The Octagon Abstract Domain Library

Version 0.9.10

Antoine Miné

mine@di.ens.fr
http://www.di.ens.fr/∼mine/oct

Semantics and Abstract Interpretation
Computer Science Lab
École Normale Supérieure, Paris, France

April 2006

Contents

Title Page 1

Table of Contents 2

License 4

1 Introduction 7
1.1 Goal . . . 7
1.2 Organization of the Package . . . 7
1.3 Building and Installing the Library . . . 8
1.4 Using the Library . . . 9
1.5 Installing Several Versions of the Library . . . 9
1.6 Thread-Safety . . . 10
1.7 Examples . . . 10

2 C Library API 11
2.1 Initialization . . . 11
2.2 Booleans . . . 11
2.3 Underlying Numerical Domain . . . 12
2.4 Octagons . . . 15
2.4.1 Octagon Management . . . 15
2.4.2 Queries . . . 15
2.4.3 Tests . . . 16
2.4.4 Operators . . . 16
2.4.5 Transfer Functions . . . 18
2.4.6 Change of Dimension . . . 19
2.4.7 Interval Manipulations . . . 21
2.4.8 Perturbation . . . 22
2.4.9 Utilities . . . 22
2.4.10 Minimized Octagons . . . 22
2.4.11 Low Level Functions . . . 23
2.4.12 Interface with the New Polka Library . . . 24
2.5 Utilities . . . 25
2.5.1 Assertions . . . 25
2.5.2 Memory Management . . . 25
2.5.3 Timing . . . 27

3 OCaml Binding API 29
3.1 Initialization . . . 29
3.2 Numerical Domain . . . 29
3.3 Boolean Lattice . . . 31
3.4 Octagons . . . 31
3.4.1 Octagon Creation . . . 31
3.4.2 Queries . . . 31
3.4.3 Tests . . . 32
3.4.4 Operators . . . 32
3.4.5 Transfer Functions . . . 33
3.4.6 Change of Dimensions . . . 34
3.4.7 Minimized Octagons . . . 35

3.4.8 Interval functions . . . 35
3.4.9 Perturbation . . . 36
3.4.10 Pretty-Printers . . . 36
3.5 Interface with the New Polka Library . . . 37
3.6 Utilities . . . 37
3.7 Low-level Binding API . . . 38

4 Analyzer Example **40**
4.1 How to Use the Analyzer . . . 40
4.2 Source Files . . . 40
4.3 Language Syntax . . . 40
4.4 Analysis Method . . . 41

5 Internal Structure **43**
5.1 Abstract Elements Representation (`oct_sem.c`) . . . 43
5.1.1 Original Representation . . . 43
5.1.2 Modified Representation (`oct_t`) . . . 43
5.2 Modified Strong Closure Algorithm (`oct_close`) . . . 44
5.2.1 Emptiness Test . . . 45
5.2.2 Incremental Strong Closure Algorithm (`oct_close_incremental`) . . . 45
5.3 Minimized Octagons (`moct_t`) . . . 45
5.3.1 Minimized Representation . . . 46
5.3.2 Minimization Algorithm . . . 46
5.3.3 Algorithms on Minimized Octagons . . . 46
5.4 Memory Management of Abstract Elements (`oct_sem.c`) . . . 47
5.5 Conversion between Octagons and Polyhedra (`oct_polka.c`) . . . 47
5.5.1 From Octagons to Polyhedra (`oct_to_poly`) . . . 47
5.5.2 From Polyhedra to Octagons (`oct_from_poly`) . . . 48
5.6 Operators and Transfer Functions (`oct_sem.c`) . . . 48

Bibliography **49**

Index **50**

License

This license applies to all files distributed in the original package, including all source code, libraries, binaries, and documentation, except the files that are part of the GNU autoconf, automake, and libtool software. A copy of this license is available in the `COPYING` file.

By using, distributing, or modifying the Octagon Abstract Domain Library, you indicate that you understand and accept all the terms of this license.

DISCLAIMER
This library is provided 'as is' without warranty of any kind, either expressed or implied, including, but not limited to, warranties of merchantability and fitness for a particular purpose, warranties of correctness and absence of error of any kind. In no event will the author be liable for any damage caused by the use, or inability to use, of the Octagon Abstract Domain Library.

USING
The Octagon Abstract Domain Library is intended for academic use only. The Octagon Abstract Domain Library may be used freely, in original or modified form, in any academic software, provided the software notice specify explicitly that it uses the Octagon Abstract Domain Library, what part of it are used, how it was modified, if it was modified, and precisely what usage of the Octagon Abstract Domain Library is done in the software. Commercial use of the library is not possible without the explicit authorization of the authors of the original and modified parts of the library.

COPYING
The Octagon Abstract Domain Library may be freely distributed in its original, unmodified package form, as long as no fee is charged for the package. Whenever the Octagon Abstract Domain Library or portions of it are distributed, this license must be included in unmodified form, and all files must contain their original copyright notices. If only a portion of a file is being distributed, the appropriate copyright notice must be copied into it from the beginning of the file, and this license must still be included in unmodified form. If the whole library or part of it is distributed in compiled or processed form, the present license must be included, as well as a a reference to the official library web site: `http://www.di.ens.fr/~mine/oct/`.

MODIFYING
The Octagon Abstract Domain Library files may be freely modified and these modified files distributed under the terms of the COPYING section, provided that these files remain under the terms of the present license, and all files carry their original copyright notice, in addition to the list of all modifications the were made and their respective authors. If a modified version of the library is distributed in source, compiled or processed form, the present license applies to all unmodified and modified parts and you must provide a

reference to the modified sources, as well as a reference to the official library web site: http://www.di.ens.fr/∼mine/oct/.

EXTERNAL TOOLS

The Octagon Abstract Domain Library is designed to operate with external programs and libraries not provided in this package, including, but not limited to, any C or C++ compiler, the OCaml programming language, the New Polka Library, the GMP library These tools must be used according to their respective license. The present license does not apply to them. The author is not responsible for any problem encountered with these tools.

Future of the library

The octagon library is now folded within the APRON project [APR], an ongoing effort to package several numerical abstract domains in a unified library using a common API. See http://www.cri.ensmp.fr/apron/.

Chapter 1

Introduction

1.1 Goal

This library implements the representation and manipulation of conjunctions of constraints of the form $(\pm x \pm y \leq c)$, so called *octagons*. Operators are intended for *Abstract Interpretation* use; so they implement *standard semantics* operators, as well *convergence acceleration* operators (see [CC77] for an Abstract Interpretation primer). This work is based on the author's Ph.D thesis [Min04] and the following publication: [Min06] (this subsumes the old, original conference paper [Min01b] and adds some more algorithms). The library is currently used in the Astrée project [BCC+03, Ast]. Also, most of its contents has been recycled into the new Apron [APR] project.

1.2 Organization of the Package

The library source is composed of the following parts:

- `clib/` C library
- `test/` C library test suite
- `ocamllib/` OCaml module (a wrapper around the C library)
- `ocamlanal/` sample OCaml static analyzer
- `doc/` this documentation

Please take a look at the following files:

- `AUTHORS` list of authors
- `COPYING` the license of the library, it is recalled in the beginning of the present documentation; *you must agree with it in order to use the library*
- `INSTALL` detailed installation informations
- `README` read this before complaining

Here is a list of what gets installed, and where:
Libraries (in `PREFIX/lib/`)

- `liboct.*` C library
- `libocamloct.*` C wrapper for the OCaml module

C headers (in `PREFIX/include/oct/`)

- `oct.h` main header
- `oct_private.h` use this to access to low-level structures (not recommended)
- `oct_num.h` underlying numerical domain

OCaml code (in `PREFIX/lib/ocaml/`)

- `oct.cma` OCaml byte-code module
- `oct.[cmx]a` OCaml native code module
- `oct*.cmi` compiled interfaces for OCaml modules
- `oct*.mli` interfaces for OCaml modules in human-readable form

Binaries (in `PREFIX/bin/`)

- `oct-config` configuration utility
- `octtest` C test-suite
- `octanal` OCaml sample analyzer

The documentation (in `PREFIX/share/oct/`)

- `doc_oct.dvi` this documentation

The library can be interfaced with the following:
- the OCaml language 3.04 [La]
- the Gnu MP and MPFR libraries 4.1 [GNU, TPpL, Mon]
- the New Polka library 2.0.0 [Jea]

In order to support several versions of the library installed with different configuration options, copies of `liboct.so` and `oct-config` that are name-mangled are installed (they are named `liboct_XXX.so` and `oct-config-XXX`, where the suffix `XXX` is derived from the choice of options).

1.3 Building and Installing the Library

There is a `configure` script that helps to compile the library easily. First of all, if the `configure` file does not exist, you can generate it using the `bootstrap` file. Then you simply have to type:

```
% ./configure --with-num=frac
% make
% make install
```

The script should discover automatically whether it can build the OCaml, GMP, MPFR, and New Polka supports. The `configure` script accepts many arguments:

- To install the C library in another directory than `/usr/local`, use `--prefix=DIR` (*e.g.*, `--prefix=$HOME`, if you are not root).
- To install the OCaml library in another directory than `/usr/local/lib/ocaml`, use the `--with-ocaml=DIR` option. OCaml version 3.04 at least must be present.
- If the New Polka Library, or the GMP / MPFR libraries are installed in non-standard directories, use the `--with-polka=DIR`, `with-gmp=DIR`, and `with-mpfr=DIR` options. New Polka version 2.0.0 at least (version 1.x will not work) and GMP version 4.1 at least must be installed with MPFR support enabled.
- By default, the library is compiled with debugging informations and no optimization. You can switch to optimized mode (much faster, but no debugging information) with the `--disable-debug` option.
- Internal memory and time profiling can be also enabled using the `--enabled-prof` option. It is disabled by default as it makes the library a little slower.
- The Octagon Library can use several different underlying numerical representations for numbers: integers, fractions, or machine floating-point numbers. If GMP and MPFR are present, you can also use GMP arbitrary precision integers, GMP arbitrary precision fractions, or MPFR floating-point numbers. To choose which one is used, use the `--with-num=NUM` option, where `NUM` is either `int`, `frac`, `float`, `longdouble`, `gmpint`, `gmpfrac`, or `mpfrfloat`.

  Note: the option `--with-num=NUM` is now **mandatory**.
- You can also choose which integer representation will be used by the New Polka Library with the `--with-polka-num=NUM` option, where `NUM` is `long`, `longlong`, or `gmp`.

  The New Polka Library name to use is derived from the `--with-polka-num` and `--enable-debug` configuration options. `long`, `longlong`, and `gmp` select respectively one of the following prefix: `libpolkai`, `libpolkal`, or `libpolkag`. If debugging is enabled, the `_debug` suffix is added to the New Polka library name used. You must make sure this library exists and can be found by the configuration program (maybe using the `--with-polka=DIR` option).
- If you want the OCaml support in the Octagon Library to work with the OCaml support for the New Polka library, you must make sure that the OCaml libraries `polka.cma` (for byte-code support) and `polka.cmxa`, `polka.a` (for native code support) are available (you may need to use the `--with-ocaml` flag). A `libpolkaX_caml` C library must also be available; as for the `libpolka` library; its exact name depends upon the configuration flags options `--with-polka-num` and `--enable-debug` chosen: one of the `libpolkai_caml`, `libpolka_caml`, or `libpolkag_caml` prefix is chosen, and the `_debug` suffix is appended if debugging is enabled.
- If you want OCaml support for GMP and MPFR numbers, you must also have the Caml-GMP library [Mon] installed.

To see all options, type `./configure --help`. Please refer to the `INSTALL` file for a detailed description of the installation process.

1.4 Using the Library

Finding the correct compile and link flags to build a program using the Octagon Library may be difficult. Thankfully, a helper shell-script, `oct-config`, is installed; it guesses automatically all the `-D`, `-I`, `-l`, and `-L` flags needed.

- In order to compile a C program `toto.c`, simply type:

  ```
  % cc `oct-config --cflags` -c toto.c
  ```

  Your C program must include the `oct/oct.h` header file in order to access to the functions in the library.

- To link C object files `toto.o` `toto2.o`, type:

  ```
  % cc `oct-config --libs` toto.o toto2.o -o totor
  ```

  If the New Polka Library support is enabled, it will also link the correct version of the New Polka Library.

- You can use both `--cflags` and `--libs` if you plan to compile and link at the same time.

- To compile or link a C++ program with the library, proceed as with a C program. Note however, that you may need to disable the New Polka library whose headers do not seem to be C++-clean.

- To build a OCaml byte-code program, type:

  ```
  % ocamlc `oct-config --mlflags` toto.ml toto2.ml
  ```

- To build a OCaml native code program, type:

  ```
  % ocamlopt `oct-config --mlflags --with-ocamlopt` toto.ml toto2.ml
  ```

- To launch a OCaml top-level with the Octagon Library linked in, type:

  ```
  % `oct-config --mltop`
  ```

  Your OCaml program should open the `Oct` module in order to access to the functions in the library.

  If the OCaml New Polka Library support is enabled, the New Polka Library will also be automatically available within the `Polka` module.

1.5 Installing Several Versions of the Library

You may want to experiment with different `configure` options at the same time; this is supported easily with by the Octagon Library. Simply run the sequence `./configure [options]; make; make install; make clean` one time for each option set you want to be installed. Do not forget to invoke `make clean` between two compilations. To choose which library is actually used, you simply recall all the desired options in `oct-config`, *e.g.*:

```
% ./configure --with-num=frac
% make
% make install
% make clean
% ./configure --disable-debug --with-num=frac
% make
% make install
% cc `oct-config --libs --cflags --enable-debug` toto.c -o toto_debug
% cc `oct-config --libs --cflags --disable-debug` toto.c -o toto
```

All options that are not recalled when calling `oct-config` are defaulted to the value they had in the last installation of the library; they can be figured out by typing `oct-config` without any argument.

If `oct-config` cannot find the library corresponding to the desired options, it will issue an error message.

This feature is supported using name-mangling. A suffix `XXX` is derived from the set of options passed to the configuration script. Then the files `liboct_XXX.so` and `oct-config-XXX` are installed. The `oct-config` scripts parses the optional arguments to retrieve the name suffix `XXX`, then calls `oct-config-XXX`

which gives the correct compilation flags and links against the correct `liboct_XXX.so`. The installation process always install a `liboct.so` file so that you can compile easily with the last library installed using the `-loct` link option, would you wish to bypass `oct-config`.

If you use the OCaml binding, name-mangled `Oct_XXX` modules are available; use `oct-config --mlmodule` to guess the right module name. One name-mangled OCaml top-level is also generated for each set of options; `oct-config --mltop` automatically choose the right one.

Note that you can use several versions of the library *a the same time* in one application. In order to support this, all non-static functions in the library have a name-mangling prefix (however, this process is hidden to the user). For example, you can do:

```
%  ./configure --with-num=frac
%  cc -c a.c ‘oct-config --cflags --with-num=float‘
%  cc -c b.c ‘oct-config --cflags --with-num=frac‘
%  cc a.o b.o -o toto ‘oct-config --libs --with-num=float‘
       ‘oct-config --libs --with-num=frac‘
```

Remark, however, that one C file can only use one library version; the multi-library feature is only supported at the linker level! Also, do not directly use an octagon created by one library with a function of another library.

1.6 Thread-Safety

When internal memory and time profiling is not enabled (default), the Octagon Library is thread-safe. Internal profiling makes use of statically allocated data, resulting in a non thread-safe implementation.

The New Polka Library is not thread-safe. You must not use the New Polka Library directly, nor the interface provided by the Octagon Library if you want to make thread-safe library calls.

1.7 Examples

The package comes with a C example and an OCaml example. The C example is a simple test-suite. The OCaml example is a basic abstract analyzer for a toy language. This analyzer is also available on-line at `http://cgi.di.ens.fr/cgi-bin/mine/octanalhtml-ndi/octanalweb`.

Chapter 2

C Library API

This chapter describes the C API used to access to the Octagon Abstract Domain Library. It is also safe to call these functions from a C++ program. The API was designed to be as close as possible to the New Polka convex polyhedra library [Jea]. If the New Polka Library is installed, some conversion functions between octagons and polyhedra are available (Section 2.4.12).

