance, if the array *tab* is `[(0,2);(3,1);(5,3)]`,

```
the vector  [d,b,a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,...,a_{d-1}]
becomes     [d,b,        a_2,    a_4,            a_8, ...,a_{d-1}].
```

The array *tab* is supposed to be sorted in ascending order w.r.t. `tab[i].pos`. *q2* is supposed to have a sufficient size.

Change of dimensions together with permutation

void **vector_add_permute_dimensions** (pkint_t* *newq*, const pkint_t* *q*, size_t *size*, int *dimsup*, const int* *permut*) Function

dimsup is supposed to be positive or null. Add *dimsup* dimensions to *q* and apply the permutation *permutation*. The result is stored in *newq*. The array *permutation* is supposed to be of size size+dimsup-polka_dec.

The permutation *permutation* defines a permutation (i.e., a bijection) from [0..(size+dimsup-polka_dec)-1 to itself. BE CAUTIOUS: value 0 in the permutation means columns polka_dec.

void **vector_permute_remove_dimensions** (pkint_t* *newq*, const pkint_t* *q*, size_t *size*, int *dimsup*, const int* *permut*) Function

dimsup is supposed to be strictly positive. Apply the permutation *permutation* to *q*, then delete the last *dimsup* dimensions. The result is stored in *newq*. The array *permutation* is supposed to of size size-dimsup-polka_dec.

The permutation *permutation* defines a permutation (i.e., a bijection) from [0..(size-polka_dec)-1 to itself. BE CAUTIOUS: value 0 in the permutation means columns polka_dec.

5.4.7 Miscellaneous on vectors

bool **vector_is_null_strict** (const pkint_t* *q*, size_t *size*) Function

Tests if the vector projected on the non *epsilon* constraints is null. The function use global variables polka_strict and polka_dec.

bool **vector_is_positivity_constraint** (const pkint_t* *q*, size_t *size*) Function

Tests if the vector represents the positivity constraint. The function use global variables polka_strict and polka_dec.

bool **vector_is_strictness_constraint** (const pkint_t* *q*, size_t *size*) Function

Tests if the vector represents the strictness constraint, if strict option is set. Otherwise, returns false. The function use global variables polka_strict and polka_dec.

bool **vector_is_dummy_constraint** (const pkint_t* *q*, size_t *size*) Function

Tests if the vector represents the positivity or the strictness constraint (if strict option is set for the latter case). The function use global variables polka_strict and polka_dec.

5.5 Module Matrix

5.5.1 The type matrix_t

`matrix_t` datatype

A matrix is represented by a structure whose elements prefixed by `_` should be considered as private. You access to the element of row `i` and column `j` of the matrix `matrix_t* mat` with the expression `mat->p[i][j]`, whereas the vector (of type `vector_t`) corresponding to the row `i` is accessed with `mat->p[i]`. `mat->nbrows` and `mat->nbcolumns` gives you the actual number of rows and columns (the *used part*), on which ordinary functions operates, but space is allocated for up to `mat->_maxrows` rows.

For information, it is defined as follows:

```
typedef struct matrix_t {
  /* public part */
  pkint_t** p;     /* array of pointers to rows */
  int nbrows;      /* number of effective rows */
  int nbcolumns;   /* size of rows */

  /* private part */
  pkint_t* _pinit;  /* array of coefficients */
  int  _maxrows;   /* number of rows allocated */
  bool _sorted;
} matrix_t;
```

5.5.2 Access functions on matrices

They are implemented as macros.

`int` **`matrix_get_maxrows`** (`const matrix_t*` *mat*) Macro

Return the maximum number of rows the matrix can contains. The active rows are always located in the first `mat->nbrows` columns.

`bool` **`matrix_is_sorted`** (`const matrix_t*` *mat*) Macro

Tests if the matrix is sorted (this is only an access function).

5.5.3 Basic Operations on matrices

`matrix_t*` **`matrix_alloc`** (`int` *nr*, `int` *nc*, `bool s`) Function

Allocates a new matrix with *nr* rows and *nc* columns and sets all elements to zero. *s* indicates if once filled the matrix should be considered as already sorted or not, according to the lexicographic order.

If either *nr* or *nc* is zero, the matrix is said to be *degenerated* and no space is allocated for coefficients (fields `p` and `_pinit` of the matrix are null). *Only functions of this paragraph* handle correctly such matrices (this happens to be useful, for instance when interfacing with OCaml language).

void **matrix_free** (matrix_t* *mat*) Function

Frees the matrix *mat* and finalizes referenced elements.

matrix_t* **matrix_copy** (const matrix_t* *mat*) Function

Return a copy of *mat*. The new matrix is dimensionned from the active part of *mat*.

void **matrix_print** (const matrix_t* *mat*) Function

Prints the matrix.

void **matrix_clear** (matrix_t* *mat*) Function

Sets to zeros the active coefficients of *mat*.

5.5.4 Comparison & Hashing on matrices

int **matrix_compare** (const matrix_t* *ma*,const matrix_t* *mb*) Function

Compares the two matrices row by row. The order is compatible with that defined on vectors, that is, if for the first non equal row `i`, `ma->p[i]` is less than `mb->p[i]`, then the result is negative. In the case where the dimensions are different, compare first the number of rows, then the number of columns.

unsigned int **matrix_hash** (const matrix_t* *mat*) Function

This function computes a key associated to the matrix. The function is compatible with the previous comparison function, that is, two matrices which are identical with `matrix_compare` give the same key.

int **matrix_compare_sort** (matrix_t* *ma*, matrix_t* *mb*) Function

Compares matrices considered as sets of vectors. The two matrices are sorted if they are not yet sorted, and then compared row by row with `matrix_compare`. In the case where the dimensions are different, compare first the number of rows, then the number of columns.

unsigned int **matrix_hash_sort** (matrix_t* *mat*) Function

This function computes a key associated to the matrix considered as a set of vectors. The matrix is sorted if it is not yet sorted, and then hashed with the `matrix_hash`. The function is compatible with the previous comparison function, that is, two matrices which are identical with `matrix_compare_sort` gives the same key.

5.5.5 Operations on rows of matrices

int **matrix_compare_rows** (const matrix_t* *mat*, int *l1*, int *l2*) Function

Compares lexicographically rows *l1* and *l2* of matrix *mat* (see `vector_compare`).

void **matrix_normalize_row** (const matrix_t* *mat*, int *l*) Function

Normalizes the row *l* of the matrix (see `vector_normalize`).

void **matrix_combine_rows** (matrix_t* *mat*, int *l1*, int *l2*, int *l3*) Function

Combine rows *l1* and *l2* of the matrix and put the result in the row *l3* (see vector_combine).

void **matrix_exch_rows** (matrix_t* *mat*, int *l1*, int *l2*) Function

Exchange rows *1* and *l2* of the matrix.

bool **matrix_is_row_dummy_constraint** (const matrix* const mat, int *row*) Function

Tests if the given matrix row represents the positivity or the strictness constraint (if strict option is set for the latter case). The function use global variables poly_strict and poly_dec. Implemented as a macro.

5.5.6 Sorting & Merging matrices

void **matrix_sort_rows** (matrix_t* *mat*) Function

Sorts the rows of the matrix (by insertion sort), and remove duplicates.

void **matrix_sort_rows_with_sat** (matrix_t* *mat*, satmat_t* sat) Function

This variant permutes also rows of the saturation matrix *sat* together with rows of *mat*. There is here no handling of duplicates.

void **matrix_add_rows_sort** (matrix_t* *mat*, matrix_t* *cmat*) Function

Adds to the matrix *mat* the rows of matrix *cmat*. The matrices are sorted if they are not already sorted, and the output is sorted. Identical rows are eliminated. *mat* is supposed to be big enough to store the new rows (mat->_maxrows >= mat->nbrows + cmat->nbrows).

matrix_t* **matrix_merge_sort** (matrix_t* *mata*, matrix_t* *matb*) Function

This function merges the rows of *mata* and *matb*, which are sorted if they are not sorted, and returns a new sorted matrix. Identical rows are eliminated.

