MLApronIDL: OCaml interface for APRON library

Bertrand Jeannet

June 9, 2008

Contents

Chapter 1

Introduction

This package is an OCaml interface for the APRON library/interface. The interface is accessed via the module Apron, which is decomposed into 15 submodules, corresponding to C modules:

Scalar	:	scalars (numbers)
Interval	:	intervals on scalars
Coeff	:	coefficients (either scalars or intervals)
Dimension	:	dimensions and related operations
Linexpr0	:	(interval) linear expressions, level 0
Lincons0	:	(interval) linear constraints, level 0
Generator0	:	generators, level 0
texpr0	:	tree expressions, level 0
tcons0	:	tree constraints, level 0
Manager	:	managers
Abstract0	:	abstract values, level 0
Var	:	variables
Environment	:	environment binding variables to dimensions
Linexpr1	:	(interval) linear expressions, level 1
Lincons1	:	interval) linear constraints, level 1
Generator1	:	generators, level 1
texpr1	:	tree expressions, level 1
tcons1	:	tree constraints, level 1
Abstract1	:	abstract values, level 1
Parser	:	strings parsing

The package also includes the MLGmpIDL wrapper to GMP and MPFR libraries.

Requirements

- APRON library
- GMP library version 4.2 or up (tested with version 4.2.1)
- MPFR library version 2.2 or up (tested with version 2.2.1)
- OCaml 3.0 or up (tested with 3.09)
- Camlidl (tested with 1.05)

Optionally,

- GNU M4 preprocessor
- GNU sed

Installation

Library Set the file ../Makefile.config to your own setting.

type 'make', and then 'make install'

The OCaml part of the library is named apron.cma (.cmxa, .a) The C part of the library, which is automatically referenced by apron.cma/apron.cmxa, is named libapron_caml.a (libapron_caml_debug.a)

'make install' installs not only .mli, .cmi, but also .idl files.

Documentation The documentation (currently very sketchy) is generated with ocamldoc.

'make mlapronidl.pdf'

'make html' (put the HTML files in the html subdirectoy)

Miscellaneous 'make clean' and 'make distclean' have the usual behaviour.

Compilation and Linking

To make things clearer, we assume an example file `example.ml` which uses both NewPolka (convex polyhedra) and Box (intervals) libraries, in their versions where rationals are GMP rationals (which is the default). We assume that C and OCaml interface and library files are located in directory $APRON/lib. The native-code compilation command looks like

```
ocamlopt -I $APRON/lib -o example.opt bigarray.cmxa gmp.cmxa apron.cmxa box.cmxa
polka.cmxa
```

Comments:

1. You need at least the libraries `bigarray` (standard OCaml distribution), `gmp`, and `apron` (standard APRON distribution), plus the one implementing an effective abstract domains: here, `box`, and `polka`.
2. The C libraries associated to those OCaml libraries (*e.g.,* `gmp_caml`, `box_caml`, ...) are automatically looked for, as well as the the libraries implementing abstract domains (*e.g.,* `polka`, `box` *in their default version* (which corresponds to the MPQ suffix).

   If other versions of abstract domains library are wanted, you should use the `-noautolink` option as explained below.

The byte-code compilation process looks like

```
ocamlc -I $APRON/lib -make-runtime -o myrun bigarray.cma gmp.cma apron.cma box.cma
polka.cma

ocamlc -I $APRON/lib -use-runtime myrun -o exampleg bigarray.cma gmp.cma apron.cma
box.cma polka.cma example.ml
```

Comments:

1. One first build a custom bytecode interpreter that includes the new native-code needed;
2. One then compile the `example.ml` file.

The automatic search for C libraries associated to these OCaml libraries can be disabled by the option `-noautolink` supported by both `ocamlc` and `ocamlopt` commands. For instance, the command for native-code compilation can alternatively looks like:

```
ocamlopt -I $APRON/lib -noautolink -o example.opt bigarray.cmxa gmp.cmxa apron.cmxa
box.cmxa polka.cmxa -cclib "-lpolka_caml lpolka -lbox_caml -lbox -lapron_caml
-lapron -lgmp_caml -LMPFR -lmpfr -LGMP/lib -lgmp"
```

This is mandatory if you want to use non-default versions of abstract domains library. For insta,ce; if you want to use POLKA in its `Ill` version (`long long int`) and BOX in its `D` version (`double`), the command looks like:

```
ocamlopt -I $APRON/lib -noautolink -o example.opt bigarray.cmxa gmp.cmxa apron.cmxa
box.cmxa polka.cmxa -cclib "-lpolka_caml lpolkaIll -lbox_caml -lboxD -lapron_caml
-lapron -lgmp_caml -LMPFR -lmpfr -LGMP/lib -lgmp"
```

The option `-verbose` helps to understand what is happening in case of problem.