The beginning of the chapter presents the high-level functions, available when you include the header file `oct/oct.h`. The low-level functions and internal representations, available when you include `oct/oct_private.h`, are presented later in Sections 2.4.11 and 5.1

2.1 Initialization

- `int oct_init ()`

  Always call this function before using the library. If you use several versions of the library at the same time, call `oct_init` one time for each version. When using machine floating-point representation, `oct_init` tries to set the FPU rounding mode towards $+\infty$, thus ensuring the soundness of the library. Returns `1` when the library was successfully initialized, `0` elsewhere.

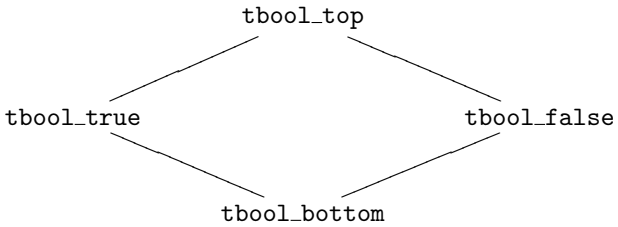
2.2 Booleans

- `bool`

  The classical boolean type `bool` and the constants `true=1`, `false=0` are defined, if not already defined by the compiler (the `configure` scripts tests this and defines the symbol `OCT_HAS_BOOL` in `oct_config.h`).

- `tbool`

  Defines the boolean lattice:



  `tbool_bottom` is returned when the answer to a question is undefined, `tbool_top` when it is not known.

2.3 Underlying Numerical Domain

The underlying numerical domain, denoted by $\mathbb{I}$, can be either $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. The implementation uses machine integers (`long`), machine floating-points numbers (`double` or `long double`) or multi-precision numbers (using GMP and MPFR) to represent $\mathbb{I}$. Then, it is lifted to $\bar{\mathbb{I}} = \mathbb{I} \cup \{+\infty\}$.

The choice of $\mathbb{I}$ and its actual implementation is done using the `--with-num=` option of the `configure` script. It can be set to `int` or `gmpint` (integers), `frac` or `gmpfrac` (rationals), `float`, `longdouble` or `mpfrfloat` (reals). Note that `float` uses, in fact, the 64−bit `double` C data-type.

Fast integers, `--with-num=int`, is unsafe in case of overflow, whether all other numerical domains either saturate to $+\infty$[1], or have arbitrary precision. `--with-num=float`, `--with-num=longdouble`, and even `--with-num=mpfrfloat`, can have some stability problems (leading to precision degradations, or difficult fixpoint convergence).

Choosing integers, `--with-num=int` or `--with-num=gmpint`, causes certain tests to be only semi-corrects (`true` means "really true" and `false` means "true or false").

name	mathematical domain	internal representation	correct when overflow	test correctness
`int`	$\mathbb{Z}$	`long`	no	semi
`gmpint`	$\mathbb{Z}$	`mpz_t` (GMP)	yes	semi
`frac`	$\mathbb{Q}$	`long`×`long`	yes	full
`gmpfrac`	$\mathbb{Q}$	`mpq_t` (GMP)	yes	full
`float`	$\mathbb{R}$	`double`	yes	full
`longdouble`	$\mathbb{R}$	`long double`	yes	full
`mpfrfloat`	$\mathbb{R}$	`mpfr_t` (MPFR)	yes	full

All numerical types and operators are defined in the file `clib/num_oct.h`. Operators are implemented as macro, when possible, or in-line functions.

- `num_t`

  Abstract type of numerical variables. This type represents values in $\bar{\mathbb{I}} = \mathbb{I} \cup \{+\infty\}$. Remark that type `num_t` can be a complex structure, involving pointers to dynamic blocks of memory (especially when using GMP and MPFR numbers); this means that variables of type `num_t` must be explicitly initialized, freed, and cannot be copied with a mere `=` (use, respectively, the `num_init`, `num_clear`, and `num_set` families of functions described bellow).

- ```
  void num_init    (num_t* a)
  void num_init_n  (num_t* a, size_t n)
  ```

  Initialize a single `num_t`, or an array of `n` `num_t`. This function must be called only once, after the variable is created and before it is used with any operator.

- ```
  void num_clear    (num_t* a)
  void num_clear_n  (num_t* a, size_t n)
  ```

  Uninitialize a single `num_t`, or an array of `n` `num_t`. This function must be called only once, before the variable is deleted, if it has been initialized.

- ```
  void num_set (num_t* a, const num_t* b)
  ```

  Copy the value of the number `b` into `a`.

- ```
  void num_set_n (num_t* a, const num_t* b, size_t n)
  ```

  Copy the `n` numbers from the array `b` into the array `a`.

[1] when `--with-num=float` or `--with-num=longdouble` is used, `oct_init` automatically sets your FPU in "round to $+\infty$" mode.

- ```
  void num_set_int    (num_t* a, long i)
  void num_set_float  (num_t* a, double d)
  void num_set_frac   (num_t* a, long i, long j)
  void num_set_infty  (num_t* a)
  ```

  Set `a` to the integer `i`, the double `d`, the fraction `i/j`, or the $+\infty$ number. If the number cannot be exactly represented in the current `num_t` type, an over-approximation is put instead (possibly $+\infty$).

- ```
  void num_init_set        (num_t* a, const num_t* b)
  void num_init_set_n      (num_t* a, const num_t* b, size_t n)
  void num_init_set_int    (num_t* a, long i)
  void num_init_set_float  (num_t* a, double d)
  void num_init_set_frac   (num_t* a, long i, long j)
  void num_init_set_infty  (num_t* a)
  ```

  These operators are combinations of an initialization operator (`num_init` or `num_init_n`) and a copy operator. You must not initialize `a` before using these functions, and they can be used only once on an non initialized number (subsequent copies are done using the corresponding `num_set_` function).

- ```
  bool    num_fits_int    (const num_t* a)
  bool    num_fits_float  (const num_t* a)
  bool    num_fits_frac   (const num_t* a)

  long    num_get_int     (const num_t* a)
  double  num_get_float   (const num_t* a)
  long    num_get_num     (const num_t* a)
  long    num_get_den     (const num_t* a)
  ```

  Converts the number `a` into an integer (`num_get_int`), a float (`num_get_float`), or a fraction (`num_get_num` for the numerator, and `num_get_den` for the denominator). If the number cannot be represented in the desired format, an over-approximation is returned. This only works when the corresponding `num_fits_` function returns `true`. Elsewhere, the number should be considered as infinite (either truly infinite, or too big to be over-approximated in the desired type).

- ```
  bool num_infty(const num_t* a)
  ```

  Returns `true` if and only if `a` represents $+\infty$.

- ```
  void num_set_mpz  (num_t* a, const mpz_t b)
  void num_set_mpq  (num_t* a, const mpq_t b)
  void num_get_mpz  (mpz_t a, const num_t* b)
  void num_get_mpq  (mpq_t q, const num_t* b)
  ```

  These functions are defined only if the GMP library is available. They can be used to convert between multi-precision integers or fractions, and numbers. There is no `num_fits_` associated functions as an over-approximation can always be put in a multi-precision number, as long as it is not $+\infty$ (this can be tested with `num_infty`).

- ```
  void num_set_mpfr  (num_t* a, const mpfr_t b)
  void num_get_mpfr  (mpfr_t a, const num_t* b)
  ```

  These functions are defined only if the MPFR library is available. They can be used to convert between multi-precision floating-point values and numbers. There is no `num_fits_` associated function, you should use `num_infty` instead.

- ```
  int num_cmp       (const num_t* a, const num_t* b)
  int num_cmp_int   (const num_t* a, long b)
  int num_cmp_zero  (const num_t* a)
  ```

  Returns a strictly positive integer if `a` is strictly greater than `b`, a strictly negative integer if `a` is strictly lesser than `b`, and zero if `a` and `b` are equal. One can compare two `num_t` numbers, or a `num_t` number with an integer. The last form is the special case `b=0`: it returns the sign of `a`.

- ```
  void num_max       (num_t* r, const num_t* a, const num_t* b)
  void num_min       (num_t* r, const num_t* a, const num_t* b)
  void num_add       (num_t* r, const num_t* a, const num_t* b)
  void num_sub       (num_t* r, const num_t* a, const num_t* b)
  void num_mul       (num_t* r, const num_t* a, const num_t* b)
  void num_mul_by_2  (num_t* r, const num_t* a)
  void num_div_by_2  (num_t* r, const num_t* a)
  void num_neg       (num_t* r, const num_t* a)
  ```

  Put in `r` the result of the operation (respectively $\max(\mathtt{a},\mathtt{b})$, $\min(\mathtt{a},\mathtt{b})$, $\mathtt{a}+\mathtt{b}$, $\mathtt{a}-\mathtt{b}$, $\mathtt{a}*\mathtt{b}$, $\mathtt{a}*2$, $\mathtt{a}/2$, and $-\mathtt{a}$). The result is always either exact, or an over-approximation (possibly $+\infty$). $+\infty$ arguments are correctly handled, except for $-(+\infty)$, $x-(+\infty)$, and $x*(+\infty)$ which are undefined. One can safely have `r=a=b`.

- ```
  void num_print     (const num_t* a)
  void num_snprintf  (char* s, size_t n, const num_t* a)
  ```

  Prints the number `a`, either directly to `stdout`, or to a string (up to `n` characters).

- `OCT_NUM_EXACT`

  This symbol is only defined when `num_t` always allows the exact representation of all numerical operators (up to overflowing to $+\infty$). Only fractional types are exact: integers are not closed under division by 2, and floating-point numbers are subject to rounding errors.

- `OCT_NUM_CLOSED`

  This symbol is only defined when `num_t` is closed under division by 2. That is to say, when $\mathbb{I} \neq \mathbb{Z}$. When the underlying numerical domain is closed, the strong closure algorithm is a normal form, all tests are complete and we have a best approximation for union (not taking into account rounding errors that can occur).

- `OCT_DOMAIN`

  This symbol recalls the current underlying numerical domain $\mathbb{I}$. It is defined to be one of `OCT_DOMAIN_INT`, `OCT_DOMAIN_FRAC`, or `OCT_DOMAIN_REAL`. `oct_domain_string[OCT_DOMAIN]` is a human-readable string representation of $\mathbb{I}$ (`"integers"`, `"rationals"`, or `"reals"`).

- `OCT_IMPLEMENTATION_STRING`

  This string recall which internal data type is used to represent the current underlying numerical domain $\mathbb{I}$ (one of `"long"`, `"double"`, `"mpz"`, `"mpq"`, or `"mpfr"`).