5.5.7 Linear transformation of matrices

In these functions, variables are referenced by their *real* index in the matrix. There is no check of bounds.

matrix_t* **matrix_assign_variable** (const matrix_t* *mat*, int *var*, const pkint_t* tab) Function

Computes the image of *mat* by the assignment of affine expression *tab* to variable *var*.

matrix_t* **matrix_substitute_variable** (const matrix_t* *mat*, int *var*, const pkint_t* tab) Function

Computes the image of *mat* by the substitution of variable *var* by affine expression *tab*.

`matrix_t*` **`matrix_assign_variables`** (`const matrix_t*` *mat*, `const equation_t*` *eqn*, `size_t` *size*) Function

Computes the image of *mat* by the parallel assignment of `eqn[i].var` by `eqn[i].expr`, for *i* between `0` and `size-1`. The array *eqn* is supposed to be sorted w.r.t. the field `.var`. You may use the function `sort_equations` to ensure this (see '`polka.h`').

`matrix_t*` **`matrix_substitute_variables`** (`const matrix_t*` *mat*, `const equation_t*` *eqn*, `size_t` *size*) Function

Computes the image of *mat* by the parallel substitution of `eqn[i].expr` by `eqn[i].expr`, for *i* between `0` and `size-1`. The array *eqn* is supposed to be sorted w.r.t. the field `.var`. You may use the function `sort_equations` to ensure this (see '`polka.h`').

5.5.8 Change of dimension of matrices

At the end

`matrix_t*` **`matrix_add_dimensions`** (`const matrix_t*` *mat*, `int` *dimsup*) Function

If *dimsup* is positive, adds `dimsup` columns on the righ in *mat*; if *dimsup* is negative, deletes the `-dimsup` last columns of *mat*.

Anywhere

See Section 5.4.6 [Change of dimension of vectors], page 19, for more details about the meaning of arrays of elements of type `dimsup_t`.

`matrix_t*` **`matrix_add_dimensions_multi`** (`const matrix_t*` *mat*, `const dimsup_t*` *tab*, `size_t` *size*) Function

Inserts new columns with null values in matrix *mat*, according to the array *tab* of size *size*.

`matrix_t*` **`matrix_remove_dimensions_multi`** (`const matrix_t*` *mat*, `const dimsup_t*` *tab*, `size_t` *size*) Function

Deletes some columns in matrix *mat*, according to the array *tab* of size *size*.

Change of dimensions together with permutation

`matrix_t*` **`matrix_add_permute_dimensions`** (`const matrix_t*` *mat*, `int` *dimsup*, `const int*` *permutation*) Function

Scale `vector_add_permute_dimensions` to matrices.

`matrix_t*` **`matrix_permute_remove_dimensions`** (`const matrix_t*` *mat*, `int` *dimsup*, `const int*` *permutation*) Function

Scale `vector_permute_remove_dimensions` to matrices.

5.6 Module Bit

This modules implements bitstrings and operations on them.

5.6.1 Type `bitindex_t`: accessing bits in bitstrings

To reference a particular bit in a bitstring, you have to use an index of type `bitindex_t`.

`bitindex_t` datatype

For information, the private datatype is defined as follows:

```
typedef struct bitindex_t {
  int index;  /* the index of the referenced bit, starting from 0 */
  int word;   /* the index of the word containing the bit, starting from 0 */
  bitstring_t bit; /* the mask selecting the right bit */
} bitindex_t;
```

`bitindex_t` **`bitindex_init`** (`int` *col*) Function

Return a bitindex referencing the *col*th bit of any bitstring.

`void` **`bitindex_inc`** (`bitindex_t*` *b*) Function

Increments the bitindex *b* by one, i.e. after the call the bitindex references the next bit.

`void` **`bitindex_dec`** (`bitindex_t*` *b*) Function

Decrements the bitindex *b* by one, i.e. after the call the bitindex references the next bit.

`int` **`bitindex_size`** (`int` *n*) Function

Return the number of elements of type `bitstring_t` needed to represent *N* bits.

5.6.2 Type bitstring_t: bitstrings

Users normally don't need to know anything in this section.

`bitstring_t` datatype

Bit strings are implemented as arrays of elements of type `bitstring_t`, equivalent to some kind of unsigned integer.

`int` **`bitstring_size`** Constant

Number of bits contained in an element of type `bitstring_t`.

`bitstring_t` **`bitstring_msb`** Constant

Mask selecting the most significant bit of an element of type `bistring_t`.

`bitstring_t*` **`bitstring_alloc`** (`int` *n*) Function

Allocates an array of size *n* (the size is the number of `bitstring_t` and *not* the number of bits (see `bitindex_size`).

`void` **`bitstring_free`** (`bitstring_t*` *b*) Function

Frees the array.

`void` **`bitstring_clear`** (`bitstring_t*` *b*, `size_t` *size*) Function

Clears all bits of the array of size *size*.

`int` **`bitstring_cmp`** (`const bitstring_t*` *r1*, `const bitstring_t*` Function

r2, size_t *size*) Compares the two bitstring of size *size* lexicographically.

`void` **`bitstring_print`** (`const bitstring_t*` *b*, `size_t` *size*) Function

Prints the array as a string of bits.

`int` **`bitstring_get`** (`const bitstring_t*` *b*, `bitindex_t` *ix*) Function

Return the value of the bit of array *b* referenced by the index *ix*. A null value represents a bit set to zero, a non-null value a bit set to one.

`void` **`bitstring_set`** (`const bitstring_t*` *b*, `bitindex_t` *ix*) Function

Sets to one the bit of array *b* referenced by the index *ix*.

`void` **`bitstring_clr`** (`const bitstring_t*` *b*, `bitindex_t` *ix*) Function

Clears the bit of array *b* referenced by the index *ix*.

5.7 Module Satmat

Saturation matrices allows the user to know which generator saturates which constraint, i.e. their scalar product is zero. This means that the generator belongs to the constraint. Such matrices are obtained from polyhedra (see next section) in two versions: either columns are indexed by constraints and rows by generator, either the opposite. A cleared bit indicates saturation, and bit set to one indicates that the generator only satisfies the constraint.

5.7.1 The type satmat_t

`satmat_t` — datatype

Saturation matrices are represented as an array of rows, each of which is a string of bits. A row `i` of the matrix `satmat_t* sat` is accessed with `sat_>p[i]`, of type `bitstring_t*`.

For information, the private datatype is defined as follows:

```
typedef struct satmat_t {
  bitstring_t** p;
  int nbrows;
  int nbcolumns;
  bitstring_t* p_init;
} satmat_t;
```

5.7.2 Basic Operations on saturation matrices

`satmat_t*` **`satmat_alloc`** (`int` *nr*, `int` *nc*) — Function

Allocates a saturation matrix with *nr* rows and *nc* elements of type `bitindex_t` per row (see `bitindex_size`).

`void` **`satmat_free`** (`satmat* sat`) — Function

Frees the saturation matrix.

`satmat_t*` **`satmat_copy`** (`const satmat* sat`) — Function

Makes a copy of the saturation matrix.

`void` **`satmat_print`** (`const satmat* sat`) — Function

Prints the saturations matrix on standard output.

`void` **`satmat_clear`** (`satmat_t*` *sat*) — Function

Clears all bits of the saturation matrix.

5.7.3 Bit manipulations on saturation matrices

`bitstring_t` **`satmat_get`** (`const satmat_t*` *sat*, `int` *i*, `bitindex_t` *jx*) — Function

Return the bit corresponding to the row *i* and the column *jx*. If the result is null, the bit is set to zero, a non null value indicates a bit set to one.

void **satmat_set** (satmat_t* *sat*, int *i*, bitindex_t *jx*) Function

Sets to one the bit of row *i* and column *jx*.

void **satmat_set** (satmat_t* *sat*, int *i*, bitindex_t *jx*) Function

Sets to zero the bit of row *i* and column *jx*.