More details are given in the modules implementing a specific abstract domain.

Part I

Coefficients

Chapter 2

Module `Scalar` : APRON Scalar numbers.

See `Mpqf`[33] for operations on GMP multiprecision rational numbers and `Mpfr`[30] for operations on MPFR multi-precision floating-point numbers.

```
type t =
  | Float of float
  | Mpqf of Mpqf.t
  | Mpfrf of Mpfrf.t
```

APRON Scalar numbers. See `Mpqf`[33] for operations on GMP multiprecision rational numbers and `Mpfr`[30] for operations on MPFR multi-precision floating-point numbers.

```
val of_mpq : Mpq.t -> t
val of_mpqf : Mpqf.t -> t
val of_int : int -> t
val of_frac : int -> int -> t
```

Create a scalar of type `Mpqf` from resp.

- A multi-precision rational `Mpq.t`
- A multi-precision rational `Mpqf.t`
- an integer
- a fraction `x/y`

```
val of_mpfr : Mpfr.t -> t
val of_mpfrf : Mpfrf.t -> t
```

Create a scalar of type `Mpfrf` with the given value

```
val of_float : float -> t
```

Create a scalar of type `Float` with the given value

```
val of_infty : int -> t
```

Create a scalar of type `Float` with the value multiplied by infinity (resulting in minus infinity, zero, or infinity

```
val is_infty : t -> int
```

Infinity test. `is_infty x` returns `-1` if x is `-oo`, `1` if x is `+oo`, and `0` if `x` is finite.

```
val sgn : t -> int
```

Return the sign of the coefficient, which may be a negative value, zero or a positive value.

`val cmp : t -> t -> int`

Compare two coefficients, possibly converting to `Mpqf.t`. `compare x y` returns a negative number if `x` is less than `y`, `0` if they ar equal, and a positive number if `x` is greater than `y`.

`val cmp_int : t -> int -> int`

Compare a coefficient with an integer

`val equal : t -> t -> bool`

Equality test, possibly using a conversion to `Mpqf.t`. Return `true` if the 2 values are equal. Two infinite values of the same signs are considered as equal.

`val equal_int : t -> int -> bool`

Equality test with an integer

`val neg : t -> t`

Negation

`val to_string : t -> string`

Conversion to string, using `string_of_double`, `Mpqf.to_string` or `Mpfr.to_string`

`val print : Format.formatter -> t -> unit`

Print a coefficient

Chapter 3

Module `Interval` : APRON Intervals on scalars

```
type t = {
  mutable inf : Scalar.t ;
  mutable sup : Scalar.t ;
}
```

APRON Intervals on scalars

`val of_scalar : Scalar.t -> Scalar.t -> t`

Build an interval from a lower and an upper bound

`val of_infsup : Scalar.t -> Scalar.t -> t`

depreciated

`val of_mpq : Mpq.t -> Mpq.t -> t`

`val of_mpqf : Mpqf.t -> Mpqf.t -> t`

`val of_int : int -> int -> t`

`val of_frac : int -> int -> int -> int -> t`

`val of_float : float -> float -> t`

`val of_mpfr : Mpfr.t -> Mpfr.t -> t`

Create an interval from resp. two

- multi-precision rationals `Mpq.t`
- multi-precision rationals `Mpqf.t`
- integers
- fractions `x/y` and `z/w`
- machine floats
- Mpfr floats

`val is_top : t -> bool`

Does the interval represent the universe (`[-oo,+oo]`) ?

`val is_bottom : t -> bool`

Does the interval contain no value (`[a,b]` with a>b) ?