When `OCT_NUM_CLOSED` is not defined ($\mathbb{I} = \mathbb{Z}$), the strong closure is not a full normal form; thus emptiness, inclusion, and equality tests are only semi-decidable (`true` means really true, whereas `false` means *I do not know*), and some operators and transfer functions are not the most accurate possible. However the octagon domain *remains sound*.
  ```

2.4 Octagons

A set of constraints of the form $(\pm x_i \pm x_j \leq c)$, with $i, j \in \{0, \ldots, N-1\}$ and $c \in \overline{\mathbb{I}}$ is represented internally by a *matrix of dimension N*.

- `oct_t*`

  Abstract type of octagons. The real definition lies in `oct_private.h` but is somewhat complex (see Section 5.1). You only manipulate *pointers* to oct_t structures. Use the library (see Section 2.4.1) to create, copy and destroy octagons: do not call `malloc`, `memcpy` or `free` on the pointers.

- `var_t`

  This is the integer type representing a domain dimension, or a variable index. Presently, it is `unsigned int`.

2.4.1 Octagon Management

- `oct_t* oct_empty (var_t n)`

  Creates a new octagon representing the empty domain $\emptyset$ in `n` dimensions. No constraint is stored in an empty octagon, so the representation has a $\mathcal{O}(1)$ memory cost. The returned octagon must be freed by `oct_free`, or used as a destructive argument when it is no longer used.

- `oct_t* oct_universe (var_t n)`

  Creates a new octagon representing the full space $\mathbb{I}^{\mathtt{n}}$ in `n` dimensions. The representation has a $\mathcal{O}(\mathtt{n}^2)$ memory cost. The returned octagon must be freed by `oct_free`, or used as a destructive argument when it is no longer used.

- `oct_t* oct_copy (oct_t* m)`

  Returns a copy of the octagon `m`. This is a *lazy copy*, as octagons are reference counted; so it is very fast ($\mathcal{O}(1)$ memory and time cost): the library takes care of making a full copy when needed (see Section 5.4). The returned octagon must be freed by `oct_free`, or used as a destructive argument when it is no longer used.

- `void oct_free (oct_t* m)`

  Decrease reference count of the octagon `m`, and actually discards it if it is no longer needed. Every octagon returned by a function must be freed, except if the octagon is used as a destructive argument in a function call.

2.4.2 Queries

- `var_t oct_dimension (oct_t* m)`

  Returns the dimension N of the octagon.

- `size_t oct_nbconstraints (oct_t* m)`

  Returns the number of constraints used to represent internally `m`. Does not take into account constraints of the form $(\pm x_i \pm x_j \leq +\infty)$, nor $(x_i - x_i \leq 0)$; however there may be redundancy (one constraint can be the sum of two other constraints, for example). If the octagon is empty, it is not represented as a set of constraints, so the function returns 0. $\mathcal{O}(N^2)$ time cost.

2.4.3 Tests

Different matrices can represent the same octagon. Thus, there exists a *normal form* called the *closed form*. Some test and manipulation algorithms require to compute the closed form of its argument; this process has a $\mathcal{O}(N^3)$ time cost, and a $\mathcal{O}(N)$ memory cost. Closure computation is done automatically when needed, you do not need to bother about it.

If you choose $\mathbb{I} = \mathbb{Z}$ (`OCT_NUM_CLOSED` not defined), these tests are only semi-decidable: `true` means "really true", whereas `false` means *"I do not know"*.

- `bool oct_is_empty (oct_t* m)`

  Returns `true` if the octagon `m` has an empty domain, `false` elsewhere. This call computes the closure of the matrix; the octagon `m` is not modified, but its closure is cached for future use. $\mathcal{O}(1)$ time cost, on behalf of the cost of the closure.

- `tbool oct_is_empty_lazy (oct_t* m)`

  This test does not perform the closure algorithm. So, if the closed form is not already available, it returns `tbool_top`. $\mathcal{O}(1)$ time cost in all cases.

- `bool oct_is_universe (oct_t* m)`

  Returns `true` if the domain of `m` is $\mathbb{I}^N$; `false` elsewhere. $\mathcal{O}(N^2)$ time cost.

- `bool oct_is_included_in (oct_t* ma, oct_t* mb)`

  Returns `true` if the domain of `ma` is included in or equal to the domain of `mb`, and `false` elsewhere. This call computes the closure of `ma` (but not the closure of `mb`) and caches. $\mathcal{O}(N^2)$ time cost, on behalf of the cost of the closure.

- `tbool oct_is_included_in_lazy (oct_t* ma, oct_t* mb)`

  This test does not perform the closure algorithm. So, if the closed form of `ma` is not already available, it may return `tbool_top`. $\mathcal{O}(N^2)$ time cost in all cases.

- `bool oct_is_equal (oct_t* ma, oct_t* mb)`

  Returns `true` if `ma` and `mb` have the same domain, and `false` elsewhere. This call computes the closure of `ma` and `mb` and caches them. $\mathcal{O}(N^2)$ time cost, on behalf of the cost of the closures.

- `tbool oct_is_equal_lazy (oct_t* ma, oct_t* mb)`

  This test does not perform the closure algorithm. So, if the closed form of `ma` and `mb` are not already available, it may return `tbool_top`. $\mathcal{O}(N^2)$ time cost in all cases.

- `bool oct_is_in (oct_t* m, const num_t* v)`

  Returns `true` if the point represented by `v` is in the domain of `m`. `v` is an array containing N elements of type `num_t`. $\mathcal{O}(N^2)$ time cost.

2.4.4 Operators

It is implied that the octagon arguments of a binary operator have the same dimension N.

When `destructive` is `true`, the arguments are destroyed and cannot be reused after the call (you must not call `oct_free` on them); if it is set to `false`, the arguments are left untouched. The octagons returned by the functions are always new octagons that must eventually be either freed with `oct_free`, or used as destructive arguments. Destructive operators are more efficient: they save some internal allocations and copies.

■ `oct_t* oct_intersection (oct_t* ma, oct_t* mb, bool destructive)`

Returns the exact intersection of two octagons. $\mathcal{O}(N^2)$ time cost.

■ `oct_t* oct_convex_hull (oct_t* ma, oct_t* mb, bool destructive)`

Returns the smallest octagon that contains the union of two octagons. It must perform closure on the two octagons, but it does not cache them for future use as it is seldom useful. $\mathcal{O}(N^2)$ time cost, on behalf of the cost of the closure.

If you choose $\mathbb{I} = \mathbb{Z}$ (`--with-num=int` or `--with-num=gmpint`), the returned octagon is not the *best* approximation of the union; however it still contains both octagons.

■ `oct_t* oct_widening( oct_t* ma, oct_t* mb, bool destructive, oct_widening_type type )`

Returns the widening (`ma`) $\triangledown$ (`mb`$^\bullet$). One of the following three widenings is chosen according to the value of `type`:

- `OCT_WIDENING_FAST`, this is the original widening presented in [Min01b]
$$[m \triangledown n]_{ij} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij}, \\ +\infty & \text{elsewhere;} \end{cases}$$
- `OCT_WIDENING_ZERO`, this widening is a little more precise but has a slower convergence
$$[m \triangledown n]_{ij} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij}, \\ 0 & \text{if } m_{ij} < n_{ij} \leq 0, \\ +\infty & \text{elsewhere;} \end{cases}$$
- `OCT_WIDENING_UNIT`, this widening is more precise and converges more slowly
$$[m \triangledown n]_{ij} \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij}, \\ -1 & \text{if } m_{ij} < n_{ij} \leq -1, \\ 0 & \text{if } m_{ij} < n_{ij} \leq 0, \\ 1 & \text{if } m_{ij} < n_{ij} \leq 1, \\ +\infty & \text{elsewhere.} \end{cases}$$
- `OCT_PRE_WIDENING`, this is not an actual widening, but a degenerate hull (precisely, a hull without closure of the left argument). It is tantalizing to interleave widenings and hulls to improve the precision of fix-point computations but, unfortunately, this destroys the converge property and makes analyzes loop forever. `OCT_PRE_WIDENING` is a middle-ground. It does not ensure convergence by itself, but can be safely interleaved with widenings. As long as proper widenings occur infinitely often, the interleaved sequence will converge. Also, it converges more slowly, and so, gives a better precision.

For all `type`, `mb` is closed for better accuracy, but `ma` is not as it would prevent convergence (see [Min01a]). $\mathcal{O}(N^2)$ time cost, on behalf of the cost of the closure.

■ `oct_t* oct_widening_steps( oct_t* ma, oct_t* mb, bool destructive, int nb_steps, num_t* steps )`

This widening uses a set of threshold values to gradually loosen unstable constraints. `steps` is an array of `nb_steps` numbers in increasing order. If the constraint $\pm x \pm y \leq c$ in `ma` is not stable in `mb`, it will replace it by $\pm x \pm y \leq$ `steps`[`i`], such that `steps`[`i`] $\geq c$ and `i` is minimal, or remove the constraint if there is no such `i`.

■ `oct_t* oct_narrowing (oct_t* ma, oct_t* mb, bool destructive)`

Returns the narrowing (`ma`$^\bullet$) $\triangle$ (`mb`$^\bullet$).

$$[m \triangle n]_{ij} \begin{cases} n_{ij} & \text{if } m_{ij} = +\infty, \\ m_{ij} & \text{elsewhere.} \end{cases}$$

Both octagons are closed. $\mathcal{O}(N^2)$ time cost, on behalf of the cost of the closure.

2.4.5 Transfer Functions

If you choose $\mathbb{I} = \mathbb{Z}$ (`--with-num=int` or `--with-num=gmpint`), some transfer functions may not be as accurate as possible.

- `oct_t* oct_forget (oct_t* m, var_t k, bool destructive)`

  Returns an octagon where all informations about the variable $v_{\mathtt{k}}$ has been forgotten. This transfer function corresponds to both instructions $v_{\mathtt{k}} \leftarrow ?$ and $v_{\mathtt{k}} \rightarrow ?$ (*i.e.*, forward and backward assignment). It computes the closure of its argument. $\mathcal{O}(N)$ time cost, on behalf of the cost of the closure.

- `oct_t* oct_add_bin_constraints ( oct_t* m, unsigned int nb, const oct_cons* cons, bool destructive )`

  Adds some constraints of the form $(\pm v_{\mathtt{x}} \pm v_{\mathtt{y}} \leq c)$ or $(\pm v_{\mathtt{x}} \leq c)$ to the octagon `m`. It corresponds to the semantics of the several instructions of the form `assert(`$\pm v_{\mathtt{x}} \pm v_{\mathtt{y}} \leq c$`)` and `assert(`$\pm v_{\mathtt{x}} \leq c$`)`.

  The array `cons` contains the `nb` constraints to add. The structure `oct_cons` has the following fields:

  - `enum type` is the type of the constraint, it can be
    - `px`, constraint of the form $(v_{\mathtt{x}} \leq c)$,
    - `mx`, constraint of the form $(-v_{\mathtt{x}} \leq c)$,
    - `pxpy`, constraint of the form $(v_{\mathtt{x}} + v_{\mathtt{y}} \leq c)$,
    - `pxmy`, constraint of the form $(v_{\mathtt{x}} - v_{\mathtt{y}} \leq c)$,
    - `mxpy`, constraint of the form $(-v_{\mathtt{x}} + v_{\mathtt{y}} \leq c)$,
    - `mxmy`, constraint of the form $(-v_{\mathtt{x}} - v_{\mathtt{y}} \leq c)$;
  - `var_t x` is the index of the first variable;
  - `var_t y` is the index of the second variable; it is not used if the constraint is a unary constraint (`type` is `px` or `mx`);
  - `num_t c` is the numerical constant appearing in the right member of the constraint. Do not forget to initialize `c` with a function from the `num_init` family!

  $\mathcal{O}($`nb`$)$ time cost.

- `oct_t* oct_assign_variable ( oct_t* m, var_t x, const num_t* tab, bool destructive )`

  Returns a new octagon representing the domain of `m` after the linear variable assignment $v_{\mathtt{x}} \leftarrow \left(\sum_{i=0}^{N-1} v_i \cdot \mathtt{tab}[i]\right) + \mathtt{tab}[N]$.

  When the the result of the assignment is not representable by an octagon, or would be too costly to compute, a sound over-approximation is returned. The transfer function is exact for assignments of the form $x \leftarrow c$, $x \leftarrow \pm x + c$, and $x \leftarrow \pm y + c$. Sound approximations include the use of interval arithmetic.

- `oct_t* oct_interv_assign_variable ( oct_t* m, var_t x, const num_t* tab, bool destructive )`

  Same as `oct_assign_variable`, but allowing *interval* coefficients instead of constant ones: $v_{\mathtt{x}} \leftarrow \left(\sum_{i=0}^{N-1} v_i \cdot [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]\right) + [-\mathtt{tab}[2N+1], \mathtt{tab}[2N]]$. For each variable $v_{\mathtt{y}}$, upper bounds for $\pm v_{\mathtt{x}} \pm v_{\mathtt{x}}$ are derived by evaluating the right member using interval arithmetics. Thus, it can infer new relational constraints, but does not use the relational information within the argument octagon. It is reasonably precise yet quite fast.

■ `oct_t* oct_substitute_variable ( oct_t* m, var_t x, const num_t* tab, bool destructive )`

Returns a new octagon representing the domain of `m` after the linear variable substitution $v_{\mathtt{x}} \to \left(\sum_{i=0}^{N-1} v_i \cdot \mathtt{tab}[i]\right) + \mathtt{tab}[N]$. It is the *backward* semantics of the corresponding assignment.

When the the result of the substitution is not representable by an octagon, or would be too costly to compute, a sound over-approximation is returned. The transfer function is exact for substitution of the form $x \to c$, $x \to \pm x + c$, and $x \to \pm y + c$. For the moment, the general case is not very precise and could be improved by using interval arithmetic, as it is the case for `oct_assign_variable`.

■ `oct_t* oct_interv_substitute_variable ( oct_t* m, var_t x, const num_t* tab, bool destructive )`

Same as `oct_substitute_variable`, but allowing *interval* coefficients instead of constant ones: $v_{\mathtt{x}} \to \left(\sum_{i=0}^{N-1} v_i \cdot [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]\right) + [-\mathtt{tab}[2N+1], \mathtt{tab}[2N]]$. For the moment, only the cases $v_{\mathtt{x}} \to \pm v_{\mathtt{y}} + [a, b]$ are treated precisely, while the general case falls back to forgetting the value of $v_{\mathtt{x}}$. It could be improved using techniques similar to `oct_interv_assign_variable`.

■ `oct_t* oct_add_constraint ( oct_t* m, const num_t* tab, bool destructive )`

Returns a new octagon representing the domain of `m` with the linear constraint added $\left(\sum_{i=0}^{N-1} v_i \cdot \mathtt{tab}[i]\right) + \mathtt{tab}[N] \geq 0$.

When the the result of the constraint is not representable by an octagon, or would be too costly to compute, a sound over-approximation is returned. The transfer function is exact for constraints of the form $c \geq 0$, $\pm x + c \geq 0$, and $\pm x \pm y + c \geq 0$. The general case simply returns its argument unchanged.

■ `oct_t* oct_interv_add_constraint ( oct_t* m, const num_t* tab, bool destructive )`

Same as `oct_add_constraint`, but allowing *interval* coefficients instead of constant ones: $\left(\sum_{i=0}^{N-1} v_i \cdot [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]\right) + [-\mathtt{tab}[2N+1], \mathtt{tab}[2N]] \geq 0$.

As `oct_interv_assign_variable`, it is able to infer new relational constraints but does not take into account relational information in the octagon argument. It is reasonably precise and fast.

■ `oct_t* oct_time_flow ( oct_t* m, const num_t* nmin, const num_t* nmax, const num_t *tab, bool destructive )`

Let time flow for an amount $t \in [\mathtt{nmin}, \mathtt{nmax}]$, where $0 \leq \mathtt{nmin} \leq \mathtt{nmax}$: $\forall i$, $v_i \leftarrow v_i + [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]] \times t$. For each variable v_i, the range of evolution rates is given by the non-empty interval $[-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]$.

2.4.6 Change of Dimension

The following functions change the dimension N of an octagon by adding or subtracting variables *at the end* of $\mathcal{V}$. It is most useful to model creation and destruction of local variables when entering and exiting from nested instruction blocks.

■ `oct_t* oct_add_dimensions_and_embed( oct_t* m, var_t dimsup, bool destructive )`

Adds `dimsup` variables at the end of $\mathcal{V}$ without adding any constraint. If the original domain is denoted by $\mathcal{D}^+(\mathtt{m})$, the new domain is $\mathcal{D}^+(\mathtt{m}) \times \mathbb{I}^{\mathtt{dimsup}}$. $\mathcal{O}(\mathtt{dimsup}^2)$ time cost; however the call to `realloc` may induce an extra $\mathcal{O}(N^2)$ time for copy.

■ `oct_t* oct_add_dimensions_and_project( oct_t* m, var_t dimsup, bool destructive )`

Adds `dimsup` variables at the end of $\mathcal{V}$. The new variables are initialized to 0. If the original domain is denoted by $\mathcal{D}^+(\mathtt{m})$, the new domain is $\mathcal{D}^+(\mathtt{m}) \times \{\ 0\ \}^{\mathtt{dimsup}}$. $\mathcal{O}(\mathtt{dimsup}^2)$ time cost; however the call to `realloc` may induce an extra $\mathcal{O}(N^2)$ time for copy.

■ `oct_t* oct_remove_dimensions( oct_t* m, var_t dimmin, bool destructive )`

Removes the `dimmin` last variables from $\mathcal{V}$. It computes the closure of its argument. No time cost, on behalf of the cost of the closure; however the call to `realloc` may induce an extra $\mathcal{O}((N - \mathtt{dimmin})^2)$ time for copy.

The following functions are used to add and remove dimensions not necessarily at the end of $\mathcal{V}$. They all use arrays of `dimsup_t` objects to define where dimensions are added or removed. Note that these functions are a little less efficient (by a constant factor) than the functions that add and remove dimensions at the end of $\mathcal{V}$.

■ `struct dimsup_t { var_t pos; var_t nbdims; }`

Specifies that `nbdims` variables must be added or removed just before the variable at position `pos`. All positions are counted from 0, and always with respect to the variable set of the *argument* octagon.

■ `oct_t* oct_add_dimensions_and_embed_multi( oct_t* m, const dimsup_t* t, size_t size_t, bool destructive )`

Adds variables in $\mathcal{V}$ without adding any constraint. The positions and numbers of variables to add are specified in the array `t` containing `size_t` structure of type `dimsup_t`. Note that the `pos` fields of `t` elements must be sorted in *strictly increasing order*.

$\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *after* insertion.

■ `oct_t* oct_add_dimensions_and_project_multi( oct_t* m, const dimsup_t* t, size_t size_t, bool destructive )`

Adds zero-initialized variables in $\mathcal{V}$.

$\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *after* insertion.

■ `oct_t* oct_remove_dimensions_multi( oct_t* m, const dimsup_t* t, size_t size_t, bool destructive )`

Removes variables from $\mathcal{V}$.

$\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *before* deletion, plus an optional extra cubic time to compute the closure.

The following functions add and remove dimensions but also apply a permutation of the variables. Each permutation is given by an array `t` of `var_t` values. It has the following semantics: in each constraint, each variable $V_{\mathtt{i}}$ is replaced with the variable $V_{\mathtt{t[i]}}$. These functions are less efficient than the functions that add and remove dimensions at arbitrary positions, and much less efficient (although by a constant factor) than the functions that add and remove dimensions at the end of $\mathcal{V}$.

- ```
  oct_t* oct_add_permute_dimensions_and_embed(  oct_t* m,
                                                const var_t d,
                                                const var_t* t,
                                                bool destructive  )
  ```

  Adds `d` uninitialized variables at the end of $\mathcal{V}$. Then, applies a permutation. The array `t` must thus contain a permutation of integers from 0 to `d` + `oct_dimension(m)` − 1.

  $\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *after* insertion.

- ```
  oct_t* oct_add_permute_dimensions_and_project(  oct_t* m,
                                                  const var_t d,
                                                  const var_t* t,
                                                  bool destructive  )
  ```

  Adds zero-initialized variables in $\mathcal{V}$.

  $\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *after* insertion.

- ```
  oct_t* oct_permute_remove_dimensions(  oct_t* m,
                                         const var_t d,
                                         const var_t* t,
                                         bool destructive
  ```

  First applies the permutation, then removes some variables from $\mathcal{V}$. Thus, the array `t` is a permutation of integers from 0 to `oct_dimension(m)` − 1. When `t[i]` ≥ `oct_dimension(m)` − `d`, all information on the original variable $V_{\mathtt{i}}$ is lost.

  $\mathcal{O}(\mathtt{n}^2)$ time cost, in the size of the variable set $\mathcal{V}$ *before* deletion, plus an optional extra cubic time to compute the closure.

2.4.7 Interval Manipulations

- ```
  void oct_get_bounds (  oct_t* m, var_t k,
                         num_t* up, num_t* down  )
  ```

  Returns the interval where the variable $v_{\mathtt{k}}$ lies: $v_{\mathtt{k}} \in [-*\mathtt{down}, *\mathtt{up}]$. If the octagon is empty, `*up` and `*down` are not updated and should be ignored. This call computes the closed form of its argument, and caches it for future use.