5.7.4 Matrix manipulations

satmat_t* **satmat_transpose** (const satmat_t* *org*, int *nbcols*) Function

Return the transposed matrix of *org*. *nbcols* is the number *of bits* to be transposed for each row of *org* (and is the number of rows of the result).

void **satmat_exch_rows** (satmat_t* *sat*, int *l1*, int *l2*) Function

Exchanges rows *l1* and *l2* of the saturation matrix.

void **satmat_sort_rows** (satmat_t* *sat*) Function

Sorts the rows of the saturation matrix (lexicographic order).

int **satmat_index_in_sorted_rows** (bitstring_t* *satline*, satmat_t* *sat*) Function

Does *satline* correspond to a row of *sat* ? If it is the case, return the index of *satline* in *sat*, otherwise return -1. The size of *satline* and *sat* is supposed to be the same, with the convention that unused bits of *sat* are supposed to be zero.

5.8 Module Poly

A polyhedron is represented by a structure whose all elements should considered as private. Operations on polyhedra and possibly vectors and matrices assumes compatible dimensions of the parameters. This is checked when possible (we cannot check the compatibility of vectors with matrices or polyhedra).

A polyhedron can be in minimal form or not. In the first case, the constraints, the generators, the saturation matrix, and the dimension of equality and lineality space are available. Otherwise, only the constraints or the generators are available.

5.8.1 Basic Operations on polyhedra

`void` **`poly_free`** (`poly_t*` *po*) Function

Frees the polyhedron and finalize referenced elements.

`poly_t*` **`poly_copy`** (`const poly_t*` *po*) Function

Makes a copy of the polyhedron. Referenced elements are recursively duplicated.

`void` **`poly_print`** (`const poly_t*` *po*) Function

Prints the polyhedron on standard output.

`void` **`poly_minimize`** (`const poly_t*` *po*) Function

Computes the minimal representation of the polyhedron. Once minimized, both constraints and generators are available, as such as the saturation matrix and the dimension of equality and lineality spaces.

`void` **`poly_canonicalize`** (`const poly_t*` *po*) Function

If `polka_strict` is false, same effect as `poly_minimize`, but ensures also normalization of equalities and lines spaces (with Gauss elimination). Otherwise, normalizes the strict constraints of *po* and performs minimization on the new set of constraints (also with normalization of equalities and lines spaces). This allows to remove constraints which are redundant considering the special meaning of the epsilon dimension.

5.8.2 Basic Constructors for polyhedra

`poly_t*` **`poly_empty`** (`int` *dim*) Function

Creates an empty polyhedron of affine dimension *dim*, in minimized form.

`poly_t*` **`poly_universe`** (`int` *dim*) Function

Creates an universe polyhedron of affine dimension *dim*, in minimized form.

`poly_t*` **`poly_of_constraints`** (`matrix_t*` *mat*) Function

Creates a polyhedron defined by the constraints stored in *mat*. The matrix *mat* will be referenced by the result, so don't touch it any more after the call. The dimension of the polyhedron is equal to the number of columns of the matrix minus `polka_dec`. The returned polyhedron is not in a minimal form.

It's the user responsability to put in the matrix the constraint \xi> = 0 or the constraints \xi> = \epsilon> = 0, if they are not implied by the other constraints. If you are not sure of what you are doing, use rather `poly_universe` and `poly_add_constraints`.

`poly_t*` **`poly_of_frames`** (`matrix_t*` *mat*) Function

Creates a polyhedron defined by the generators stored in *mat*. The same remarks as above holds. The defined polyhedra have to be included in \xi> = 0 or \xi> = \epsilon> = 0. If you are not sure of what you are doing, use rather `poly_empty` and `poly_add_frames`.

5.8.3 Access functions for polyhedra

Among the functions below, the first three don't imply any computation. The obtention of a saturation matrix, as the four last functions, needs minimal form and performs minimization if necessary.

`const matrix_t*` **`poly_constraints`** (`const poly_t*` *po*) Function

Return the matrix of constraints of the polyhedron, which is referenced by it, when this matrix is available, or else the null pointer. Don't modify the matrix neither free it, as it is pointed by *po*. The obtained set of constraints may not be minimal.

`const matrix_t*` **`poly_frames`** (`const poly_t*` *po*) Function

Return the matrix of generators of the polyhedron, if available, else the null pointer. The same remarks as above holds.

`const satmat_t*` **`poly_satC`** (`const poly_t*` *po*) Function

Return the saturation matrix, whose rows are indexed by generators and columns by constraints. The same remarks as above holds.

`const satmat_t*` **`poly_satF`** (`const poly_t*` *po*) Function

Return the saturation matrix, whose rows are indexed by constraints and columns by generators. The same remarks as above holds.

`int` **`poly_dimension`** (`const poly_t*` *po*) Function

Return the (affine) dimension of the polyhedron (i.e., without taking into accound the additional columns of vectors and matrices).

`int` **`poly_nbequations`** (`const poly_t*` *po*) Function

Return the dimension of the equality space, i.e. the number of linearly independant equations satisfied by the polyhedron. Require minimization.

`int` **`poly_nblines`** (`const poly_t*` *po*) Function

Return the dimension of the lineality space, i.e. the number of linearly independant lines included in the polyhedron. Require minimization.

`int` **`poly_nbconstraints`** (`const poly_t*` *po*) Function

Return the number of constraints in minimal form.

`int` **`poly_nbframes`** (`const poly_t*` *po*) Function

Return the number of generators in minimal form.

5.8.4 Predicates on polyhedra

`bool` **`poly_is_minimal`** (`const poly_t*` *po*) Function

Says if the polyhedron is minimized. Doesn't imply any computation.

`bool` **`poly_is_empty`** (`const poly_t*` *po*) Function

Tests if the polyedron is empty. Can imply minimization.

`bool` **`poly_is_universe`** (`const poly_t*` *po*) Function

Tests if the polyhedron is the universe one. Imply minimization.

`tbool` **`poly_is_empty_lazy`** (`const poly_t*` *po*) Function

This function tests emptiness without minimize the polyhedron. As a result, the answer can be: I don't know (`tbool_bottom`).

`tbool` **`poly_versus_constraint`** (`const poly_t*` *po*,`const pkint_t*` *tab*) Function

Tests the relation between the polyhedron and the constraint, which must have the same dimension. If the constraint is an inequality the result has the following meaning:

- `tbool_top`: all frames belongs to the hyperplane defined by the constraint;
- `tbool_true`: all frames satisfies the constraint but do not verify the preceding property (the polyhedron is on the positive side of the constraint);
- `tbool_false`: no frame satisfies the constraint (the polyhedron is on the strictly negative side of the constraint);
- `tbool_bottom`: the constraint splits the polyhedron.

In the case where the constraint is an equality, the two possible results are `tbool_top` and `tbool_bottom`.

`bool` **`poly_is_generator_included_in`** (`const pkint_t*` *tab*, `const poly_t*` *po*) Function

Tests if a generator is included in the polyhedron. The function may minimize the polyhedron in order to get its constraints.

`bool` **`poly_is_included_in`** (`const poly_t*` *pa*, `const poly_t*` *pb*) Function

Tests the inclusion of the first polyhedron in the second one. This function may minimize the two polyhedra.

`bool` **`poly_is_equal`** (`const poly_t*` *pa*, `const poly_t*` *pb*) Function

Tests the equality of two polyhedra. Requires minimal form for both polyhedra.

5.8.5 Intersection and Convex Hull of polyhedra

All functions described in this paragraph are functional.

Strict version

These functions return polyhedra in minimal form and their parameters are minimized when it is not already the case.