`val is_leq : t -> t -> bool`

Inclusion test. `is_leq x y` returns `true` if `x` is included in `y`

```
val cmp : t -> t -> int
```

Non Total Comparison: 0: equality -1: i1 included in i2 +1: i2 included in i1 -2: i1.inf less than or equal to i2.inf +2: i1.inf greater than i2.inf

```
val equal : t -> t -> bool
```

Equality test

```
val is_zero : t -> bool
```

Is the coefficient equal to scalar 0 or interval `0,0` ?

```
val neg : t -> t
```

Negation

```
val top : t
val bottom : t
```

Top and bottom intervals (using `DOUBLE` coefficients)

```
val set_infsup : t -> Scalar.t -> Scalar.t -> unit
```

Fill the interval with the given lower and upper bouunds

```
val set_top : t -> unit
val set_bottom : t -> unit
```

Fill the interval with top (resp. bottom) value

```
val print : Format.formatter -> t -> unit
```

Print an interval, under the format `[inf,sup]`

Chapter 4

Module `Coeff` : APRON Coefficients (either scalars or intervals)

```
type union_5 =
  | Scalar of Scalar.t
  | Interval of Interval.t
```

```
type t = union_5
```

APRON Coefficients (either scalars or intervals)

```
val s_of_mpq : Mpq.t -> t
val s_of_mpqf : Mpqf.t -> t
val s_of_int : int -> t
val s_of_frac : int -> int -> t
```

Create a scalar coefficient of type `Mpqf.t` from resp.

- A multi-precision rational `Mpq.t`
- A multi-precision rational `Mpqf.t`
- an integer
- a fraction `x/y`

```
val s_of_float : float -> t
```

Create an interval coefficient of type `Float` with the given value

```
val s_of_mpfr : Mpfr.t -> t
```

Create an interval coefficient of type `Mpfr` with the given value

```
val i_of_scalar : Scalar.t -> Scalar.t -> t
```

Build an interval from a lower and an upper bound

```
val i_of_mpq : Mpq.t -> Mpq.t -> t
val i_of_mpqf : Mpqf.t -> Mpqf.t -> t
val i_of_int : int -> int -> t
val i_of_frac : int -> int -> int -> int -> t
val i_of_float : float -> float -> t
val i_of_mpfr : Mpfr.t -> Mpfr.t -> t
```

Create an interval coefficient from resp. two

- multi-precision rationals `Mpq.t`
- multi-precision rationals `Mpqf.t`
- integers
- fractions `x/y` and `z/w`
- machine floats
- Mpfr floats

`val is_scalar : t -> bool`

`val is_interval : t -> bool`

`val cmp : t -> t -> int`

Non Total Comparison:

- If the 2 coefficients are both scalars, corresp. to Scalar.cmp
- If the 2 coefficients are both intervals, corresp. to Interval.cmp
- otherwise, -3 if the first is a scalar, 3 otherwise

`val equal : t -> t -> bool`

Equality test

`val is_zero : t -> bool`

Is the coefficient equal to scalar 0 or interval `0,0` ?

`val neg : t -> t`

Negation

`val reduce : t -> t`

Convert interval to scalar if possible

`val print : Format.formatter -> t -> unit`

Printing

Part II

Managers and Abstract Domains

Chapter 5

Module `Manager` : APRON Managers

```
type funid =
  | Funid_unknown
  | Funid_copy
  | Funid_free
  | Funid_asize
  | Funid_minimize
  | Funid_canonicalize
  | Funid_hash
  | Funid_approximate
  | Funid_fprint
  | Funid_fprintdiff
  | Funid_fdump
  | Funid_serialize_raw
  | Funid_deserialize_raw
  | Funid_bottom
  | Funid_top
  | Funid_of_box
  | Funid_dimension
  | Funid_is_bottom
  | Funid_is_top
  | Funid_is_leq
  | Funid_is_eq
  | Funid_is_dimension_unconstrained
  | Funid_sat_interval
  | Funid_sat_lincons
  | Funid_sat_tcons
  | Funid_bound_dimension
  | Funid_bound_linexpr
  | Funid_bound_texpr
  | Funid_to_box
  | Funid_to_lincons_array
  | Funid_to_tcons_array
  | Funid_to_generator_array
  | Funid_meet
  | Funid_meet_array
  | Funid_meet_lincons_array
  | Funid_meet_tcons_array
  | Funid_join
  | Funid_join_array
```

```
  | Funid_add_ray_array
  | Funid_assign_linexpr_array
  | Funid_substitute_linexpr_array
  | Funid_assign_texpr_array
  | Funid_substitute_texpr_array
  | Funid_add_dimensions
  | Funid_remove_dimensions
  | Funid_permute_dimensions
  | Funid_forget_array
  | Funid_expand
  | Funid_fold
  | Funid_widening
  | Funid_closure
  | Funid_change_environment
  | Funid_rename_array
```

```
type funopt = {
  algorithm : int ;
  timeout : int ;
  max_object_size : int ;
  flag_exact_wanted : bool ;
  flag_best_wanted : bool ;
}
```

```
type exc =
  | Exc_none
  | Exc_timeout
  | Exc_out_of_space
  | Exc_overflow
  | Exc_invalid_argument
  | Exc_not_implemented
```

```
type exclog = {
  exn : exc ;
  funid : funid ;
  msg : string ;
}
```

```
type 'a t
```