- ```
  oct_t* oct_set_bounds (  oct_t* m, var_t k,
                           const num_t* up,const num_t* down,
                           bool destructive                    )
  ```

  Returns a new octagon with the same domain as `m` except for $v_{\mathtt{k}}$ which is now in the interval $[-*\mathtt{down}, *\mathtt{up}]$. All prior informations about $v_{\mathtt{k}}$ are lost. This is strictly equivalent to a call to `oct_forget` followed by two calls to `oct_add_bin_constraints`. Used in combination with `oct_get_box` or `oct_get_bounds`, it provides a way to the user to model complex arithmetic using its own interval arithmetic.
  ```

- `num_t* oct_get_box (oct_t* m)`

  Returns the box enclosing the octagon as an array `r` of `2n` numbers: $v_{\mathtt{i}} \in [-\mathtt{r}[\mathtt{2i}+\mathtt{1}], \mathtt{r}[\mathtt{2i}]]$. If the octagon is empty, `r` is `NULL`. This call computes the closed form of its argument, and caches it for future use. `r` must be freed with `oct_mm_free`.

- `oct_t* oct_get_box (var_t n, const num_t* b)`

  Creates a new octagon of dimension `n` the domain of which is defined by the box `b` containing `2n` numbers: $v_{\mathtt{i}} \in [-\mathtt{r}[\mathtt{2i}+\mathtt{1}], \mathtt{r}[\mathtt{2i}]]$.

2.4.8 Perturbation

- `oct_t* oct_add_epsilon ( oct_t* m, const num_t* epsilon, bool destructive)`

  Returns an enlarged octagon, where all constraints $\pm x \pm y \leq a$ are replaced by $\pm x \pm y \leq a + \mathtt{epsilon}|a|$. `epsilon` must be positive.

  $\mathcal{O}(N^2)$ time cost. The normal form is lost.

- `oct_t* oct_add_epsilon_max ( oct_t* m, const num_t* epsilon, bool destructive)`

  As `oct_add_epsilon`, but all constraints are replaced by $\pm x \pm y \leq a + \mathtt{epsilon} \times M$. where $M = \max\{\ |m| \mid \pm x \pm y \leq m,\ m \neq +\infty\ \}$. `epsilon` must be positive.

  $\mathcal{O}(N^2)$ time cost. The normal form is lost.

- `oct_t* oct_add_epsilon_bin ( oct_t* ma, oct_t* mb, const num_t* epsilon, bool destructive)`

  Binary version of `oct_add_epsilon_max`, where only the non-stable constraints in `mb` are enlarged by $\mathtt{epsilon} \times M$. This is a form of widening. `epsilon` must be positive.

  $\mathcal{O}(N^2)$ time cost. The normal form is lost.

2.4.9 Utilities

- `void oct_print (const oct_t* m)`

  Prints on the standard output the domain of the octagon `m` as a constraint set, or `empty` if the octagon is empty. Also prints `(closed)` if `m` is in closed form. Does not print constraints of the form $(\pm x_i \pm x_j \leq +\infty)$, nor $(x_i - x_i \leq 0)$; however there may be redundancy (one constraint can be the sum of two other constraints, for example). See `oct_m_print` to print the domain of an octagon without redundancy.

2.4.10 Minimized Octagons

If the $\mathcal{O}(N^2)$ memory cost of octagons become prohibitive, you can use the `moct_t` minimized representation. In the minimized form, only a minimal set of non-redundant constraints is kept and stored in a hollow matrix. This is space-efficient, but not suitable for manipulation: only few operators are available for minimized octagons. Minimized octagons are intended mostly for storage. A notable exception is the equality test `oct_m_is_equal`, which has a $\mathcal{O}(N^2)$ time cost in the worst case, as minimization is also a normal form.

- `moct_t*`

  Abstract type of minimized octagons. Use `oct_m_from_oct` to construct one, `oct_m_free` to free one. Unlike regular octagons, minimized octagons are not in-place modifiable; so there is no memory-friendly destructive operations on them.

- `moct_t* oct_m_from_oct (oct_t* m)`

  The only way of constructing a minimized octagon is conversion from a regular octagon. The closure algorithm is first applied, then a $\mathcal{O}(N^3)$ time cost minimization algorithm that removes redundant constraints (see Section 5.3.2). `m` is not modified. $\mathcal{O}(N^3)$ time and $\mathcal{O}(N)$ memory cost.

- `void oct_m_free (moct_t* m)`

  Call this to discard a minimized octagon that is no longer used.

- `oct_t* oct_m_to_oct (moct_t* m)`

  Converts the minimized octagon into a regular octagon. Does not modify `m`.

- `bool oct_m_is_equal(moct_t* ma, moct_t* mb)`

  Return `true` is the two minimized octagons represent the same domain, and `false` elsewhere. $\mathcal{O}(N^2)$ worst-case time cost, and no memory cost.

- `bool oct_m_is_empty (moct_t* m)`

  Returns `true` if `m` represents an empty octagon, `false` elsewhere. Emptiness of a minimized octagon is determined at creation-time, so this test is constant-time.

- `int oct_m_dimension (moct_t* a)`

  Returns the dimension N of the minimized octagon.

- `void oct_m_print (moct_t* a)`

  Prints on the standard output the domain of the minimized octagon as a constraint list. Because the octagon is in minimal form, printed constraints are not redundant.

2.4.11 Low Level Functions

Because they are not of common use, these low-level functions are only available if you include the file `oct/oct_private.h`. You need to understand the internal representation of octagons before using these functions (Section 5.1).

Low-Level Access

- `num_t* oct_elem (oct_t* m, var_t i, var_t j)`

  Returns a pointer to the element at line `i`, column `j` in the internal matrix representation of `m`: $0 \leq \texttt{i} < 2N$ and $0 \leq \texttt{j} < 2N$. This element is denoted by $\mathbf{m}_{ij}^{+}$, and corresponds to the constraint $(\alpha_j v_{j/2} - \alpha_i v_{i/2} \leq \mathbf{m}_{ij}^{+})$, where $\alpha_i = +1$ if i is odd, and $\alpha_i = -1$ if i is even.

  If you modify the content of the pointer, you are responsible for updating `m->close` and `m->state`.

  $\mathcal{O}(1)$ time cost.

Octagon Management

- `oct_t* oct_alloc (var_t n)`

  Allocates space for a non-empty octagon. The internal matrix is allocated (unlike a call to `oct_empty`) but its content is not initialized (unlike `oct_universe`); thus $\mathbf{m}^+_{ii} \neq 0$ and the representation is initially *invalid* (making this call a bit faster than `oct_universe`). You are responsible for filling the internal matrix representation and setting `m->state` accordingly.

- `oct_t* oct_full_copy (oct_t* m)`

  Allocates a new octagon which is an exact copy of `m`, except its reference count is 1. The internal matrix representation is also copied. This procedure is used internally whenever the library needs to modify an octagon in-place but the original octagon must be kept. $\mathcal{O}(N^2)$ time cost.

Closure

- `oct_t* oct_close (oct_t* m, bool destructive, bool cache)`

  Returns a new octagon which has the same domain as `m` but is in closed form. Also checks for emptiness and, if `m` has an empty domain, returns an empty octagon. If `cache` is `true` and the argument `m` is not destroyed by the call, makes `m` remember that the closed form is available; this prevents re-computation of `m`'s closure, but also prevents the returned octagon to be actually freed before `m` is. $\mathcal{O}(N^3)$ time cost, $\mathcal{O}(N)$ memory cost (no temporary matrix is created during this call). See Section 5.2.

- `void oct_close_incremental (oct_t* m, var_t v)`

  Performs one step of incremental closure on the octagon `m`. The argument `m` is supposed to be equal to a closed octagon, except for the informations regarding the variable `v`. It is modified in-place to recover the closure. $\mathcal{O}(N^2)$ time cost and $\mathcal{O}(N)$ memory cost. See Section 5.2.

- `bool oct_is_closed (oct_t* m)`

  Returns `true` if `m` is in closed form, or is an empty octagon. Returns `false` if a call to `oct_close` is necessary to discover whether `m` has an empty domain, and to get its closed form. Constant cost.

- `oct_t* oct_close_lazy (oct_t* m, bool destructive)`

  As `oct_close`, but returns an octagon in closed form only if it is available without computation (either `m` is already in closed form, or empty, or the closed form of `a` has already been computed and cached). Elsewhere, simply returns `m`. Constant cost.

- `bool oct_check_closed (oct_t* m, bool quiet)`

  This procedure checks that `m` is in a valid closed form (see Section 5.2). It should always return `true` for octagons created by `oct_close` and is intended only for debugging purpose. $\mathcal{O}(N^3)$ time cost.

2.4.12 Interface with the New Polka Library

When the `configure` script discovers the New Polka Library, it defines the symbol `OCT_HAS_NEW_POLKA`, and the following two functions are available to convert between octagons and polyhedra. By setting the `--with-polka-num=` option to `long`, `longlong`, or `gmp` when calling `configure`, you can change the integer representation used in the New Polk Library. This will set the symbol `POLKA_NUM` to the appropriate value (`1`, `2`, or `3`), and choose the correct New Polka Library to link with (see the New Polka Library Manual [Jea] for more informations). Remark that if the Octagon Library is compiled with debugging enabled

(`--enable-debug` configuration option), then the New Polka Library with debugging informations will be chosen.

You need to call `polka_initialize` before calling the following functions, as well as functions in the New Polka Library.

Beware the New Polka Library and the Octagon Abstract Domain Library may not use the same underlying numerical domain; conversion may result in over-approximations, but are *always sound*. To have the best precision, you should use `--with-polka-num=gmp` and `--with-num=frac` (or `--with-num=gmpfrac`).

- `poly_t* oct_to_poly (oct_t* m)`

  Creates a polyhedron with the same domain as the octagon `m`. The polyhedron can then be manipulated with the New Polka Library, and you must free it with `poly_free`. This function calls `poly_add_constraints` to add at most $\mathcal{O}(N^2)$ constraints to the universal polyhedron. The returned polyhedron may have a slightly larger domain than the original octagon due to sound number conversions.

- `oct_t* oct_from_poly (poly_t* p)`

  Returns an octagon with the smallest domain containing the polyhedron `p`: it is guaranteed to be the *best approximation*. You can then manipulate the octagon with the Octagon Abstract Domain Library, and you must free it with `oct_free`. This function uses the frame representation of the polyhedron and has a $\mathcal{O}(N^2 \times M)$ time and $\mathcal{O}(N)$ memory cost, M being the size of the frame representation.

2.5 Utilities

The functions described in this section are not tied to octagons and can be used to instrument and monitor any C function. They were successfully used to debug and optimize the Octagon Abstract Domain Library.

2.5.1 Assertions

The library uses assertions to perform basic sanity checks:

- `OCT_ASSERT(c,text)`

  When the library is configured with the `--enable-assert` option (the `ENABLE_ASSERT` symbol is defined), the library checks the condition `c`. If the condition `c` is false, then it aborts after displaying the message `text` as well as the source file and line where the error occurred. Aborting the program is done by sending a `SIGABRT` signal; this allows easy access to the program state in a debugger, or via the core file.

  If the `--disable-assert` configuration option is used (`ENABLE_ASSERT` is not defined), this function does nothing.

In the Octagon Library, the assertion mechanism is mostly used to check arguments supplied by the user; thus we strongly recommend to always use the `--enable-assert` option. Disabling assertions will not result in significant speed increase.

2.5.2 Memory Management

The following useful functions and macros are defined:

- `void* oct_mm_malloc(size_t t)`

  Works as `malloc`, but may be monitored (see `oct_mm_alloc_t` below). All library functions use `oct_mm_malloc` instead of `malloc`. Do not mix blocks allocated with `oct_mm_malloc` and blocks allocated with `malloc`.

- `void* oct_mm_realloc(void* p, size_t t)`

  Works as `realloc`. The block `p` must have been allocated with `oct_mm_malloc`. If the block was allocated with `malloc`, then use `realloc` instead of `oct_mm_realloc`.

- `void oct_mm_free(void* p)`

  Works as `free`. The block `p` must have been allocated with `oct_mm_malloc`. If the block was allocated with `malloc`, then use `free` instead of `oct_mm_free`.

- `t* new_t (t)`

  Allocates a new object of type `t` in the heap with `oct_mm_malloc`. It must be freed with `oct_mm_free`.

- `t* new_n (t, size_t n)`

  Allocates an array of `n` objects of type `t` in the heap with `oct_mm_malloc`. It must be freed with `oct_mm_free`.

- `t* renew_n (t* c, t, size_t n)`

  Resizes the block pointed by `c` so that it contains `n` objects of type `t` with `oct_mm_realloc`. The block `c` must have been allocated using one of the preceding functions, not `malloc` nor `realloc`. The returned block must be freed with `oct_mm_free`.

When the `--enable-prof` configuration option is used, the symbol `ENABLE_MALLOC_MONITORING` is defined and all calls to the memory functions `oct_mm_malloc`, `oct_mm_realloc`, `oct_mm_free`, `new_t`, `new_n`, and `renew_n` are redirected through a memory monitoring routine that monitors memory consumption. You can then use the following interface.

- `mmalloc_t*`

  Abstract type of memory monitors. They are allocated by `oct_mmalloc_new`, and never deallocated. There is always a current monitor to which `oct_mm_malloc`-ed blocks are attached. When these blocks are `oct_mm_realloc`-ed, or `oct_mm_free`-ed, they affect the monitor they are attached to, not the current monitor. A monitor stores the number of calls to `oct_mm_malloc`, `oct_mm_realloc`, `oct_mm_free`, as well as the total sum of memory allocated, and the maximum this sum has reached.

- `mmalloc_t* oct_mmalloc_new ()`

  Creates a new monitor. Monitors are never deallocated because there may still be memory blocks attached to them. You must not call `oct_mm_free`, nor `free` on them.

- `void oct_mmalloc_print (mmalloc_t* mm)`

  Prints on the standard output the statistics for the monitor `mm`.

- `void oct_mmalloc_use (mmalloc_t* mm)`

  Makes `mm` the current monitor. It will stay the current monitor until the next call to `oct_mmalloc_use`.

- `mmalloc_t* oct_mmalloc_get_current ()`

  Returns the current monitor. Before any call to `oct_mmalloc_use`, there exists a default current monitor; you can retreive it using this function.

- `void oct_mmalloc_reset (mmalloc_t* mm)`

  Forgets about all the blocks attached to the monitor `mm` and resets all counters.

When the `--disable-prof` configuration option is used, the symbol `ENABLE_MALLOC_MONITORING` is not defined and `oct_mm_malloc`, `oct_mm_realloc`, and `oct_mm_free` are synonymous to `malloc`, `realloc`, and `free`.

IMPORTANT WARNING: when memory monitoring is enabled, you must call `oct_mm_free` only on blocks allocated with `oct_mm_malloc`, and `free` on blocks allocated with `malloc`. All the memory used by the Octagon Library is allocated through `oct_mm_malloc` to enable you to monitor the library memory usage. Blocks allocated by other libraries (like the C library) are allocated using implicitly `malloc`, so they must be freed with `free`.

One simple usage of monitors is to call `oct_mmalloc_print (mmalloc_get_current())` at the end of your program to get the total number of `malloc`s, and the maximum amount of memory used by the program. It can also be used to detect memory leaks (it is sometime customary to allocate some static structures at the beginning of the program and never deallocate them; they must not be taken into account when looking for memory leaks; `mmalloc_t` objects are one such example).

Beware that enabling memory profiling makes all calls to the Octagon Library thread-unsafe.

2.5.3 Timing

The library defines *clocks* to help you measure precisely elapsed time.

- `chrono_t`

  Type of clocks. This is a public structure, so you can safely allocate a new clock using simply `chrono_t a; oct_chrono_reset(&a);` .

- `void oct_chrono_reset (chrono_t* c)`

  Initializes or resets the clock. Elapsed time is reset to 0 and the clock is set in the stopped state.

- `void oct_chrono_start (chrono_t* c)`

  Starts the clock. You must not call `oct_chrono_start` on a clock that is already started.