`poly_t*` **`poly_intersection`** (`const poly_t*` *pa*, `const poly_t*` *pb*) Function

Return the intersection of the two polyhedra. The function choose one of the polyhedron as starting point, and adds to it the constraints of the other one. One chooses the polyhedron that have the greatest number of equalities, or else the greatest number of constraints.

`poly_t*` **`poly_intersection_array`** (`const poly_t* const*` *po*, `int` *size*) Function

Return the intersection of the (non-empty) array of polyhedra *po* of size *size*. The function chooses one of the polyhedron as starting point, and adds to it the constraints of all the other ones. One choose the polyhedron that have the greatest number of equalities, or else the greatest number of constraints.

`poly_t*` **`poly_add_constraints`** (`const poly_t*` *po*, `matrix_t*` *mat*) Function

Return the intersection of the polyhedron with the set of constraints given by the matrix. The matrix may be sorted by the function.

`poly_t*` **`poly_add_constraint`** (`const poly_t*` *po*, `const pkint_t*` *tab*) Function

Return the intersection of the polyhedron with the constraint given by *tab*.

`poly_t*` **`poly_convex_hull`** (`const poly_t*` *pa*, `const poly_t*` *pb*) Function

Return the convex hulls of the two polyhedra. The function choose one of the polyhedron as starting point, and adds to it the generators of the other one. One chooses the polyhedron that have the greatest number of lines, or else the greatest number of generators.

`poly_t*` **`poly_convex_hull_array`** (`const poly_t* const*` *po*, `int` *size*) Function

Return the convex hull of the (non-empty) array of polyhedra *po* of size *size*. The function choose one of the polyhedron as starting point, and adds to it the generators of all the other ones. One chooses the polyhedron that have the greatest number of lines, or else the greatest number of generators.

`poly_t*` **`poly_add_frames`** (`const poly_t*` *po*, `matrix_t*` *mat*) Function

Return the convex hull of the polyhedron and the set of generators given by the matrix. The matrix may be sorted by the function.

`poly_t*` **`poly_add_frame`** (`const poly_t*` *po*, `const pkint_t*` *tab*) Function

Return the convex hull of the polyhedron and the generator given by *tab*.

Lazy version

These functions are the lazy version of the preceding ones. They return polyhedra in non minimal form and their parameters are minimized only if it is necessary.

poly_t* **poly_intersection_lazy** (const poly_t* *pa*, const poly_t* *pb*) Function
poly_t* **poly_intersection_array_lazy** (const poly_t* const* *po*, int *size*) Function
poly_t* **poly_add_constraints_lazy** (const poly_t* *po*, matrix_t* *mat*) Function

poly_t* **poly_add_constraint_lazy** (const poly_t* *po*, const pkint_t* *tab*) Function
poly_t* **poly_convex_hull_lazy** (const poly_t* *pa*, const poly_t* *pb*) Function
poly_t* **poly_convex_hull_array_lazy** (const poly_t* const* *po*, int *size*) Function
poly_t* **poly_add_frames_lazy** (const poly_t* *po*, matrix_t* *mat*) Function
poly_t* **poly_add_frame_lazy** (const poly_t* *po*, const pkint_t* *tab*) Function

5.8.6 Change of dimension of polyhedra

At the end

poly_t* **poly_add_dimensions_and_embed** (const poly_t* *po*, int *dimsup*) Function

This function adds *dimsup* dimension to the given polyhedron and embed it in the new space. The new dimensions are the last one, i.e. the corresponding coefficients are in the last columns of vectors and matrices. Preserves the minimality of the polyhedron: if the parameter is in minimal form, then its the case for the result, otherwise not.

poly_t* **poly_add_dimensions_and_project** (const poly_t* *po*, int *dimsup*) Function

This function adds *dimsup* dimension to the given polyhedron and project it onto the old dimensions, i.e. the constraints $x\_i = 0$ are satisfied for the new dimensions i. The new dimensions are the last one, i.e. the corresponding coefficients are in the last columns of vectors and matrices. Preserves the minimality of the polyhedron.

poly_t* **poly_remove_dimensions** (const poly_t* *po*, int *dimsup*) Function

This function projects the given polyhedron onto the $po->dim-dimsup$ first dimensions, and eliminates the last coefficients. Minimality is lost. The parameter may be minimzed in order to get its generators.

Anywhere

See Section 5.4.6 [Change of dimension of vectors], page 19, for more details about the meaning of arrays of elements of type dimsup_t.

poly_t* **poly_add_dimensions_and_embed_multi** (const poly_t* *po*, const dimsup_t* *tab*, size_t *size*) Function

Adds new dimensions in the polyhedron *po*, according to the array *tab* of size *size*, and embed it in the new space. Preserves the minimality of the polyhedron.

poly_t* **poly_add_dimensions_and_project_multi** (const poly_t* *po*, const dimsup_t* *tab*, size_t *size*) Function

Adds new dimensions in the polyhedron *po*, according to the array *tab* of size *size*, and project it onto the old dimensions. Preserves the minimality of the polyhedron.

poly_t* **poly_remove_dimensions_multi** (const poly_t* *po*, const dimsup_t* *tab*, size_t *size*) Function

Deletes some dimensions in the polyhedron *po*, according to the array *tab* of size *size*. Minimality is lost.

Change of dimensions together with permutation

void **poly_add_permute_dimensions_and_embed** (const poly_t* *po*, int *dimsup*, const int* *permutation*) Function

dimsup is supposed to be positive or null. Add first *dimsup* dimensions at the end (the corresponding coefficients are in the last columns of vectors and matrices), embed the polyehdron into the new space and then apply the permutation *permutation* of size `poly_dim(po)+dimsup` to the dimensions of the polyhedron.

The permutation *permutation* defines a permutation (i.e., a bijection) from `[0..(poly_dim(po)+dimsup)-1]` to itself. BE CAUTIOUS: value `0` in the permutation means columns *polka_dec*.

void **poly_add_permute_dimensions_and_project** (const poly_t* *po*, int *dimsup*, const int* *permutation*) Function

dimsup is supposed to be positive or null. Add first *dimsup* dimensions at the end (the corresponding coefficients are in the last columns of vectors and matrices), project the polyehdron onto the old dimensions, and then apply the permutation *permutation* of size `poly_dim(po)+dimsup` to the dimensions of the polyhedron.

The permutation *permutation* defines a permutation (i.e., a bijection) from `[0..(poly_dim(po)+dimsup)-1]` to itself. BE CAUTIOUS: value `0` in the permutation means columns *polka_dec*.

void **poly_permute_remove_dimensions** (const poly_t* *po*, int *dimsup*, const int* *permutation*) Function

dimsup is supposed to be positive or null. Permute first the dimensions of the polyhedron according to the permutation *permutation* of size `poly_dim(po)`, then remove the *dimsup* last dimensions and project the polyhedron onto the reduced space.

The permutation *permutation* defines a permutation (i.e., a bijection) from `[0..poly_dim(po)-1]` to itself. BE CAUTIOUS: value `0` in the permutation means columns *polka_dec*.

5.8.7 Linear transformations on polyhedra

Single variable/expression

poly_t* **poly_assign_variable** (const poly_t* *po*, int *rank*, const pkint_t* *tab*) Function

This function applies to the polyhedron the linear transformation $x'\_rank := \backslash sum\_\{i = 0\}\hat{}\{dim - 1\}a\_i/dx\_i{+}b/d$ with *rank* the rank of the variable (rank 0 corresponding to the first normal variable) and *tab*=[d,b,[0],a_0,...,a_{dim-1}]. *po* may be minimized in order to obtain its generators. Minimality is then preserved if the transformation is inversible.

poly_t* **poly_substitute_variable** (const poly_t* *po*, int *rank*, const pkint_t* *tab*) Function

This function applies to the polyhedron the linear substitution $x'\_rank < -\backslash sum\_\{i = 0\}\hat{}\{dim - 1\}a\_i/dx\_i{+}b/d$ with *rank* denoting the rank of the variable (rank 0 corresponding to the first normal variable) and *tab*=[d,b,[0],a_0,...,a_{dim-1}]. *po* may be minimized in order to obtain its constraints. Minimality is then preserved if the transformation is inversible.