APRON Managers

The type parameter `'a` allows to distinguish managers allocated by different underlying abstract domains. Concerning the other types,

- `funid` defines identifiers for the generic function working on abstrat values;
- `funopt` defines the options associated to generic functions;
- `exc` defines the different kind of exceptions;
- `exclog` defines the exceptions raised by APRON functions.

```
val get_library : 'a t -> string
```

Get the name of the effective library which allocated the manager

```
val get_version : 'a t -> string
```

Get the version of the effective library which allocated the manager

```
val funopt_make : unit -> funopt
```

Return the default options for any function (`0` or `false` for al fields)

`val get_funopt : 'a t -> funid -> funopt`

Get the options sets for the function. The result is a copy of the internal structure and may be freely modified. `funid` should be different from `Funid_change_environment` and `Funid_rename_array` (no option associated to them).

`val set_funopt : 'a t -> funid -> funopt -> unit`

Set the options for the function. `funid` should be different from `Funid_change_environment` and `Funid_rename_array` (no option associated to them).

`val get_flag_exact : 'a t -> bool`

Get the corresponding result flag

`val get_flag_best : 'a t -> bool`

Get the corresponding result flag

`exception Error of exclog`

Exception raised by functions of the interface

```
val string_of_funid : funid -> string
val string_of_exc : exc -> string
val print_funid : Format.formatter -> funid -> unit
val print_funopt : Format.formatter -> funopt -> unit
val print_exc : Format.formatter -> exc -> unit
val print_exclog : Format.formatter -> exclog -> unit
```

Printing functions

`val set_deserialize : 'a t -> unit`

Set / get the global manager used for deserialization

`val get_deserialize : unit -> 'a t`

Chapter 6

Module `Box` : Intervals abstract domain

`type t`

Type of boxes.

Boxes constrains each dimension/variable `x_i` to belong to an interval `I_i`.

Abstract values which are boxes have the type `t Apron.AbstractX.t`.

Managers allocated for boxes have the type `t Apron.manager.t`.

`val manager_alloc : unit -> t Apron.Manager.t`

Create a Box manager.

6.1 Compilation information

6.1.1 Bytecode compilation

To compile to bytecode, you should first generate a custom interpreter with a command which should look like:

```
ocamlc -I $APRON_PREFIX/lib -make-runtime -o myrun bigarray.cma gmp.cma apron.cma box.cma
```

and then you compile and link your example `X.ml` with

`ocamlc -I $APRON_PREFIX/lib -c X.ml` and

```
ocamlc -I $APRON_PREFIX/lib -use-runtime myrun -o X bigarray.cma gmp.cma apron.cma box.cma
X.cmo
```

Comments: The C libraries related to `gmp.cma` and `apron.cma` are automatically looked for (thanks to the auto-linking feature provided by `ocamlc`). For `box.cma`, the library `libbox.a`, identic to `libboxMPQ.a`, is selected by default. The `-noautolink` option should be used to select a differetn version. See the C documentation of `Box` library for details.

With the `-noautolink` option, the generation of the custom runtime executable should be done with

```
ocamlc -I $APRON_PREFIX/lib -noautolink -make-runtime -o myrun bigarray.cma gmp.cma apron.cma
box.cma -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lbox_caml -lboxMPQ -lapron_caml -lapron
-lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl"
```

6.1.2 Native-code compilation

You compile and link with

`ocamlopt -I $APRON_PREFIX/lib -c X.ml` and

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa box.cmxa X.cmx`

Comments: Same as for bytecode compilation. With the `-noautolink` option, the linking command becomes

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa box.cmxa -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lbox_caml -lboxMPQ -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl" X.cmx`

Chapter 7

Module Oct : Octagon abstract domain.

```
type internal
```

Octagon abstract domain.

```
type t
```

Type of octagons.

Octagons are defined by conjunctions of inequalities of the form `+/-x_i +/- x_j >= 0`.

Abstract values which are octagons have the type `t Apron.AbstractX.t`.