- `void oct_chrono_stop (chrono_t* c)`

  Stops the clock. You must not call `oct_chrono_stop` on a clock that is already stopped. If you call `oct_chrono_start` and `oct_chrono_stop` several times, elapsed times are accumulated. You must call `oct_chrono_reset` to reset the elapsed time to 0.

- ```
  void oct_chrono_get (  chrono_t* c,
                         long* hour, long* min,
                         long* sec, long* usec   )
  ```

  Get the elapsed time of the clock in canonical form. You are guaranteed than $0 \leq$ `min` ≤ 59, $0 \leq$ `sec` ≤ 59, and $0 \leq$ `usec` ≤ 999.

- `void oct_chrono_print (chrono_t* c)`

  Prints the elapsed time on the standard output.

If the `--enable-prof` configuration option is used; the symbol `ENABLE_TIMING` is also defined and clocks are used to count the time elapsed in each library call, as well as the number of time each library function is called. It uses the following interface:
  ```

- `void oct_timing_enter (const char* name, unsigned key)`

  Called when entering a section named `name`. Also, a unique `key` must be given for each `name`, and this key must not exceed the size of the `timing_data` array defined in `oct_util.c`. Sections must be properly nested, and entering twice a section with the same key (recursive sections) is forbidden. Use with care.

- `void oct_timing_exit (const char* name, unsigned key)`

  Called when exiting a section named `name`, with unique key `key`. Use with care.

- `void oct_timing_print (const char* name)`

  Prints the number of time the section named `name` has been enterer, and the total elapsed time in it. You do not need to know the `key` associated.

  Two elapsed times are available for each section: *cumulative time* is the sum of time between all pairs `oct_timing_enter`, `oct_timing_exit` for the section; *self time* does not include the time elapsed in other sections entered within this section.

- `void oct_timing_print_all ()`

  Prints the number of time each section has been enterer, and the total elapsed time in each section.

- `void oct_timing_reset (const char* name)`

  Resets to 0 the number of time the section named `name` has been enterer, and the total elapsed time in it. You do not need to know the `key` associated.

- `void oct_timing_reset_all ()`

  Resets to 0 the number of time all the sections have been enterer, and the total elapsed time in them.

- `void oct_timing_clear ()`

  Call this function to free all the memory used to store the timing informations. There must not be any section currently opened.

When the `--enable-prof` configuration option is used, each library function corresponds to exactly one section named by the function's name. Thus, if you call `oct_timing_print_all()` at the end of your program, it will print how many time each library function was called, as well as the the elapsed time in them.

Beware that enabling time profiling makes all calls to the Octagon Library thread-unsafe.

Chapter 3

OCaml Binding API

This chapter describes the OCaml API used to access to the Octagon Abstract Domain Library. It is a simple binding layer that directly calls the C API. `oct.mli` and `oct.ml` contain the definition of the module `Oct`: abstract types, and external functions. The C implementation of external functions are defined in `oct_ocaml.c`; most are simply wrappers that call the C library, with proper treatment of OCaml `value` type. Thus, the OCaml API inherits all configuration-time choices made for the C API (profiling, debugging, New Polka support, underlying numerical domain). Some pretty-printers are also defined specially for the OCaml top-level.

Some functions are available only if some required OCaml library is also available. This is the case for providing octagons / polyhedra conversion (requires the New Polka `Poly` OCaml module). This is also the case for access to GMP and MPFR numbers (requires the Caml-GMP `Gmp` module [Mon]). In order to support this, `oct.ml` and `oct.mli` are generated automatically by cat-ing several `.ml` and `.mli` files in the `ocamllib` directory.

Unlike C, OCaml has automatic memory management, so there is no need for `free` or `copy` functions. Variables are not in-place modifiable and functions are *non-destructive* as it is usually the case for functional languages.

3.1 Initialization

- `init: unit -> bool`

  Always call this function before using the library (and once for each version of the library you use). Returns `true` if the library correctly initialized itself. When using machine floating-point internal representation, this tries to set the FPU rounding mode toward $+\infty$ in order to ensure soundness.

3.2 Numerical Domain

The choice of $\mathbb{I}$ and its underlying implementation are totally defined by the way the C library was configured; it cannot be changed by the OCaml library. All results regarding soundness and precision are exactly the same as for the C library.

Types `num` and `vnum` are used to specify numbers appearing in the constraint sets. `num` is the type of a single number, whereas `vnum` is the type of a number arrays (handled more efficiently than `num array` could possibly). Both are abstract types. You can construct them from integer, fraction, or floating-point values independently from the underlying numerical domain chosen to represent internally the constraints. Once constructed, you cannot manipulate them, but only pass them to octagons functions. Construct once and use many times to improve efficiency. You can use the regular OCaml comparison operators `=`, `<>`, `<`, `>`, `compare`, etc. to directly to compare `num` (but not `vnum`). Moreover, you can serialize and deserialize `num` and `vnum` (but beware that this is not yet implemented for all underlying numerical domains, in which case you will get an exception; also, deserialisation of objects serialised using a library version based on a different numerical underlying domain is not possible).

- `num_of_int:  int -> num`
  `num_of_frac:  int*int -> num`
  `num_of_float:  float -> num`

  Construct a number from an integer, a fraction (numerator and denominator), or a floating-point number.

- `num_infty:  unit -> num`

  Constructs the $+\infty$ number.

- `vnum_of_int:  int array -> vnum`
  `vnum_of_frac:  int*int array -> vnum`
  `vnum_of_float:  float array -> vnum`

  Construct a number vector from an integer, a fraction (numerator and denominator), or a floating-point number array.

- `vnum_of_int_opt:  int option array -> vnum`
  `vnum_of_frac_opt:  int*int option array -> vnum`

  Construct a number vector with infinite elements. `None` is converted into the $+\infty$ element, and `Some s` is converted into a number. There is no need for `None` when dealing with floating-points numbers as $+\infty$ already has a floating-point representation.

- `int_of_num:  num -> int option`
  `frac_of_num:  num -> int*int option`
  `float_of_num:  num -> float`

  Return an over-approximation of a number as an integer, fraction, or floating-point number, or `None` if there is no such over-approximation (this means the number should be considered $+\infty$). There is no need for `None` when dealing with floating-points numbers as $+\infty$ already has a floating-point representation.

- `int_of_vnum:  vnum -> int option array`
  `frac_of_vnum:  vnum -> int*int option array`
  `float_of_vnum:  vnum -> float array`

  The same as above, but for number vectors instead of single numbers.

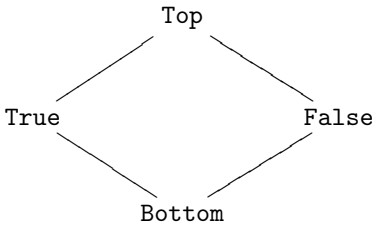
- `num_of_mpz:  Gmp.Z.t -> num`
  `num_of_mpq:  Gmp.Q.t -> num`
  `num_of_mpfr:  Gmp.FR.t -> num`
  `vnum_of_mpz:  Gmp.Z.t array -> vnum`
  `vnum_of_mpq:  Gmp.Q.t array -> vnum`
  `vnum_of_mpfr:  Gmp.FR.t array -> vnum`
  `vnum_of_mpz_opt:  Gmp.Z.t option array -> vnum`
  `vnum_of_mpq_opt:  Gmp.Q.t option array -> vnum`
  `vnum_of_mpfr_opt:  Gmp.FR.t option array -> vnum`
  `mpz_of_num:  num -> Gmp.Z.t option`
  `mpq_of_num:  num -> Gmp.Q.t option`
  `mpfr_of_num:  num -> Gmp.FR.t option`
  `mpz_of_vnum:  vnum -> Gmp.Z.t option array`
  `mpq_of_vnum:  vnum -> Gmp.Q.t option array`
  `mpfr_of_vnum:  vnum -> Gmp.FR.t option array`

  These functions are only defined when the `Gmp` module is available [Mon].

3.3 Boolean Lattice

- `type tbool`

  Defines the boolean lattice:



  `Bottom` is returned when the answer to a question is undefined, `Top` when it is not known.

3.4 Octagons

Most octagons functions are direct OCaml equivalent of the corresponding C library call. They have the same memory and time cost. Thus, most functions are simply sketched here; please refer to Section 2.4 for more detailed informations.

You can serialize and deserialize octagons (`oct` and `moct`) (but beware that this is not yet implemented for all underlying numerical domains, in which case you will get an exception; also, deserialisation of octagons serialised using a library version based on a different numerical underlying domain is not possible).

- `type oct`

  This is the *abstract type* of octagons. Octagons are *static* (not in-place modifiable), and allocated in the C heap by the C library. They can be safely copied with `let a = b in`. They can NOT be compared with the generic OCaml operators (which would suppose a total ordering, while the octagon ordering is only partial).

3.4.1 Octagon Creation

There is no need to free explicitly the octagons; this is done automatically by the OCaml garbage collector.

- `empty:` N`:int -> oct`

  Returns a new octagon of dimension N with an empty domain: $\emptyset$.

- `universe:` N`:int -> oct`

  Returns a new octagon of dimension N with an full domain: $\mathbb{I}^N$.

3.4.2 Queries

- `dim: oct -> int`

  Returns the dimension N of the octagon.

- `nbconstraints: oct -> int`

  Returns the number of constraints in the octagon. If the octagon is known to have an empty domain, returns 0. The count may include redundant constraints; but constraints of the form $(v_i - v_i \leq 0)$ and $(v_i - v_j \leq +\infty)$ are not counted.

3.4.3 Tests

- `is_empty: oct -> bool`

  Returns `true` if and only if the octagon has an empty domain.

- `is_empty_lazy: oct -> tbool`

  This lazy version of emptiness checking does not perform any computation. It returns `True` if it knows the octagon has an empty domain, `False` if it knows it has a non-empty domain, and `Top` if checking for emptiness would require some computation.

- `is_universe: oct -> bool`

  Returns `true` if and only if the octagon has a full domain $\mathbb{I}^N$.

- `is_included_in: oct -> oct -> bool`

  Returns `true` if and only if the first octagon is included in or equal to the second octagon.

- `is_included_in_lazy: oct -> oct -> tbool`

  This lazy version of inclusion checking does not perform any computation. It may return `True` (the first argument is included in or equal to the second one), `False` (the first argument is not included in nor equal to the second one), or `Top` (the accurate test would require some computation).

- `is_equal: oct -> oct -> bool`

  Returns `true` if and only if the two octagons have exactly the same domain.

- `is_equal_lazy: oct -> oct -> tbool`

  This lazy version of equality checking does not perform any computation. It may return `True` (the two octagons have the same domain), `False` (the two octagons do not have equal domains), or `Top` (the accurate test would require some computation).

- `is_in: oct -> vnum -> bool`

  Returns `true` if and only if the second argument is the coordinate of a point that is within the domain of the specified octagon. The second argument must contain exactly N elements, where N is the dimension of the specified octagon.

3.4.4 Operators

- `inter: oct -> oct -> oct`

  Returns a new octagon the domain of which is the intersection of the two octagons specified.

- `union: oct -> oct -> oct`

  Returns a new octagon the domain of which is the smallest domain containing both octagons.

- `widening: m:oct -> n:oct -> w:widen t -> oct`

  Returns a new octagon representing $\mathtt{m} \triangledown (\mathtt{n}^\bullet)$, where the widening operator $\triangledown$ is chosen by `w`:
  - `WidenFast`, fast widening, as presented in[Min01b], unstable coefficients are replaced by $+\infty$;

- `WidenZero`, slower but more accurate widening, unstable coefficients are replaced by 0, and then $+\infty$;
- `WidenUnit`, even slower and even more accurate widening, unstable coefficients are replaced by -1, 0, 1, and then $+\infty$;
- `WidenSteps` of `vnum`, uses the given set of numbers (in increasing order) to gradually loosen coefficients before replacing them by $+\infty$.
- `PreWiden`, not a proper widening, but a degenerate hull that can be safely inserted into a widening sequence (as long as proper widenings are called infinitely often) to improve somewhat the precision of an analysis.

■ `narrowing: m:oct -> n:oct -> oct`

Returns a new octagon representing $(\mathtt{m}^\bullet)\triangle(\mathtt{n}^\bullet)$.

3.4.5 Transfer Functions

■ `forget: oct -> k:int -> oct`

Returns a new octagon where the variable $v_{\mathtt{k}}$ has forgotten its value. Corresponds to both instructions $v_{\mathtt{k}} \leftarrow ?$ and $v_{\mathtt{k}} \rightarrow ?$.

■ `add_bin_constraints: oct -> c:constr array -> oct`

Returns a new octagon where the binary constraints `c` have been added. Each element of `c` can be of the following form:

- `PX of int*num` represents $(x \leq c)$;
- `MX of int*num` represents $(-x \leq c)$;
- `PXPY of int*int*num` represents $(x + y \leq c)$;
- `PXMY of int*int*num` represents $(x - y \leq c)$;
- `MXPY of int*int*num` represents $(-x + y \leq c)$;
- `MXMY of int*int*num` represents $(-x - y \leq c)$.

■ `assign_var: m:oct -> x:int -> tab:vnum -> oct`

Returns a new octagon representing the domain of `m` after the linear variable assignment $v_{\mathtt{x}} \leftarrow \left(\sum_{i=0}^{N-1} v_i \cdot \mathtt{tab}[i]\right) + \mathtt{tab}[N]$.

■ `interv_assign_var: m:oct -> x:int -> tab:vnum -> oct`

Same as `assign_var`, but allowing *interval* coefficients instead of constant ones: $v_{\mathtt{x}} \leftarrow \left(\sum_{i=0}^{N-1} v_i \cdot [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]\right) + [-\mathtt{tab}[2N+1], \mathtt{tab}[2N]]$.

■ `substitute_var m:oct -> x:int -> tab:vnum -> oct`

Returns a new octagon representing the domain of `m` after the linear variable substitution $v_{\mathtt{x}} \rightarrow \left(\sum_{i=0}^{N-1} v_i \cdot \mathtt{tab}[i]\right) + \mathtt{tab}[N]$. It is the *backward* semantics of the corresponding assignment.

■ `interv_substitute_var: m:oct -> x:int -> tab:vnum -> oct`

Same as `substitute_var`, but allowing *interval* coefficients instead of constant ones: $v_{\mathtt{x}} \rightarrow \left(\sum_{i=0}^{N-1} v_i \cdot [-\mathtt{tab}[2i+1], \mathtt{tab}[2i]]\right) + [-\mathtt{tab}[2N+1], \mathtt{tab}[2N]]$.

- `add_constraint: m:oct -> tab:vnum -> oct`

  Returns a new octagon representing the domain of `m` with the linear constraint added $\left(\sum_{i=0}^{N-1} v_i \cdot \texttt{tab}[i]\right) + \texttt{tab}[N] \geq 0$.

- `interv_add_constraint: m:oct -> tab:vnum -> oct`

  Same as `add_constraint`, but allowing *interval* coefficients instead of constant ones: $\left(\sum_{i=0}^{N-1} v_i \cdot [-\texttt{tab}[2i+1], \texttt{tab}[2i]]\right) + [-\texttt{tab}[2N+1], \texttt{tab}[2N]] \geq 0$.

- `time_flow: m:oct -> nmin:num -> nmax:num -> tab:vnum -> oct`

  Let time flow for an amount $t \in [\texttt{nmin}, \texttt{nmax}]$, where $0 \leq \texttt{nmin} \leq \texttt{nmax}$: $\forall i,\ v_i \leftarrow v_i + [-\texttt{tab}[2i+1], \texttt{tab}[2i]] \times t$. For each variable v_i, the range of evolution rates is given by the non-empty interval $[-\texttt{tab}[2i+1], \texttt{tab}[2i]]$.

3.4.6 Change of Dimensions

Add and remove dimensions at the end of $\mathcal{V}$.

- `add_dims_and_embed: o:oct -> n:int -> oct`

  Returns a new octagon with `n` more dimensions, and with domain $\mathcal{D}^+(\texttt{o}) \times \mathbb{I}^{\texttt{n}}$.

- `add_dims_and_project: oct -> int -> oct`

  Returns a new octagon with `n` more dimensions, and with domain $\mathcal{D}^+(\texttt{o}) \times \{\ 0\ \}^{\texttt{n}}$.

- `del_dims: oct -> int -> oct`

  Returns a new octagon with `n` dimensions removed.

Add and remove dimensions at arbitrary positions in $\mathcal{V}$.

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

  Specifies that `nbdims` variables must be added or removed just before the variable at position `pos`. All positions are counted from 0, and always with respect to the variable set of the *argument* octagon.

- `add_dims_and_embed_multi: oct -> dimsup array -> oct`

  Returns a new octagons where uninitialized dimensions at positions specified by `dimsup` have been inserted. (The `pos` fields in the `dimsup` array must be sorted in *strictly increasing order*.)

- `oct_add_dims_and_project_multi: oct -> dimsup array -> oct`

  Adds zero-initialized variables in $\mathcal{V}$.

- `del_dims_multi: oct -> dimsup array -> oct`

  Removes variables from $\mathcal{V}$.
  ```

Add and remove dimensions with a permutation of variables. The permutation is given as an array `t: int array` of variables indices: each variable `i` (starting from 0) appearing in a constraint is replaced with the variable `t.(i)`.