The two following functions are kept for compatibility. The variable to be assigned or substituted is not denoted by its rank, but by its index in vectors: we have the relationship index = rank + polka_dec.

poly_t* **poly_assign_variable_old** (const poly_t* *po*, int *index*, const pkint_t* *tab*) Function

poly_t* **poly_substitute_variable_old** (const poly_t* *po*, int *index*, const pkint_t* *tab*) Function

Same as poly_assign_variable and poly_substitute_variable, but with *index* denoting the index of the variable in vectors.

Several variables/expressions

poly_t* **poly_assign_variables** (const poly_t* *po*, const equation_t* *eqn*, size_t *size*) Function

Computes the image of *po* by the parallel assignment of eqn[i].var by eqn[i].expr, for *i* between 0 and size-1. The array *eqn* is supposed to be sorted w.r.t. the field .var. You may use the function sort_equations to ensure this (see 'polka.h').

poly_t* **poly_substitute_variables** (const poly_t* *po*, const equation_t* *eqn*, size_t *size*) Function

Computes the image of *poly* by the parallel substitution of eqn[i].expr by eqn[i].expr, for *i* between 0 and size-1. The array *eqn* is supposed to be sorted w.r.t. the field .var. You may use the function sort_equations to ensure this (see 'polka.h').

5.8.8 Widening operators on polyhedra lattice

These two operations are parametrized by the widening mode, see Section 5.3.2 [Widening mode], page 14.

`poly_t*` **`poly_widening`** (`const poly_t*` *pa*, `const poly_t*` *pb*) Function

This function implements the standard widening operator defined in *cousot78*. Minimal form is required for the two polyhedra. The returned polyhedron is defined by those constraints which are satisfied by both parameter polyhedra.

`poly_t*` **`poly_limited_widening`** (`const poly_t*` *pa*, `const poly_t*` *pb*, `matrix_t*` *mat*) Function

This function implements the version of the widening operator parametrized by a set of constraints. It adds to the polyhedron obtained by the standard widening the constraints of the matrix which are satisfied by both parameter polyhedra *polka:fmsd:97*. The parameter matrix may be sorted.

5.8.9 Closure operation

`poly_t*` **`poly_closure`** (`const poly_t*` *poly*) Function

`poly_t*` **`poly_closure_lazy`** (`const poly_t*` *poly*) Function

If `polka_strict` is true, these functions compute the closure (in the topological sense) of the argument polyhedron, i.e., the smallest polyhedron with loose constraints only containing the argument. If `polka_strict` is false, they return a copy of the argument polyhedron. The parameter polyhedron may be minimized.

The two functions differ by their strict/lazy implementation.

5.9 C Example

```
#include  <stdio.h>
#include "polka.h"
#include "pkint.h"
#include "poly.h"

void essai1()
{
  poly_t *PUniv, *P1, *P2, *P3;
  bool result;
  matrix_t* conP1 = matrix_alloc(2,4,false);
  /*   x >= 2  */
  pkint_set_si(conP1->p[0][0], 1);
  pkint_set_si(conP1->p[0][1], -2);
  pkint_set_si(conP1->p[0][2], 0);
  pkint_set_si(conP1->p[0][3], 1);

  /*   x > 1  */
  pkint_set_si(conP1->p[1][0], 1);
  pkint_set_si(conP1->p[1][1], -1);
  pkint_set_si(conP1->p[1][2], -1);
  pkint_set_si(conP1->p[1][3], 1);

  PUniv = poly_universe(1);
  P1 = poly_add_constraints(PUniv, conP1);
```

```
  poly_print(P1);
  poly_minimize(P1);
  poly_print(P1);

  P2 = poly_add_constraint(PUniv, conP1->p[0]);
  P3 = poly_add_constraint(PUniv, conP1->p[1]);

  result = poly_is_included_in(P2, P3);
  if(result == true)
    printf("\n\npoly[x>=2] is included in poly[x>1]\n");
  else
    printf("\n\npoly[x>=2] is not included in poly[x>1]\n");
}

void essai2()
{
  poly_t *PUniv, *PZero, *P1, *P2, *P3;
  const matrix_t* rayP1;
  matrix_t* rayP1b;
  bool result;
  int i;

  int nbdim=12;

  PUniv = poly_universe(nbdim);

  /* P1 */
  matrix_t* conP1 = matrix_alloc(nbdim*2,nbdim+3,false);
  for (i=0;i<nbdim; i++){
    pkint_set_si( conP1->p[2*i][0], 1);
    pkint_set_si( conP1->p[2*i][polka_cst], 1000000000);
    pkint_set_si( conP1->p[2*i][polka_dec+i], -1  );
    pkint_set_si( conP1->p[2*i+1][0], 1);
    pkint_set_si( conP1->p[2*i+1][polka_cst], 1000000000);
    pkint_set_si( conP1->p[2*i+1][polka_dec+i], 1    );
  }
  printf ("Here1a\n");
  P1 = poly_add_constraints(PUniv, conP1);
  poly_minimize(P1);
  matrix_free(conP1);
  printf ("Here1b\n");

  /* PZero */
  conP1 = matrix_alloc(nbdim,nbdim+3,false);
  for (i=0;i<nbdim; i++){
    pkint_set_si( conP1->p[i][polka_dec+i], 1);
  }
  PZero = poly_add_constraints(PUniv, conP1);
  poly_minimize(PZero);
  matrix_free(conP1);
```

```
  poly_free(PUniv);

  /* */
  rayP1 = poly_frames(P1);

  for (i=0; i<5; i++){
    printf ("Here2a %d\n",i);
    rayP1b = matrix_copy(rayP1);
    matrix_qsort_rows(rayP1b);
    rayP1b->_sorted = true;
    printf ("Here2b %d\n",i);
    P2 = poly_add_frames(PZero,rayP1b);
    printf ("Here2c %d\n",i);
    matrix_free(rayP1b);
    poly_free(P2);
  }
  poly_free(P1);
  poly_free(PZero);
}

int main(int argc, char** argv)
{
  polka_initialize(true,20,10000);

  essai2();

  return 0;
}
```

6 OCaml Library

This chapter describes the OCaml API of Polka. The reader is supposed to have first read the documentation of the C library.

6.1 Organization of the OCaml interface

Not all functions of type C library are interfaced; for instance, you can't manipulate saturation matrices in OCaml (buf of course, if someone asks me this feature, I can add it). The three datatypes that are interfaced are vectors, matrices, and polyhedra, and there is a corresponding module for each of these datatypes. All objects are properly finalized, thanks to the use of "custom" blocks.

The interface was developped with the CamlIDL tool. Each '`.idl`' file produces a '`_caml.c`', a '`.mli`' and a '`.ml`' file. Each file is represents a module.

The interface is decomposed as follows:

- Files '`polka_lexer.mli`', '`polka_lexer.mll`' and '`polka_parser.mly`' implements pretty-input of vectors and matrices;
- Files '`polka_caml.h`', '`polka_caml.c`', '`polka.mli`' and '`polka.ml`' take care of global stuff and defines the OCaml module `Polka`.
- '`vector.idl`' defines the module `Vector`.
- '`matrix.idl`' defines the module `Matrix`.
- '`poly.idl`' defines the module `Poly`.

When invalid arguments are given to some function (incompatible dimensions, out-of-bound access, ...), an `Invalid_argument` exception is usually thrown.

We provide a unified type for coefficients, which is the OCaml native integers. Converting to OCaml native integers coefficients as used in the C library, which can be `long int`, `long long int` or `mpz_t`, may produce overflow. In this case, an exception `Polka.Overflow(str)` is raised, carrying the string representation of the coefficient in base 10. This concerns only the functions `Vector.get` and `Matrix.get`.