Managers allocated for octagons have the type `t Apron.manager.t`.

```
val manager_alloc : unit -> t Apron.Manager.t
```

Allocate a new manager to manipulate octagons.

```
val manager_get_internal : t Apron.Manager.t -> internal
```

No internal parameters for now...

```
val of_generator_array :
  t Apron.Manager.t ->
  int -> int -> Apron.Generator0.t array -> t Apron.Abstract0.t
```

Approximate a set of generators to an abstract value, with best precision.

```
val widening_thresholds :
  t Apron.Manager.t ->
  t Apron.Abstract0.t ->
  t Apron.Abstract0.t -> Apron.Scalar.t array -> t Apron.Abstract0.t
```

Widening with scalar thresholds.

```
val narrowing :
  t Apron.Manager.t ->
  t Apron.Abstract0.t -> t Apron.Abstract0.t -> t Apron.Abstract0.t
```

Standard narrowing.

```
val add_epsilon :
  t Apron.Manager.t ->
  t Apron.Abstract0.t -> Apron.Scalar.t -> t Apron.Abstract0.t
```

Perturbation.

```
val add_epsilon_bin :
  t Apron.Manager.t ->
  t Apron.Abstract0.t ->
  t Apron.Abstract0.t -> Apron.Scalar.t -> t Apron.Abstract0.t
```

Perturbation.

```
val pre_widening : int
```

Algorithms.

7.1 Compilation information

7.1.1 Bytecode compilation

To compile to bytecode, you should first generate a custom interpreter with a command which should look like:

`ocamlc -I $APRON_PREFIX/lib -make-runtime -o myrun bigarray.cma gmp.cma apron.cma oct.cma`

and then you compile and link your example `X.ml` with

`ocamlc -I $APRON_PREFIX/lib -c X.ml` and

`ocamlc -I $APRON_PREFIX/lib -use-runtime myrun -o X bigarray.cma gmp.cma apron.cma oct.cma X.cmo`

Comments: The C libraries related to `gmp.cma` and `apron.cma` are automatically looked for (thanks to the auto-linking feature provided by `ocamlc`). For `oct.cma`, the library `liboct.a`, identic to `liboctMPQ.a`, is selected by default. The `-noautolink` option should be used to select a differetn version. See the C documentation of `Oct` library for details.

With the `-noautolink` option, the generation of the custom runtime executable should be done with

`ocamlc -I $APRON_PREFIX/lib -noautolink -make-runtime -o myrun bigarray.cma gmp.cma apron.cma oct.cma -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-loct_caml -loctMPQ -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl"`

7.1.2 Native-code compilation

You compile and link with

`ocamlopt -I $APRON_PREFIX/lib -c X.ml` and

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa oct.cmxa X.cmx`

Comments: Same as for bytecode compilation. With the `-noautolink` option, the linking command becomes

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa oct.cmxa -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-loct_caml -loctMPQ -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl" X.cmx`

Chapter 8

Module `Polka` : Convex Polyhedra and Linear Equalities abstract domains

`type internal`

Convex Polyhedra and Linear Equalities abstract domains

`type loose`

`type strict`

Two flavors for convex polyhedra: loose or strict.

Loose polyhedra cannot have strict inequality constraints like `x>0`. They are algorithmically more efficient (less generators, simpler normalization).

Convex polyhedra are defined by the conjunction of a set of linear constraints of the form `a_0*x_0 + ... + a_n*x_n + b >= 0` or `a_0*x_0 + ... + a_n*x_n + b > 0` where `a_0, ..., a_n, b, c` are constants and `x_0, ..., x_n` variables.

`type equalities`

Linear equalities.

Linear equalities are conjunctions of linear equalities of the form `a_0*x_0 + ... + a_n*x_n + b = 0`.

`type 'a t`

Type of convex polyhedra/linear equalities, where `'a` is `loose`, `strict` or `equalities`.

Abstract values which are convex polyhedra have the type `(loose t) Apron.Abstract0.t` or `(loose t) Apron.Abstract1.t` or `(strict t) Apron.Abstract0.t` or `(strict t) Apron.Abstract1.t`.

Abstract values which are conjunction of linear equalities have the type `(equalities t) Apron.Abstract0.t` or `(equalities t) Apron.Abstract1.t`.

Managers allocated by NewPolka have the type `'a t Apron.Manager.t`.

`val manager_alloc_loose : unit -> loose t Apron.Manager.t`

Create a NewPolka manager for loose convex polyhedra.

`val manager_alloc_strict : unit -> strict t Apron.Manager.t`

Create a NewPolka manager for strict convex polyhedra.

`val manager_alloc_equalities : unit -> equalities t Apron.Manager.t`

Create a NewPolka manager for conjunctions of linear equalities.

`val manager_get_internal : 'a t Apron.Manager.t -> internal`

Get the internal submanager of a NewPolka manager.