- `add_permute_dims_and_embed: oct -> n:int -> t:int array -> oct`

  Adds `n` uninitalized variables at the end of $\mathcal{V}$, then applies the permutation `t`. `t` must be a permutation of 0 to 0 to `n + dim m - 1`.

- `oct_add_permute_dims_and_project: oct -> int -> int array -> oct`

  Adds zero-initialized variables at the end of $\mathcal{V}$, then applies a permutation.

- `permute_del_dims: oct -> n:int -> t:int array -> oct`

  Applies the permutation `t` before removing the last `n` variables. `t` must be a permutation of 0 to `dim m - 1`.

3.4.7 Minimized Octagons

- `type moct`

  This is the *abstract type* of minimized octagons. As `oct` objects, they are static and allocated in the C heap by the C library. They can be safely copied with `let a = b in`.

- `m_to_oct: oct -> moct`

  Constructs an octagon from a minimized octagon.

- `m_from_oct: moct -> oct`

  Constructs a minimized octagon from a regular octagon. This implies performing the $\mathcal{O}(N^3)$ time cost minimization algorithm.

- `m_dim: moct -> int`

  Returns the dimension N of the minimized octagon.

- `m_is_empty: moct -> bool`

  Returns `true` if and only if the minimized octagon represents an empty domain.

- `m_is_equal: moct -> moct -> bool`

  Returns `true` if and only if the two minimized octagons have the same domain.

3.4.8 Interval functions

These functions are used to get or set the interval bound of one or all variables in an octagon.

- `get_box: oct -> vnum`

  Get the bounds for all the variables in the octagon. After conversion to an array `a`, using a `xxx_of_vnum` function, the interval of the variable i is in $[-\mathtt{a}.(2i+1), -\mathtt{a}.(2i)]$.

- `from_box: vnum -> oct`

  Construct an octagon from a box. If `n` is the number of dimension of the desired octagon, the array `t` that was used to construct the `vnum` must be of length 2`n`, and such that the interval $[-\texttt{a}.(2i+1), -\texttt{a}.(2i)]$ is the interval of the variable i.

- `get_bounds: oct -> int -> num*num`

  Get the upper and lower bounds of a variable in an octagon. The pair of `num` (`a`, `b`) represents the interval $[-\texttt{b}, \texttt{a}]$.

- `set_bounds: oct -> int -> num*num -> oct`

  Return a new octagon identical to the first argument, with the the upper and lower bounds of one variable set (and all other informations about this variable lost). The pair of `num` (`a`, `b`) represents the interval $[-\texttt{b}, \texttt{a}]$.

3.4.9 Perturbation

- `add_epsilon: oct -> num -> oct`

  `add_epsilon o epsilon` returns `o`, where all constraints $\pm x \pm y \leq a$ are replaced by $\pm x \pm y \leq a + \texttt{epsilon}|a|$. Thus, the returned octagon is slightly larger. `epsilon` must be positive. The normal form is lost.

- `add_epsilon_max: oct -> num -> oct`

  As `add_epsilon`, but all constraints are replaced by $\pm x \pm y \leq a + \texttt{epsilon} \times M$. where $M = \max\{\ |m| \mid \pm x \pm y \leq m,\ m \neq +\infty\ \}$. `epsilon` must be positive.

  $\mathcal{O}(N^2)$ time cost. The normal form is lost.

- `add_epsilon_bin: oct -> oct -> num -> oct`

  Binary version of `add_epsilon_max`, where only the non-stable constraints in the second argument are enlarged by $\texttt{epsilon} \times M$. This is a form of widening. `epsilon` must be positive.

  $\mathcal{O}(N^2)$ time cost. The normal form is lost.

3.4.10 Pretty-Printers

Note that pretty-printers are most useful when using the OCaml top-level ‘`oct-config --mltop`‘.

- `numprinter: num -> unit`

  Pretty-printer for objects of type `num`. Use `#install_printer Oct.numprinter;;` to install it in the top-level.

- `fnumprinter: Format.formatter -> num -> unit`

  Same as `numprinter`, but uses any formatter instead of `Format.std_formatter`.

- `vnumprinter: vnum -> unit`

  Pretty-printer for objects of type `vnum`. Use `#install_printer Oct.vnumprinter;;` to install it in the top-level.

- `fvnumprinter: Format.formatter -> vnum -> unit`

  Same as `vnumprinter`, but uses any formatter instead of `Format.std_formatter`.

- `octprinter: (int -> string) -> oct -> unit`

  Pretty-printer for objects of type `oct`. The first argument is a callback telling to `octprinter` the name of each variable, from `0` to `(dim o)-1`. You can use `let print2 = Oct.octprinter (fun i -> "v"^(string_of_int i));;` and `#install_printer print2;;`, for example, to install it in the top-level.

- `foctprinter: (int -> string) -> Format.formatter -> oct -> unit`

  Same as `octprinter`, but uses any formatter instead of `Format.std_formatter`.

- `foctnewprinter: (int -> string) -> Format.formatter -> oct -> oct -> unit`

  This pretty-printer takes *two* octagons (with the same size) as argument and compares the upper bound of each constraint in one octagon with the upper bound of the corresponding constraint in the second one. This function only prints the second version of constraints that are not the same.

- `foctdiffprinter: (int -> string) -> Format.formatter -> oct -> oct -> unit`

  Same as `foctdiffprinter`, but prints both the old (first octagon) and new version (second octagon) of constraints that are not the same, instead of only the new version.

- `moctprinter: (int -> string) -> moct -> unit`

  Pretty-printer for objects of type `moct`. You can use `let mprint2 = Oct.moctprinter (fun i -> "v"^(string_of_int i));;` and `#install_printer mprint2;;`, for example, to install it in the top-level.

- `fmoctprinter: (int -> string) -> Format.formatter -> moct -> unit`

  Same as `moctprinter`, but uses any formatter instead of `Format.std_formatter`.

3.5 Interface with the New Polka Library

If you have the New Polka Library [Jea] installed, together with its OCaml binding, it enables some conversion functions between objects of type `Oct.oct` and `Poly.t`.

Do not forget to call `Polka.initialize` before using these two functions, as well as all functions in the `Polka` module.

- `to_poly: oct -> Poly.t`

  Returns a new polyhedron that has the same domain (or a slightly larger one because of approximate, but sound, numerical type conversion).

- `from_poly: Poly.t -> oct`

  Returns the octagon with the smallest domain that contains the specified polyhedron.

3.6 Utilities

Most utility functions of Section 2.5 do not have a corresponding OCaml version. We only kept here two functions that are useful to monitor the time and memory consumption of the library.

- `memprint: unit -> unit`

  Prints on the standard output the memory usage for the C heap containing the octagon internal structures. The C library must have been configured with the `--with-prof` option (`ENABLE_MALLOC_MONITORING` symbol defined). It is equivalent to the C call `oct_mmalloc_print (mmalloc_get_current())`.

- `timeprint: unit -> unit`

  Prints on the standard output the elapsed time in the C functions of the library. The C library must have been configured with the `--with-prof` (`ENABLE_TIMING` symbol defined). It is equivalent to the C call `oct_timing_print_all ()`.

3.7 Low-level Binding API

There is no way for the OCaml code to access directly to the internal structure of an octagon. However, you may wish to write some C functions that manipulate octagons at low-level (by including the `oct_private.h` header), and then pass the octagon to some OCaml code. This facility is provided by the file `oct_ocaml.h`, which must be included by your C source files.

In order to do this properly, you must be familiar with the C and OCaml APIs to the Octagon Library, the internal representation of octagons, and the standard OCaml/C interface described in the OCaml Reference Manual [La].

- ```
  value  Val_int  (int a)
  int    Int_val  (value v)
  ```

  These functions are defined in `caml/mlvalues.h` to convert between C integers (`long`) and OCaml integers (`integer`).

  You can use them, to convert between C `tbool_t` and OCaml `Oct.tbool` types.

- `num_t* Num_val (value v)`

  This gives a pointer to the `num_t` number an OCaml value of type `Oct.num` contains.

- ```
  vnum_t
  vnum_t* Vnum_val (value v)
  ```

  `vnum_t` is a structure that holds a number vector. `Vnum_val` gives back the structure an OCaml value of type `Oct.vnum` contains. The `nb` field gives the vector length, and the `n` field is a pointer to a C array of `nb` numbers `num_t`.

- `value Val_oct (oct_t* o)`

  Returns a newly constructed OCaml value of type `Oct.oct` that contains the octagon `o`. This value is a *custom block* that simply contains a pointer equal to `o`; it points in the C heap, not in the OCaml heap, as the internal representation of the octagon has been allocated by the C library. You must not call `oct_free(o)` after the call `Val_oct(o)`: it will be called automatically when the OCaml garbage collector discovers that the returned OCaml object is no longer used. Each custom block in the OCaml heap with a pointer to `o` counts as one reference count.

- `oct_t* Oct_val(value v)`

  Returns the octagon represented by the OCaml value `v` of type `Oct.oct`.

- `value Val_moct (moct_t* o)`

  Returns a newly constructed OCaml value of type `Oct.moct` that contains the minimized octagon `o`. It works exactly as `Val_oct`.

- `moct_t* Moct_val(value v)`

  Returns the minimized octagon represented by the OCaml value `v` of type `Oct.moct`.

Chapter 4

Analyzer Example

This chapter describes the analyzer example. It is written in OCaml and uses the Octagon Abstract Domain Library to perform a basic analysis on a toy programming language.

4.1 How to Use the Analyzer

The analyzer example is compiled automatically with the library if the OCaml support is enabled. The executables are named `octanal` (for the byte-code version) and `octanalopt` (for the native version). The analyzer is run by simple invocation on the command-line. It takes its input program on the standard input (see Sect. 4.3) and does not accept any form of command-line option.

The analyzer produces several outputs, in a varity of formats:

- `result.txt`: text output of the original text program, with program point number added in bold, followed by the list of invariants octagons discovers at each control point;
- `result.html`: an HTML version of the previous output, with some coloring and links to jump between the program text and the printed invariants;
- `result.gdl`: a GDL file that describes the control-flow graph annotated with the invariants; this control-flow graph can be viewed with AbsInt's AiSee software (`http://www.absint.com/aisee/`);
- `result.debug`: (if debugging is enabled) a text file describing the computation trace together with the partial invariants discovered along the iterations.

There also exists a web-based version of the analyzer (using CGI scripts written in OCaml thanks to the OCamlHTML library) that can be queried interactively at `http://cgi.di.ens.fr/cgi-bin/mine/octanalhtml-ndi/octanalweb`.

4.2 Source Files

The analyzer comprises the following files:

- `oct_anal_lex.mll` lexer (using `ocamllex`)
- `oct_anal_yacc.mly` parser (using `ocamlyacc`)
- `oct_anal_syn.ml` abstract syntax tree and other global variables and types
- `oct_anal_core.ml` core analyzer functions
- `oct_anal.ml` main program that calls functions in `oct_anal_core.ml`

- `oct_anal_examples` a few examples program to be analyzed

4.3 Language Syntax

The analyzer uses a toy programming language that is a simply a `while`-language, with no procedure, no array, no pointer, and only finitely many numerical variables.

- The entry of the analyzer is a list of programs, each program having a name *id*:
  `program` *id*
  `beginprogram`
  stats
  `endprogram`

- A statement is:

stat	==	*id* `=` *iexpr*	assignment
	|	`assert` *bexpr*	assertion
	|	`if` *bexpr* `then` *block*	some tests
	|	`if` *bexpr* `then` *block* `else` *block*	

  Variables are all local to the program and need not to be declared.

- Statement lists are semicolon-separated (do not put a semicolon after the last statement):

stats	==	*stat*
	|	*stat*; *stats*

- Blocks are either a single statement, or a statement list enclosed in `begin`/`end` keywords:

block	==	*stat*
	|	`begin` *stats* `end`

- A numerical expression is (with standard precedence):

iexpr	==	*id*	variable
	|	*num*	floating-point constant
	|	`random`	random integer
	|	*iexpr* + *iexpr*	unary and binary arithmetic operators
	|	*iexpr* − *iexpr*	
	|	*iexpr* ∗ *iexpr*	
	|	− *iexpr*	

- A boolean expression is (with standard precedence):

bexpr	==	`true`	
	|	`false`	
	|	`brandom`	random boolean value
	|	*iexpr* > *iexpr*	some tests
	|	*iexpr* < *iexpr*	
	|	*iexpr* = *iexpr*	
	|	*iexpr* ! = *iexpr*	
	|	*iexpr* >= *iexpr*	
	|	*iexpr* <= *iexpr*	
	|	*bexpr* `and` *bexpr*	unary and binary boolean operators
	|	*bexpr* `or` *bexpr*	
	|	`not` *bexpr*	

  WARNING: >, <, and ! = have an *integer* semantics, that is to say $a > b \iff a >= b + 1$, etc. To compare floats, use only =, >= and <=!

4.4 Analysis Method

The program is first cut into several program point: a program point at the beginning and at the end of each statement block, a program point between two statements, and a program point at the end of each `if-then` or `if-then-else` instructions (control-flow join point).

An octagon is associated to each program point. It is initially empty, except for the first program point which has a universe octagon.

The analyzer then uses a simple forward analysis method to propagate informations from program points to program point, applying transfer functions until a fix-point is reached.

It uses a simple *work-list* algorithm that maintains the set of control points whose state need to be recomputed. The first control point of the list is picked, it is updated by recomputing the transfer function, and all its successor control points are marked dirty and added *at the end* of the work-list. In fact, when a control point that has several predecessors is at the head of the work-list, it not chosen directly, but

postponed so that there is more chance that all its predecessors have been updated; intuitively, it saves some computation.

In order to reach the fix-point in finite time, a widening application is performed at the beginning of each `while` statement.

Finally, if the New Polka library and its OCaml binding are present, the analyzer can use polyhedra instead of octagons (see the file `oct_anal_core.ml`).

Chapter 5

Internal Structure

This chapter describes in details the internal structure and implementation choices. It may be skipped at first lecture.

5.1 Abstract Elements Representation (`oct_sem.c`)

In order to understand the representation of abstract elements, one have to be familiar with the Octagon Abstract Domain algorithms, as described in [Min06]. However, the internal representation of abstract elements differs slightly compared to what is described in theses papers.

5.1.1 Original Representation

Let $\mathcal{V} = \{v_0, \ldots, v_{N-1}\}$ be the set of N numerical variables we wish to manipulate.