6.2 Polka

This modules defines datatypes, some global read-only variables, and library initialization and finalization functions.

Exceptions and Datatypes

`Overflow` of string — Exception

Exception raised when an overflow occurs when converting from internal type `pkint_t` in the C library to OCaml native integer. The exception carries the string representation of the number in base 10.

`dimsup` — Datatype

```
type dimsup = {
  pos: int;
  nbdims: int;
}
```

Data-type for insertion and deletion of columns in vectors, matrices, and polyhedra.

Initialization and finalization funtions

`initialize` `: bool -> int -> int -> unit` — Function

`initialize strict maxdims maxrows` initializes internal data-structures and global variables of the library:

- `strict` indicates wether strict inequalities are enabled or not;
- `maxdims` is the maximum number of dimensions allowed in polyhedra; the maximum number of columns allowed in vectors and matrices is thus equal to this number plus `polka_dec` (see below);
- `maxrows` is the maximum number of rows or vectors allowed in matrices.

Set variables `strict` and `dec` (see below).

`set_gc` `: int -> unit` — Function

Sets the ratio *used/max* of the OCaml garbage collector for all the abstract objects allocated by Polka. The default is 2ˆ26, which means that there is at most 64MB of unreclaimed memory.

`finalize` `: unit -> unit` — Function

Free internal data-structure used in the libary.

`set_widening_affine` `: unit -> unit` — Function

`set_widening_linear` `: unit -> unit` — Function

Select respectively the ‘affine mode” or the “linear mode” for the widening operation on polyhedra (see Section 5.3.2 [Widening mode], page 14).

Variables

strict : bool ref Variable

Indicate if strict inequalities are enabled or not.

dec : int ref Variable

Index of the first “real” dimension in vectors and matrices.

6.3 Vector

t Datatype

Abstract datatype for vectors.

6.3.1 Basic Operations on OCaml vectors

make : int -> t Function

`make size`

Return a vector of size `size` with all coefficient initialized to 0. `size=0` is accepted.

copy : t -> t Function

Return a copy of the vector.

_print : t -> unit Function

Print in a unformatted way on the C standard output the vector.

get : t -> int -> int Function

`get vec index`

Return the corresponding coefficient of the vector; in case of overflow during the conversion from `pkint_t` to `int`, raise exception `Polka.Overflow`.

get_str10 : t -> int -> string Function

`get_str10 vec index`

Return the string representation in base 10 of the corresponding coefficient of the vector.

set : t -> int -> int -> unit Function

`set vec index val`

Store the value `val` in the corresponding coefficient of the vector.

set_str10 : t -> int -> string -> unit Function

`set_str10 vec index val`

Store the value `val` represented in base 10 in the corresponding coefficient of the vector.

length : t -> int Function

Return the length of the vector.

6.3.2 Comparison & Hashing of OCaml vectors

compare : t -> t -> int Function

Compare the two vectors as does `vector_compare`. In the case where the vectors have different lengths, compare the length.

compare_expr : t -> t -> int Function

Compare the two vectors as does `vector_compare_expr`. In the case where the vectors have different lengths, compare the length.

hash : t -> int Function

Compute a hash key for vectors, as does `vector_hash`.

6.3.3 Normalization of OCaml vectors

norm : t -> unit Function

norm_expr : t -> unit Function

Normalize in place the vector, respectively as a constraint or a generator, or as an affine expression.

6.3.4 Algebraic Operations on OCaml vectors

product : t -> t -> int Function

Scalar product. The first coefficient is ignored. Raise `Invalid_argument` when the sizes are different.

product_strict : t -> t -> int Function

As the previous function, but the *epsilon*-coefficients are also ignored.

add_expr : t -> t -> t Function

sub_expr : t -> t -> t Function

Respectively add and substract two vectors of same size considered as affine expressions. Raise `Invalid_argument` when the sizes are different.

scale_expr : int -> int -> t -> t Function

`scale num den vec`

Scale the vector considered as an affine expression with the rational number `num/den` and returns the result.

6.3.5 Change of dimensions on OCaml vectors

At the end

add_dims : int -> t -> t Function

`add_dims n vec`

Add or remove `(abs n)` dimensions to the vector, depending on the sign of `n`. Dimensions are added or removed at the end of the vector.

Anywhere

add_dims_multi : t -> Polka.dimsup array -> t Function

Same as C function `vector_add_dimensions_multi`.

del_dims_multi : t -> Polka.dimsup array -> t Function

Same as C function `vector_remove_dimensions_multi`.

Change of dimensions together with permutation

add_permute_dims : `t -> int -> int array -> t` Function

Same as C function `vector_add_permute_dimensions`.

permute_del_dims : `t -> int -> int array -> t` Function

Same as C function `vector_permute_remove_dimensions`.

6.3.6 Miscellaneous on OCaml vectors

is_positivity_constraint : `t -> bool` Function

Tell wether the vector is the positivity constraint as does `vector_is_positivity_constraint`.

is_strictness_constraint : `t -> bool` Function

Tell wether the vector is the strictness constraint as does `vector_is_strictness_constraint`.

is_dummy_constraint : `t -> bool` Function

Tell wether the vector is the positivity or the strictnes constraint, as does `vector_is_dummy_constraint`.

6.3.7 Input & Output of OCaml vectors

These functions use the pretty input/output facilities described in module `Polka`.

Printing and output functions takes as first paramater a function of type `(int -> string)` associating variable names to variable *ranks*. By definition, `variable index = polka_dec + variable rank`.

print_constraint : `(int -> string) -> Format.formatter -> t -> unit` Function

Pretty-print on the given formatter the vector considered as as a constraint.

print_frame : `(int -> string) -> Format.formatter -> t -> unit` Function

Pretty-print on the given formatter the vector considered as a generator. The output has the following meaning:

- `V:0` represents the origin vertex;
- `V:3x+4y` represents the vertex defined by $x = 3, y = 4$;
- `V:(3x+4y)/2` represents the vertex defined by $x = 3/2, y = 2$;
- `R:3x+4y` represents the ray $3.i\_x+4.i\_y$;
- `L:3x+4y` represents the line whose direction is given by $3.i\_x+4.i\_y$.

print_expr : `(int -> string) -> Format.formatter -> t -> unit` Function

Pretty-print on the given formatter the vector considered as an affine expression.

The next functions are input functions: they converts strings to vectors. They as first paramater a function of type `(string -> int)` associating variable ranks to variable names, and as second parameter the *dimension* of the returned vector. By definition, `length = polka_dec + dimension`.

`of_expr` `: (string -> int) -> int -> string -> t` Function

Make a vector from a string parsed as an affine expression.

The syntax of an expression is
`(+|-) a_x/b_x x (+|-) ... (+|-) a_y/b_y y`
where the a's are numerators, b's are *optional* denominators, and $x, y, ...$ are variable names. If there is no name, then the constant coefficient is concerned, and if there is no numerator, 1 is assumed. An example is `x+2y-1/2z-1/3w`. There is no way to reference the special dimension *epsilon* (and it is unnecessary in this case any way).

`of_constraint` `: (string -> int) -> int -> string -> t` Function

Make a vector from a string parsed as a constraint.

The syntax of a constraint is `<expr> (>|>=|=|<=|<) <expr>`.

`of_frame` `: (string -> int) -> int -> string -> t` Function

Make a vector from a string parsed respectively as a constraint or as a generator.

The syntax of a generator is the following one:

- `V:0` represents the origin vertex.
- `V:(+|-)a_x/b_x x (+|-) ... (+|-) a_y/b_y y` represents the vertex defined by `x=(+|-)a_x/b_x, ..., y=(+/-)a_y/b_y`;
- `R:(+|-)a_x/b_x x (+|-) ... (+|-) a_y/b_y y` represents the ray `(+|-)a_x/b_x.i_x (+|-) ... (+|-) a_y/b_y.i_y`;
- `L:(+|-)a_x/b_x x (+|-) ... (+|-) a_y/b_y y` represents the line whose direction is given by `(+|-)a_x/b_x.i_x (+|-) ... (+|-) a_y/b_y.i_y`.

Non-zero constants have no sense such expressions. There is unfortunately no way to specify vertices with non-zero *epsilon* components. However, the meaning of such vertices is difficult to get !

6.4 Matrix

t Datatype

Abstract datatype for matrices.

equation Datatype

Datatype for parallel assignements/substitutions.

```
type equation = {
  var: int;
  expr: Vector.t;
}
```

6.4.1 Basic Operations on OCaml matrices

make : int -> int -> t Function

make nbrows nbcols

Return a matrix of size nbrows X nbcolumns with all coefficient initialized to 0. A null value for any parameter is accepted.

copy : t -> t Function

Return a copy of the matrix.

_print : t -> unit Function

Print in a unformatted way on the C standard output the matrix.

get : t -> int -> int -> int Function

get vec row col

Return the corresponding coefficient of the matrix; in case of overflow during the conversion from pkint_t to int, raise exception Polka.Overflow.

get_str10 : t -> int -> int -> string Function

get vec row col

Return the string representation in base 10 the corresponding coefficient of the matrix.

set : t -> int -> int -> int -> unit Function

set vec row col val

Store the value val in the corresponding coefficient of the matrix.

set_str10 : t -> int -> int -> string -> unit Function

set vec row col val

Store the value val represented in base 10 in the corresponding coefficient of the matrix.

get_row : t -> int -> Vector.t Function

get_row mat row

Return a vector corresponding to the row row of the matrix mat. There is no sharing in memory between the result and the argument matrix.

set_row : t -> int -> Vector.t -> unit Function

set_row mat row vec

Store in the row row of the matrix mat the vector vec. The coefficients are copied (no sharing memory).

nbrows : t -> int Function

Return the number of rows of the matrix.

nbcolumns : t -> int Function

Return the number of columns of the matrix.

6.4.2 Comparison & Hashing of OCaml matrices

compare : t -> t -> int Function
compare_sort : t -> t -> int Function

Compare the two matrices, as does respectively matrix_compare and matrix_compare_sort.

hash : t -> int Function
hash_sort : t -> int Function

Compute a hash key for the matrix, as does respectively matrix_hash and matrix_hash_sort.

6.4.3 Sorting & Merging of OCaml matrices

sort_rows : t -> unit Function

Sort in place the rows of the matrix.

merge_sort : t -> t -> t Function

Merge the matrices as does matrix_merge_sort.

6.4.4 Linear transformations on matrices

Be cautious: unlike the C version, dimensions are here referenced by their rank (from 0 up to the number of dimensions) and not their index in vectors and matrices (we have the relationship index = rank + polka_dec).

assign_var : t -> int -> Vector.t -> t Function
assign_vars : t -> equation array -> t Function

Linear transformation of a dimension (resp. a set of dimensions) by linear expression(s). The array of equations is supposed to be sorted w.r.t. the field .var.

substitute_var : t -> int -> Vector.t -> t Function
substitute_vars : t -> equation array -> t Function

Substitution of a dimension (resp. a set of dimensions) by linear expression(s). The array of equations is supposed to be sorted w.r.t. the field .var.

6.4.5 Change of dimension of OCaml matrices

At the end

add_dims : t -> int -> t — Function

Same as C function `matrix_add_dimensions`.

Anywhere

add_dims_multi : t -> Polka.dimsup array -> t — Function

Same as C function `matrix_add_dimensions_multi`.

del_dims_multi : t -> Polka.dimsup array -> t — Function

Same as C function `matrix_remove_dimensions_multi`.

Change of dimensions together with permutation

add_permute_dims : t -> int -> int array -> t — Function

Same as C function `matrix_add_permute_dimensions`.

permute_del_dims : t -> int -> int array -> t — Function

Same as C function `matrix_permute_remove_dimensions`.

6.4.6 Miscellaneous on OCaml matrices

is_row_dummy_constraint : t -> int -> bool — Function

`is_row_dummy_constraint mat row`

Tell wether the row `row` of the matrix `mat` represents the positivity or the strictness constraint.

6.4.7 Input & Output of OCaml matrices

This functions use the pretty input/output facilities described in module `Vector`. See Section 6.3 [Vector], page 42, for more details.

print_constraints : (int -> string) -> Format.formatter -> t -> unit — Function

print_frames : (int -> string) -> Format.formatter -> t -> unit — Function

Pretty-print on the given formatter the matrix considered as respectively as a constraint or as a generator matrix.

of_lconstraints : (string -> int) -> int -> string list -> t — Function

of_lframes : (string -> int) -> int -> string list -> t — Function

Make a matrix from respectively a list of constraints or a list of generators, each string being parsed as does `Vector.of_constraint` or `Vector.of_frame`.

6.5 Poly

t Datatype

Abstract datatype for convex polyhedra.

equation Datatype

Datatype for parallel assignements/substitutions.

```
type equation = Matrix.equation = {
  var: int;
  expr: Vector.t;
}
```

6.5.1 Constructors for OCaml polyhedra

empty : int -> t Function

universe : int -> t Function

Return respectively the empty and the universe polyhedron of the given dimension.

of_constraints : Matrix.t -> t Function

of_frames : Matrix.t -> t Function

Makes a non-minimized polyhedron from respectively a constraint matrix and a geenrator matrix. There is no sharing of values, unlike in the C functions.

minimize : t -> unit Function

Minimizes in place the polyhedron.

canonicalize : t -> unit Function

Canonicalizes in place the polyhedron (see C function).

6.5.2 Access functions for OCaml polyhedra

dim : t -> int Function

nbequations : t -> int Function

nblines : t -> int Function

nbconstraints : t -> int Function

nbframes : t -> int Function

Same semantics as the corresponding C functions.

constraints : t -> Matrix.t option Function

If the constraint matrix `mat` is available, return `Some(mat)`, otherwise return `None`. There is no sharing of elements in memory. The constraints are not necessarily in a minimal form.

frames : t -> Matrix.t option Function

Same as the previous function for generator matrix.

6.5.3 Predicates on OCaml polyhedra

is_minimal : t -> bool — Function
is_empty : t -> bool — Function
is_universe : t -> bool — Function
is_empty_lazy : t -> bool — Function
is_universe_lazy : t -> bool — Function

Same semantics as the corresponding C functions.

constraints_available : t -> bool — Function

Is the constraint matrix available ?

frames_available : t -> bool — Function

Is the generator matrix available ?

poly_versus_constraint : t -> Vector.t -> tbool — Function
is_generaotr_included_in : Vector.t -> t -> tbool — Function
is_included_in : t -> t -> bool — Function
is_equal : t -> t -> bool — Function

Same semantics as the corresponding C functions.

6.5.4 Change of dimension of OCaml polyhedra

At the end

add_dims_and_embed : t -> int -> t — Function

Same as C function `poly_add_dimensions_and_embed`.

add_dims_and_project : t -> int -> t — Function

Same as C function `poly_add_dimensions_and_project`.

del_dims : t -> int -> t — Function

Same as C function `poly_remove_dimensions`.

Anywhere

add_dims_and_embed_multi : t -> Polka.dimsup array -> t — Function

Same as C function `poly_add_dimensions_and_embed_multi`.

add_dims_and_project_multi : t -> Polka.dimsup array -> t — Function

Same as C function `poly_add_dimensions_and_project_multi`.

del_dims_multi : t -> Polka.dimsup array -> t — Function

Same as C function `poly_remove_dimensions_multi`.

Change of dimensions together with permutation

add_permute_dims_and_embed : `t -> int -> int array -> t` Function
Same as C function `poly_add_permute_dimensions_and_embed`.

add_permute_dims_and_project : `t -> int -> int array -> t` Function
Same as C function `poly_add_permute_dimensions_and_project`.

permute_del_dims : `t -> int -> int array -> t` Function
Same as C function `poly_permute_remove_dimensions`.