In the original work [Min01b], we introduced an auxiliary set of $2N$ variables $\mathcal{X}' = \{x_0, \ldots x_{2N-1}\}$ with the semantics $x_{2i} = v_i$, $x_{2i+1} = -v_i$. It allows to represent constraints of the form $(\pm v_i \pm v_j \leq c)$ by *potential constraints*[1] in $\mathcal{X}$. Even interval constraints $(v_i \leq c)$, $(-v_j \leq c)$ can be represented in $\mathcal{X}$ by $(x_{2i} - x_{2i+1} \leq 2c)$ and $(x_{2j+1} - x_{2j} \leq 2c)$. This is most useful because there exists a well-known theory, called *Separation Theory*, developed by Pratt [Pra77] dealing with satisfiability of potential constraint sets. This theory was enhanced in [Min01a] in order to build a full numerical abstract domain. Thus, [Min01b] chose to perform the transformation from $\mathcal{V}$ to $\mathcal{X}$ to reuse and adapt the results of [Min01a] to build the Octagon Abstract Domain. This choice also resulted in compact mathematical presentation and easy proofs of theorems, especially for normal form and lattice operators. See the journal version [Min06] of the [Min01b] paper for the latest presentation of all algorithms toghether with their proof of correctness.

However, this representation has a drawback when it comes to implementing: it has redundancy. For example, the constraint $(v_i - v_j \leq c)$ can be represented as $(x_{2i} - x_{2j} \leq c)$ or $(x_{2j+1} - x_{2i+1} \leq c)$. This results in almost twice memory consumption as necessary. It also make some transfer functions less easy to understand (as we must take care to change the two equivalent representations of a constraint at the same time).

We now recall the matrix and index notation of the original representation. We denote by $\mathbf{m}^+$ a $2N \times 2N$ matrix that represent the constraint set $\{\ x_j - x_i \leq \mathbf{m}^+_{ij}\ \}$. We denote by $\bar{\imath}$ the number $i \oplus 1$—where $\oplus$ is the bitwise exclusive or. Thus, if x_i corresponds to v_j, then $x_{\bar{\imath}}$ corresponds to $-v_j$, and if x_i corresponds to $-v_j$, then $x_{\bar{\imath}}$ corresponds to v_j. A matrix $\mathbf{m}^+$ is said to be *coherent* if and only if $\forall i, j,\ \mathbf{m}^+_{ij} = \mathbf{m}^+_{\bar{\jmath}\bar{\imath}}$. Coherence means that any two redundant constraints in $\mathbf{m}^+$ $(\epsilon_1 v_i - \epsilon_2 v_j \leq c)$ and $((-\epsilon_2)v_j - (-\epsilon_1)v_i \leq d)$ are such that $c = d$. All matrices are supposed to be coherent in [Min01b].

5.1.2 Modified Representation (`oct_t`)

In our implementation, we chose to store only elements $\mathbf{m}^+_{ij}$ such that $i = \bar{\jmath}$ or $i \geq j$. The memory layout is the almost the lower triangle of a matrix, and can be stored with roughly half memory consumption.

[1] A potential constraint is a constraint of the form $(x - y \leq c)$. The term *potential* comes from the remark that solutions of such constraint sets are defined to a constant.

	0	1	2	3	4	5
0	$\mathbf{m}^+_{0,0}$	$\mathbf{m}^+_{0,1}$				
1	$\mathbf{m}^+_{1,0}$	$\mathbf{m}^+_{1,1}$				
2	$\mathbf{m}^+_{2,0}$	$\mathbf{m}^+_{2,1}$	$\mathbf{m}^+_{2,2}$	$\mathbf{m}^+_{2,3}$		
3	$\mathbf{m}^+_{3,0}$	$\mathbf{m}^+_{3,1}$	$\mathbf{m}^+_{3,2}$	$\mathbf{m}^+_{3,3}$		
4	$\mathbf{m}^+_{4,0}$	$\mathbf{m}^+_{4,1}$	$\mathbf{m}^+_{4,2}$	$\mathbf{m}^+_{4,3}$	$\mathbf{m}^+_{4,4}$	$\mathbf{m}^+_{4,5}$
5	$\mathbf{m}^+_{5,0}$	$\mathbf{m}^+_{5,1}$	$\mathbf{m}^+_{5,2}$	$\mathbf{m}^+_{5,3}$	$\mathbf{m}^+_{5,4}$	$\mathbf{m}^+_{5,5}$

Because $i < j \Longrightarrow \bar{j} \geq \bar{i}$ or $i = \bar{j}$, we have the property that when $\mathbf{m}^+_{ij}$ does not appear in our modified representation, then $\mathbf{m}^+_{\bar{j}\bar{i}}$ does: thus, the upper triangle of the matrix can be recovered by coherence. Our algorithm have been slightly modified to work on this halved representation, but proofs of their correctness remain valid. With this representation, the strong closure algorithm is a bit harder to understand, but most transfer functions are easier to read.

The half-matrix is not stored as a full matrix, but as a flat array of size `matsize(`N`)`; $\mathbf{m}^+_{ij}$ is stored in `c[matpos2(i,j)]`, where:

```
#define matsize(n)     (2*(size_t)(n)*((size_t)(n)+1))
#define matpos(i,j)    ((size_t)(j)+(((size_t)(i)+1)*((size_t)(i)+1))/2)
#define matpos2(i,j)   ((i)<(j)?matpos((j^1),(i^1)):matpos(i,j))
```

Remark that elements of the form $\mathbf{m}^+_{ii}$ correspond to constraints of the form $0 \leq c$ which do not carry much information. However we keep them in our representation for two reasons: it is simpler to handle a connected structure, and these elements will prove useful to check for emptiness in the modified strong closure algorithm.

An octagon can be in one of three different states, as specified by the `state` field: `OCT_EMPTY`, `OCT_CLOSED`, `OCT_NORMAL`. `OCT_EMPTY` states that the octagon has an empty domain. `OCT_CLOSED` means that the matrix is already in closed form. In all other cases (we do not now if the matrix is empty or in normal form) the state is `OCT_NORMAL`.

Here is the detailed structure, as one can see in the `oct_private.h` file:

```
struct oct_tt {
  var_t           n;       /* number of variables, aka dimension */
  int             ref;     /* reference counting */
  oct_state       state;   /* is it empty, closed, etc. ? */
  struct oct_tt*  closed;  /* pointer to the closed version, or NULL */
  num_t*          c;       /* the matrix, contains matsize(n) elements */
}
```

See Section 5.4 for an explanation of the fields `ref` and `closed`. `c` contains the half-matrix array; it is only allocated if `state` is not `OCT_EMPTY`.

5.2 Modified Strong Closure Algorithm (`oct_close`)

The strong closure algorithm is the core algorithm of the library. It is also the more complex and most costly one.

The strong closure algorithm—which we will sometime simply call *closure algorithm* because there is no ambiguity—presented in [Min01b] has been modified in two ways:

- it has been adapted to the new representation of octagons;
- it does no longer need to be called with a matrix the domain of which is not empty, it performs the emptiness check itself.

Here, we recall the original strong closure algorithm $\mathbf{m}^+ \to (\mathbf{m}^+)^\bullet$:

$$\left\{ \begin{array}{lcl} \mathbf{m}_0^+ & = & \mathbf{m}^+, \\ \mathbf{m}_{k+1}^+ & = & S^+(C_{2k}^+(\mathbf{m}_k^+)) \quad \forall k,\ 0 \leq k < N, \\ (\mathbf{m}^+)^\bullet & = & \mathbf{m}_N^+, \end{array} \right.$$

where C_k^+ is defined, $\forall k$, by:

$$\left\{ \begin{array}{l} \left[C_k^+(\mathbf{n}^+)\right]_{ii} = 0, \\ \left[C_k^+(\mathbf{n}^+)\right]_{ij} = \min(\quad \mathbf{n}_{ij}^+,\ (\mathbf{n}_{ik}^+ + \mathbf{n}_{kj}^+),\ (\mathbf{n}_{i\bar{k}}^+ + \mathbf{n}_{\bar{k}j}^+), \\ \qquad\qquad\qquad (\mathbf{n}_{ik}^+ + \mathbf{n}_{k\bar{k}}^+ + \mathbf{n}_{\bar{k}j}^+),\ (\mathbf{n}_{i\bar{k}}^+ + \mathbf{n}_{\bar{k}k}^+ + \mathbf{n}_{kj}^+) \ ) \end{array} \right.$$

and S^+ is defined by:

$$[S^+(\mathbf{n}^+)]_{ij} = \min(\ \mathbf{n}_{ij}^+,\ (\mathbf{n}_{i\bar{\imath}}^+ + \mathbf{n}_{\bar{\jmath}j}^+)/2\ )\ .$$

We say that $\mathbf{m}$ is *strongly closed* if and only if:

- $\mathbf{m}^+$ is *coherent*: $\forall i, j,\ \mathbf{m}_{ij}^+ = \mathbf{m}_{\bar{\jmath}\bar{\imath}}^+$;
- $\mathbf{m}^+$ is *closed*: $\forall i,\ \mathbf{m}_{ii}^+ = 0$ and $\forall i, j, k,\ \mathbf{m}_{ij}^+ \leq \mathbf{m}_{ik}^+ + \mathbf{m}_{kj}^+$;
- $\forall i, j,\ \mathbf{m}_{ij}^+ \leq (\mathbf{m}_{i\bar{\imath}}^+ + \mathbf{m}_{\bar{\jmath}j}^+)/2$.

An important theorem of [Min01b] is that the strong closure algorithm applied to a satisfiable constraint set gives a strongly closed matrix if $\mathbb{I} = \mathbb{Q}$ or $\mathbb{R}$. Another important theorem is that this strongly closed form is a normal form.

If $\mathbb{I} = \mathbb{Z}$, the strong closure algorithm *does not lead to a normal form.*

5.2.1 Emptiness Test

Thanks to the following theorem, emptiness testing can be done with the strong closure algorithm:

Theorem 1. *Let $\mathbf{m}^+$ be a matrix the domain of which can be empty or non empty; we have:*

- *either $\mathcal{D}^+(\mathbf{m}^+) \neq \emptyset$ and $(\mathbf{m}^+)^\bullet$ is strongly closed,*
- *or $\mathcal{D}^+(\mathbf{m}^+) = \emptyset$ and $\exists i,\ (\mathbf{m}^+)_{ii}^\bullet \neq 0$.*

■

Thus, $(\mathbf{m}^+)^\bullet$ is computed using the closure algorithm. Then, $(\mathbf{m}^+)^\bullet$ is checked for indices i such that $(\mathbf{m}^+)_{ii}^\bullet \neq 0$ and the algorithm answer both questions: "is the domain empty ?" and "what is the normal form ?".

The stand-alone emptiness checking has been removed, as it was generally followed by a strong closure computation, and was not much more efficient than the strong closure algorithm itself.

If $\mathbb{I} = \mathbb{Z}$, this emptiness checking can return `false` for a matrix with an empty domain; however, if it returns `true`, the domain is empty.

5.2.2 Incremental Strong Closure Algorithm (`oct_close_incremental`)

This is an incremental version of the preceding strong closure algorithm. It allows to recover in $\mathcal{O}(n^2)$ time a closed form from an octagon that is closed except for lines and columns corresponding to *one* variable.

This is useful, for example, to quickly compute a closed form after an assignment or guard, if the original argument where closed. Indeed, the incremental closure is presently used internally by the transfer functions `oct_add_bin_constraints`, `oct_add_constraint` `oct_assign_variable`, `oct_substitute_variable`, and `oct_interv_assign_variable` whenever possible.

5.3 Minimized Octagons (`moct_t`)

Minimized octagons are an alternative to the matrix representation of octagons. Minimized octagons are very space-efficient; however they can not be manipulated very easily. They are indented manly for storage.

5.3.1 Minimized Representation

Instead of representing all `matsize(`N`)` constraints, the minimized representation only stores constraints $(x_i - x_j \leq c)$ for $c \neq +\infty$. As the matrix storing the constraints is now *hollow*, we lose random-access to elements.

Hollow matrices are stored in a `moct_t` structure. Here is the detailed structure (from the `oct_private.h` file):

```
struct moct_tt {
  var_t            n;     /* number of variables */
  size_t*          bol;   /* begin-of-line indices, array of n*2+1 indices */
  var_t*           col;   /* column indices, array of bol[n*2+1] elements */
  num_t*           data;  /* constraint array of bol[n*2+1] elements */
}
```

Constraints elements are stored in the `data` field. The `bol` (begin-of-line) and `col` (column) fields are used to recover the line and column an element had in the original full matrix representation. Data corresponding to line `i` are contained in `data[bol[i]]` to `data[bol[i+1]-1]`. The column of element `data[k]` is `col[k]`. Thus, if the dimension is `n`, `bol` contains `2n` + 1 elements, `data` and `col` contain `bol[2n+1]` elements.

5.3.2 Minimization Algorithm

In order to be even more efficient, redundant constraints are removed before a matrix is put in hollow matrix form.

For example, if $\mathbf{m}^+_{ij} = \mathbf{m}^+_{ik} + \mathbf{m}^+_{ki}$, then $\mathbf{m}^+_{ij}$ is replaced by $+\infty$. The same holds when $\mathbf{m}^+_{ij} = (\mathbf{m}^+_{i\bar{i}} + \mathbf{m}^+_{\bar{j}j})/2$. However, we have to take care of not removing all $\mathbf{m}^+_{ij}$, $\mathbf{m}^+_{jk}$, $\mathbf{m}^+_{ki}$ when $\mathbf{m}^+_{ij} + \mathbf{m}^+_{jk} + \mathbf{m}^+_{ki} = \mathbf{m}^+_{ik} + \mathbf{m}^+_{kj} + \mathbf{m}^+_{ji} = 0$.

Our minimization algorithm is inspired from the one proposed for DBMs in [LLPY97], but adapted to octagons. It has three passes. First, it calls `oct_close` to get the matrix in closed form. It is mandatory as it reduces the test $\exists \langle i = i_0, \ldots, i_N = j \rangle$, $\mathbf{m}^+_{ij} = \sum_{k=0}^{N-1} \mathbf{m}^+_{i_k i_{k+1}}$ to $\exists k$, $(\mathbf{m}^+)^\bullet_{ij} = (\mathbf{m}^+)^\bullet_{ik} + (\mathbf{m}^+)^\bullet_{kj}$. Then, it slices $\mathcal{X}$ into equivalent classes for the relation $x_i \equiv x_j \iff (\mathbf{m}^+)^\bullet_{ij} + (\mathbf{m}^+)^\bullet_{ji} = 0$. The last phase changes the value of $(\mathbf{m}^+)^\bullet_{ij}$ into $+\infty$ if we have $x_i \not\equiv x_j$, and:

- either $\exists k,\ x_k \not\equiv x_i, x_j,\ (\mathbf{m}^+)^\bullet_{ij} = (\mathbf{m}^+)^\bullet_{ik} + (\mathbf{m}^+)^\bullet_{kj}$,
- or $x_i \not\equiv x_{\bar{i}}$, $x_j \not\equiv x_{\bar{j}}$, and $(\mathbf{m}^+)^\bullet_{ij} = ((\mathbf{m}^+)^\bullet_{i\bar{i}} + (\mathbf{m}^+)^\bullet_{\bar{j}})/2$.

Let $\mathbf{n}^+$ be the resulting matrix; we have the following theorem

Theorem 2. $(\mathbf{n}^+)^\bullet = (\mathbf{m}^+)^\bullet$. ■

Because the minimization algorithm is deterministic and works on the closed form, it is a normal form. Thus, equality can be tested easily.

5.3.3 Algorithms on Minimized Octagons

It is difficult to work on minimized octagons for two reasons:

- the representation is hardly modifiable in-place because it is hollow; making it modifiable in-place would require complex dynamic memory management and hash tables;
- most algorithms need the octagon in closed form in order to perform with maximal accuracy; closure and minimization are not compatible: they are at the opposite ends of the representation spectrum of an octagon.

When analyzing a program in several forward and backward passes, it is handy to store a minimized octagon for each program point. The current analysis pass will need intersect the current result at each program point with the corresponding stored result, and store back this intersection.

5.4 Memory Management of Abstract Elements (oct_sem.c)

The library was designed be memory efficient while hiding memory management details to the user.

In order to avoid unnecessary copies and reduce memory consumption, abstract elements are reference counted:

- `oct_alloc`, `oct_universe`, `oct_empty` return new octagons with `ref`=1;
- `oct_copy(m)` returns `m` with its `ref` incremented: it is a lazy copy;
- `oct_full_copy(m)` returns a new copy of `m` with `ref`=1: it is a full copy;
- `oct_free(m)` decreases `ref` and actually frees the octagon only if `ref`=0.