6.5.5 Intersection & Convex Hull of OCaml polyhedra

inter_array : `t array -> t` Function
inter : `t -> t -> t` Function
add_constraints : `t -> Matrix.t -> t` Function
add_constraint : `t -> Vector.t -> t` Function

inter_array_lazy : `t array -> t` Function
inter_lazy : `t -> t -> t` Function
add_constraints_lazy : `t -> Matrix.t -> t` Function
add_constraint_lazy : `t -> Vector.t -> t` Function
Same as C functions `poly_intersection_XXX` and `poly_add_constraintXXX`.

union_array : `t array -> t` Function
union : `t -> t -> t` Function
add_frames : `t -> Matrix.t -> t` Function
add_frame : `t -> Vector.t -> t` Function

union_array_lazy : `t array -> t` Function
union_lazy : `t -> t -> t` Function
add_frames_lazy : `t -> Matrix.t -> t` Function
add_frame_lazy : `t -> Vector.t -> t` Function
Same as C functions `poly_convex_hull_XXX` and `poly_add_frameXXX`.

inter_list : `t list -> t` Function
union_list : `t list -> t` Function
inter_list_lazy : `t list -> t` Function
union_list_lazy : `t list -> t` Function
Versions taking lists instead of arrays.

6.5.6 Linear transformations on OCaml polyhedra

assign_var : `t -> int -> Vector.t -> t` Function
assign_vars : `t -> equation array -> t` Function
Same as C functions `poly_assign_variable` and `poly_assign_variables`. The array of equations is supposed to be sorted w.r.t. the field `.var`.

substitute_var : t -> int -> Vector.t -> t Function

substitute_vars : t -> equation array -> t Function

Same as C functions poly_substitute_variable and poly_substitute_variables. The array of equations is supposed to be sorted w.r.t. the field .var.

6.5.7 Widening operators on OCaml polyhedra

These two operations are parametrized by the widening mode, see Section 6.2 [Widening mode], page 40.

widening : t -> t -> t Function

limited_widening : t -> t -> Matrix.t -> t Function

Same as C functions

6.5.8 Closure operation on OCaml polyhedra

closure : t -> t Function

closure_lazy : t -> t Function

Same as C functions

6.5.9 Input & Output of OCaml polyhedra

This functions use the pretty input/output facilities described in module Polka.

print_constraints : Format.formatter -> t -> unit Function

Print '"empty"' if the polyhedron is empty, the constraints of the polyhedron if they are available, '"constraints not available"' otherwise.

print_frames : Format.formatter -> t -> unit Function

Print '"empty"' if the polyhedron is empty, the generators of the polyhedron if they are available, '"generators not available"' otherwise.

print : Format.formatter -> -> t -> unit Function

Combine the two previous functions.

of_lconstraints : string list -> t Function

of_lframes : string list -> t Function

Construct a polyhedron respectively from a list of constraint and from a list of generators (see the same functions in module Matrix).

6.6 PolkaIO

This module is a front-end to `Polka` and maintain a database about a set of declared variables and their rank. It is useful for an interactive use of Polka. See Section 6.7 [Example of interactive use], page 54.

`initialize` `: bool -> string list -> int -> unit` Function

`initialize strict lnames maxrows`

Initializes the library and sets up the correspondance between variables names and ranks. The relative order of variables in vectors respects the order of names in the parameter list.

`initialize` sets up among others the following values.

`nbdims` `: int ref` Variable

Number of variables in database. All the objects created by this module will be of this dimension.

`in_assoc` `: string -> int` Function
`out_assoc` `: int -> string` Function

Association between variable names and ranks.

The following I/O functions makes use of the previous correspondance functions.

`vector_of_constraint` `: string -> Vector.t` Function
`vector_of_frame` `: string -> Vector.t` Function
`vector_of_expr` `: string -> Vector.t` Function

Conversion from strings to vectors.

`matrix_of_lconstraints` `: string list -> Matrix.t` Function
`matrix_of_lframes` `: string list -> Matrix.t` Function

Conversion from list of strings to matrices.

`poly_of_lconstraint` `: string list -> Poly.t` Function
`poly_of_lframe` `: string list -> Poly.t` Function

Conversion from list of strings to polyhedra.

`vector_print_constraint` `: Format.formatter -> Vector.t -> unit` Function
`vector_print_frame` `: Format.formatter -> Vector.t -> unit` Function
`vector_print_expr` `: Format.formatter -> Vector.t -> unit` Function
`matrix_print_constraints` `: Format.formatter -> Matrix.t -> unit` Function
`matrix_print_frames` `: Format.formatter -> Matrix.t -> unit` Function
`poly_print_constraint` `: Format.formatter -> Poly.t -> unit` Function
`poly_print_frame` `: Format.formatter -> Poly.t -> unit` Function
`poly_print` `: Format.formatter -> Poly.t -> unit` Function

Printing functions.

6.7 Example of interactive use

Here is an example of interactive use. First build a custom toplevel: *make polkatopg* ⟨ret⟩ in the '`caml`' directory of the distribution. Then enter it by typing *polkatopg* ⟨ret⟩. The following program:

```
#load "polka.cma";;
open Format;;
open PolkaIO;;

#install_printer vector_print_constraint;;
#install_printer matrix_print_constraints;;
#install_printer poly_print;;

initialize true ["x";"y";"z";"w"] 1000;;
let p = Poly.universe 4;;

let c1 = vector_of_constraint "2x>y";;
let p1 = Poly.add_constraint p c1;;

let m1 = matrix_of_lconstraints ["2x>=y";"3y>=z";"5z>w";"2y>7z"];;
let p2 = Poly.add_constraints p m1;;

let pu = Poly.union p1 p2;;
let pi = Poly.inter p1 p2;;
```

will give this output:

```
Objective Caml version 3.04

# #load "polka.cma";;
# open Format;;
# open PolkaIO;;
# #install_printer vector_print_constraint;;
# #install_printer matrix_print_constraints;;
# #install_printer poly_print;;
# initialize true ["x";"y";"z";"w"] 1000;;
- : unit = ()
# let p = Poly.universe 4;;
val p : Poly.t = ({$>=0,1>=0},
  {L:w,L:z,L:y,L:x,R:$,V:0})
# let c1 = vector_of_constraint "2x>y";;
val c1 : Vector.t = 2x>y
# let p1 = Poly.add_constraint p c1;;
val p1 : Poly.t = ({$>=0,1>=0,2x>y},
   {L:w,L:z,L:-x-2y,R:-y,R:$-y,V:0})
# let m1 = matrix_of_lconstraints ["2x>=y";"3y>=z";"5z>w";"2y>7z"];;
val m1 : Matrix.t = {2x>=y,3y>=z,5z>w,2y>7z}
# let p2 = Poly.add_constraints p m1;;
val p2 : Poly.t =
  ({$>=0,1>=0,2x>=y,3y>=z,5z>w,2y>7z},
   {R:-x-2y-6z-30w,R:-w,R:7x+14y+4z+20w,R:x,R:38$-x-2y-6z-68w,V:0})
# let pu = Poly.union p1 p2;;
```

```
val pu : Poly.t = ({1>=0,$>=0,2x>=y},
   {L:w,L:z,L:x+2y,R:x,R:$,V:0})
# let pi = Poly.inter p1 p2;;
val pi : Poly.t =
  ({$>=0,1>=0,5z>w,2y>7z,3y>=z,2x>y},
   {R:7x+14y+4z+20w,R:-x-2y-6z-30w,R:-w,V:0,R:x,R:19$+9x-y-3z-34w})
#
```

Appendix A Appendices

A.1 C Datatype Index

A.2 C Variable Index

B

D

I

N

P

S

A.3 C Function Index

M

N

O

P

W

Table of Contents