Semantics functions come in two forms: a *destructive form* and a *non-destructive form*. The non-destructive form preserves its arguments. The destructive form destroys its arguments but is less memory consuming (you do not have to free the arguments after a destructive call). One can think of a destructive call as a non-destructive call followed by `oct_free` on the arguments. In reality, it is a little more efficient as the space allocated for the arguments may be directly used for the result (whereas the non-destructive call often have to perform copies of its arguments, or to allocate fresh octagons). In either case, the returned octagon is a new octagon that must be freed with `oct_free` when no longer used, or used as a destructive argument.

Because of reference counting, things are a little more complex: even within a destructive call, arguments are preserved if they are used by someone else (`ref` > 1): a copy of the argument is made, as in a non-destructive call, and its `ref` is decreased as if the argument where freed. When an operator needs to return exactly one of its argument `m`, it returns `oct_copy(m)` for a non-destructive call, and `m` for a destructive call.

There is one last feature that needs to be discussed here: remembering of the closed form. A program seldom calls `oct_close` directly, but most operators it calls need to compute the strong closure of its arguments (`oct_is_equal`, `oct_is_empty`, `oct_forget`, etc.). In order to avoid duplicate closure computation, the closed form of a matrix `m` is stored in `m->closed` the first time `oct_close` is called, and then returned without re-computation the next time `oct_close` is called on `m`. Remark that we cannot simply replace a matrix by its closed form (unlike minimization in the polyhedra domain) as changing between different matrix representations of the same octagon disrupts fix-point computation using widenings. A closed matrix has its `ref` increased by 1 for each matrix that has a `m->closed` pointer to it. Thus, `oct_close` returns newly created closed matrix with `ref`=2 when called in non-destructive mode (or when the argument has a `ref` > 1); `oct_full_copy(m)` increments `m->closed->ref`; `oct_free(m)` calls `oct_free(m->closed)` whenever `m->ref` reaches 0; and `oct_copy(m)` does not touch to `m->closed->ref`. It is important to remark that a closed matrix cannot be actually freed as long as there exists a matrix the `closed` field of which points to this matrix. Also remark that this caching mechanism can be disabled by calling `oct_close` with the proper argument. By default, caching is only enabled within *tests* functions (`oct_is_empty`, `oct_is_equal`, `oct_is_included_in`), not in operators and transfer functions.

Most of the time, you do not have to take care of reference counting and closure. In practice you simply have to:

- use `oct_copy` and not pointer assignment `oct_t* a = b;`
- call `oct_free` on octagons returned by `oct_alloc`, `oct_empty`, `oct_universe`, `oct_copy`, and the semantics operators when you no longer need them;
- know when to use destructive and non-destructive mode.

5.5 Conversion between Octagons and Polyhedra (oct_polka.c)

5.5.1 From Octagons to Polyhedra (oct_to_poly)

Converting an octagon into a polyhedron (`oct_to_poly`) is easy: one simply has to convert constraints appearing in the octagon into linear constraints. If there is no over-approximation due to different underlying numerical domains, the conversion is exact.

5.5.2 From Polyhedra to Octagons (`oct_from_poly`)

To find the smallest octagon that contains a given polyhedron, we use the *frame* description of the polyhedron, denoted by $\mathbb{P}$. The frame is composed of elements of the following type:

- vertices $\mathbb{P}_{\mathcal{V}}$: $\{ (v_0, \ldots, v_{N-1}) \}$;
- rays $\mathbb{P}_{\mathcal{R}}$: $\{ (\lambda v_0, \ldots, \lambda v_{N-1}) \mid \lambda \geq 0 \}$;
- lines $\mathbb{P}_{\mathcal{L}}$: $\{ (\lambda v_0, \ldots, \lambda v_{N-1}) \mid \lambda \in \mathbb{R} \}$.

In the first pass, we construct $\mathbf{m}^+$ taking only vertices into account:

$$\mathbf{m}^+_{ij} = \max\{ x_j - x_i \mid x_{2k} = v_k,\ x_{2k+1} = -v_k,\ (v_0, \ldots, v_{N-1}) \in \mathbb{P}_{\mathcal{V}} \}.$$

Then $\mathbf{m}^+_{ij}$ is replaced by $+\infty$ if:

- either $x_j - x_i > 0$, for $x_{2k} = v_k,\ x_{2k+1} = -v_k,\ (v_0, \ldots, v_{N-1}) \in \mathbb{P}_{\mathcal{R}}$;
- either $x_j - x_i \neq 0$, for $x_{2k} = v_k,\ x_{2k+1} = -v_k,\ (v_0, \ldots, v_{N-1}) \in \mathbb{P}_{\mathcal{L}}$.

Remark that this conversion is also sound even when strict constraints are allowed in the polyhedron: $(\alpha_0 v_0 + \cdots + \alpha_{N-1} v_{N-1} < c)$ is simply relaxed to $(\alpha_0 v_0 + \cdots + \alpha_{N-1} v_{N-1} \leq c)$.

5.6 Operators and Transfer Functions (`oct_sem.c`)

Most operators and transfer functions are direct implementations of those described in [Min06], but adapted to our more compact representation.

Bibliography

[APR] APRON. Analyse de PROgrammes Numériques (Numerical Program Analysis). http://www.cri.ensmp.fr/apron/.

[Ast] Astrée. Analyse Statique de logiciels Temps-Réel embarqués (static analysis of critical real-time embedded software) RNTL project page. http://www.di.ens.fr/~cousot/projets/ASTREE/.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN PLDI'03*, volume 548030, pages 196–207. ACM Press, June 2003. http://www.di.ens.fr/~mine/publi/pldi045-blanchet.pdf.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977. http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml.

[GNU] GNU. Gnu multiple precision library (version 4.1 at least). http://www.swox.com/gmp/.

[Jea] B. Jeannet. New polka: A library to handle convex polyhedra in any dimension (version 2.0.0 at least). http://www.irisa.fr/prive/bjeannet/newpolka.html.

[La] X. Leroy and al. The objective caml system (version 3.0 at least). http://pauillac.inria.fr/ocaml/.

[LLPY97] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *IEEE RTSS'97*, pages 14–24. IEEE CS Press, December 1997. http://www.docs.uu.se/docs/rtmv/papers/llpw-rtss97.ps.gz.

[Min01a] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001. http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf.

[Min01b] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf.

[Min04] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, dec 2004. http://www.di.ens.fr/~mine/these/.

[Min06] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. (to appear) http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf.

[Mon] D. Monniaux. Caml-gmp, an extended precision computation library. http://caml.inria.fr/hump.html.

[Pra77] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology. Cambridge., September 1977.

[TPpL] Inria Lorraine "The PolKA project and LORIA". The multiple precision floating-point reliable library. http://www.loria.fr/projets/mpfr/.

Index

•, 44
$\mathbb{I}$, 11
$\bar{\mathbb{I}}$, 11
N, 14, 42
$\mathbb{P}_{\mathcal{L}}$, 47
$\mathbb{P}_{\mathcal{R}}$, 47
$\mathbb{P}_{\mathcal{V}}$, 47
$\mathcal{V}$, 42
$\mathcal{X}$, 42
$\bar{\imath}$, 42
$\mathbf{m}^+$, 42
v_i, 42
x_i, 42

bool, 10

chrono_t, 26
closure, 15, 23, 43
closure remembering, 15, 46

destructive, 15, 46
dimsup_t, 19

ENABLE_ASSERT, 24
ENABLE_MALLOC_MONITORING, 25, 36
ENABLE_TIMING, 26, 37

false, 10

incremental,closure, 44
Int_val, 37

minimization, 22, 34, 45
mmalloc_t, 25
moct_t, 21, 45
moct_tt, 45
Moct_val, 37

new_n, 25
new_t, 25
num_add, 13
num_clear, 11
num_clear_n, 11
num_cmp, 12
num_cmp_int, 12
num_cmp_zero, 12
num_div_by_2, 13
num_fits_float, 12
num_fits_frac, 12
num_fits_int, 12
num_get_den, 12
num_get_float, 12
num_get_int, 12
num_get_mpfr, 12
num_get_mpq, 12
num_get_mpz, 12
num_get_num, 12
num_infty, 12
num_init, 11
num_init_n, 11
num_init_set, 12
num_init_set_float, 12
num_init_set_frac, 12
num_init_set_infty, 12
num_init_set_int, 12
num_init_set_n, 12
num_max, 13
num_min, 13
num_mul, 13
num_mul_by_2, 13
num_neg, 13
num_print, 13
num_set, 11
num_set_float, 11
num_set_frac, 11
num_set_infty, 11
num_set_int, 11
num_set_mpfr, 12
num_set_mpq, 12
num_set_mpz, 12
num_set_n, 11
num_snprintf, 13
num_sub, 13
num_t, 11
Num_val, 37

Oct, 28
Oct.add_bin_constraints, 32
Oct.add_constraint, 32
Oct.add_dims_and_embed, 33
Oct.add_dims_and_embed_multi, 33
Oct.add_dims_and_project, 33
Oct.add_dims_and_project_multi, 33
Oct.add_epsilon, 35
Oct.add_epsilon_bin, 35
Oct.add_epsilon_max, 35
Oct.add_permute_dims_and_embed, 34
Oct.add_permute_dims_and_project, 34
Oct.assign_var, 32
Oct.Bottom, 30
Oct.del_dims, 33
Oct.del_dims_multi, 33
Oct.dim, 30

Oct.dimsup, 33
Oct.empty, 30
Oct.False, 30
Oct.float_of_num, 29
Oct.float_of_vnum, 29
Oct.fmoctprinter, 36
Oct.fnumprinter, 35
Oct.foctdiffprinter, 36
Oct.foctnewprinter, 36
Oct.foctprinter, 36
Oct.forget, 32
Oct.frac_of_num, 29
Oct.frac_of_vnum, 29
Oct.from_box, 34
Oct.from_poly, 36
Oct.fvnumprinter, 35
Oct.get_bounds, 35
Oct.get_box, 34
Oct.init, 28
Oct.int_of_num, 29
Oct.int_of_vnum, 29
Oct.inter, 31
Oct.interv_add_constraint, 33
Oct.interv_assign_var, 32
Oct.interv_substitute_var, 32
Oct.is_empty, 31
Oct.is_empty_lazy, 31
Oct.is_equal, 31
Oct.is_equal_lazy, 31
Oct.is_in, 31
Oct.is_included_in, 31
Oct.is_included_in_lazy, 31
Oct.is_universe, 31
Oct.m_dim, 34
Oct.m_from_oct, 34
Oct.m_is_empty, 34
Oct.m_is_equal, 34
Oct.m_to_oct, 34
Oct.memprint, 36
Oct.moct, 34
Oct.moctprinter, 36
Oct.mpfr_of_num, 29
Oct.mpfr_of_vnum, 29
Oct.mpq_of_num, 29
Oct.mpq_of_vnum, 29
Oct.mpz_of_num, 29
Oct.mpz_of_vnum, 29
Oct.narrowing, 32
Oct.nbconstraints, 30
Oct.num, 28
Oct.num_infty, 29
Oct.num_of_float, 28
Oct.num_of_frac, 28
Oct.num_of_int, 28
Oct.num_of_mpfr, 29
Oct.num_of_mpq, 29
Oct.num_of_mpz, 29
Oct.numprinter, 35
Oct.oct, 30
Oct.octprinter, 35
Oct.permute_del_dims, 34
Oct.PreWiden, 31
Oct.set_bounds, 35
Oct.substitute_var, 32
Oct.tbool, 30
Oct.time_flow, 33
Oct.timeprint, 37
Oct.to_poly, 36
Oct.Top, 30
Oct.True, 30
Oct.union, 31
Oct.universe, 30
Oct.vnum, 28
Oct.vnum_of_float, 29
Oct.vnum_of_float_opt, 29
Oct.vnum_of_frac, 29
Oct.vnum_of_frac_opt, 29
Oct.vnum_of_int, 29
Oct.vnum_of_int_opt, 29
Oct.vnum_of_mpfr, 29
Oct.vnum_of_mpfr_opt, 29
Oct.vnum_of_mpq, 29
Oct.vnum_of_mpq_opt, 29
Oct.vnum_of_mpz, 29
Oct.vnum_of_mpz_opt, 29
Oct.vnumprinter, 35
Oct.WidenFast, 31
Oct.widening, 31
Oct.WidenSteps, 31
Oct.widen, 31
Oct.WidenUnit, 31
Oct.WidenZero, 31
oct_add_bin_constraints, 17
oct_add_constraint, 18
oct_add_dimensions_and_embed, 18
oct_add_dimensions_and_embed_multi, 19
oct_add_dimensions_and_project, 19
oct_add_dimensions_and_project_multi, 19
oct_add_epsilon, 21
oct_add_epsilon_bin, 21
oct_add_epsilon_max, 21
oct_add_permute_dimensions_and_embed, 20
oct_add_permute_dimensions_and_project, 20
oct_alloc, 23, 46
oct_assign_variable, 17
oct_check_closed, 23
oct_chrono_get, 26
oct_chrono_print, 26
oct_chrono_reset, 26
oct_chrono_start, 26
oct_chrono_stop, 26
oct_close, 23, 46
oct_close_incremental, 23
oct_close_lazy, 23, 46
oct_cons, 17
oct_convex_convex_hull, 16
oct_copy, 14, 46
oct_dimension, 14
OCT_DOMAIN, 13
oct_domain_string, 13

oct_elem, 22
oct_empty, 14, 46
oct_forget, 17
oct_free, 14, 46
oct_from_box, 21
oct_from_poly, 24, 47
oct_full_copy, 23, 46
oct_get_bounds, 20
oct_get_box, 20
OCT_HAS_BOOL, 10
OCT_HAS_NEW_POLKA, 23
OCT_IMPLEMENTATION_STRING, 13
oct_init, 10
oct_intersection, 15
oct_interv_add_constraint, 18
oct_interv_assign_variable, 17
oct_interv_substitute_variable, 18
oct_is_closed, 23
oct_is_empty, 15
oct_is_empty_lazy, 15
oct_is_equal, 15
oct_is_equal_lazy, 15
oct_is_in, 15
oct_is_included_in, 15
oct_is_included_in_lazy, 15
oct_is_universe, 15
oct_m_dimension, 22
oct_m_free, 22
oct_m_from_oct, 22, 45
oct_m_is_empty, 22
oct_m_is_equal, 22
oct_m_print, 22
oct_m_to_oct, 22
oct_mm_free, 25
oct_mm_malloc, 24
oct_mm_realloc, 24
oct_mmalloc_get_current, 25
oct_mmalloc_new, 25
oct_mmalloc_print, 25, 37
oct_mmalloc_reset, 25
oct_mmalloc_use, 25
oct_narrowing, 16
oct_nbconstraints, 14
OCT_NUM_CLOSED, 13
OCT_NUM_EXACT, 13
oct_permute_remove_dimensions, 20
OCT_PRE_WIDENING, 16
oct_print, 21
oct_remove_dimensions, 19
oct_remove_dimensions_multi, 19
oct_set_bounds, 20
oct_substitute_variable, 17
oct_t, 14, 42
oct_time_flow, 18
oct_timing_clear, 27
oct_timing_enter, 26
oct_timing_exit, 27
oct_timing_print, 27
oct_timing_print_all, 27, 37
oct_timing_reset, 27
oct_timing_reset_all, 27
oct_to_poly, 24, 46
oct_tt, 43
oct_universe, 14, 46
Oct_val, 37
oct_widening, 16
OCT_WIDENING_FAST, 16
oct_widening_steps, 16
oct_widening_type, 16
OCT_WIDENING_UNIT, 16
OCT_WIDENING_ZERO, 16

precision, 13, 15–17, 24
pretty-printers, 35

reference counting, 14, 37, 46
renew_n, 25

soundness, 11, 24

tbool, 10
tbool_bottom, 10
tbool_false, 10
tbool_top, 10
tbool_true, 10
true, 10

Val_int, 37
Val_moct, 37
Val_oct, 37
var_t, 14
vnum_t, 37
Vnum_val, 37