Various options. See the C documentation

```
val set_max_coeff_size : internal -> int -> unit
val set_approximate_max_coeff_size : internal -> int -> unit
val get_max_coeff_size : internal -> int
val get_approximate_max_coeff_size : internal -> int
```

8.1 Compilation information

8.1.1 Bytecode compilation

To compile to bytecode, you should first generate a custom interpreter with a command which should look like:

`ocamlc -I $APRON_PREFIX/lib -make-runtime -o myrun bigarray.cma gmp.cma apron.cma polka.cma -cclib "-lpolkag"`

and then you compile and link your example `X.ml` with

`ocamlc -I $APRON_PREFIX/lib -c X.ml` and

`ocamlc -I $APRON_PREFIX/lib -use-runtime myrun -o X bigarray.cma gmp.cma apron.cma polka.cma X.cmo`

Comments: The C libraries related to `gmp.cma` and `apron.cma` are automatically looked for (thanks to the auto-linking feature provided by `ocamlc`). For `polka.cma`, the library `libpolka.a`, identic to `libpolkaMPQ.a`, is selected by default. The `-noautolink` option should be used to select a differetn version. See the C documentation of `Polka` library for details.

With the `-noautolink` option, the generation of the custom runtime executable should be done with

`ocamlc -I $APRON_PREFIX/lib -noautolink -make-runtime -o myrun bigarray.cma gmp.cma apron.cma polka.cma -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lpolka_caml -lpolka -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl"`

8.1.2 Native-code compilation

You compile and link with

`ocamlopt -I $APRON_PREFIX/lib -c X.ml` and

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa polka.cmxa -cclib "-lpolkag" X.cmx`

Comments: Same as for bytecode compilation. With the `-noautolink` option, the linking command becomes

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa polka.cmxa -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lpolka_caml -lpolkaMPQ -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl" X.cmx`

Chapter 9

Module `Ppl` : Convex Polyhedra and Linear Congruences abstract domains (PPL wrapper)

This module is a wrapper around the Parma Polyhedra Library.

`type loose`

`type strict`

Two flavors for convex polyhedra: loose or strict.

Loose polyhedra cannot have strict inequality constraints like `x>0`. They are algorithmically more efficient (less generators, simpler normalization).

Convex polyhedra are defined by the conjunction of a set of linear constraints of the form `a_0*x_0 + ... + a_n*x_n + b >= 0` or `a_0*x_0 + ... + a_n*x_n + b > 0` where `a_0, ..., a_n, b, c` are constants and `x_0, ..., x_n` variables.

`type grid`

Linear congruences.

Linear congruences are defined by the conjunction of equality constraints modulo a rational number, of the form `a_0*x_0 + ... + a_n*x_n = b mod c`, where `a_0, ..., a_n, b, c` are constants and `x_0, ..., x_n` variables.

`type 'a t`

Type of convex polyhedra/linear congruences, where `'a` is `loose`, `strict` or `grid`.

Abstract values which are convex polyhedra have the type `loose t Apron.AbstractX.t` or `strict t Apron.AbstractX.t`.

Abstract values which are conjunction of linear congruences equalities have the type `grid t Apron.AbstractX.t`.

Managers allocated by PPL have the type `'a t Apron.Manager.t`.

`val manager_alloc_loose : unit -> loose t Apron.Manager.t`

Allocate a PPL manager for loose convex polyhedra.

`val manager_alloc_strict : unit -> strict t Apron.Manager.t`

Allocate a PPL manager for strict convex polyhedra.

`val manager_alloc_grid : unit -> grid t Apron.Manager.t`

Allocate a new manager for linear congruences (grids)

9.1 Compilation information

9.1.1 Bytecode compilation

To compile to bytecode, you should first generate a custom interpreter with a command which should look like:

`ocamlc -I $APRON_PREFIX/lib -make-runtime -o myrun -cc "g++" bigarray.cma gmp.cma apron.cma ppl.cma`

and then you compile and link your example `X.ml` with

`ocamlc -I $APRON_PREFIX/lib -c X.ml` and

`ocamlc -I $APRON_PREFIX/lib -use-runtime myrun -o X bigarray.cma gmp.cma apron.cma ppl.cma X.cmo`

Comments:

Do not forget the `-cc "g++"` option: PPL is a C++ library which requires a C++ linker.

The C libraries related to `gmp.cma`, `apron.cma` and `ppl.cma` are automatically looked for (thanks to the auto-linking feature provided by `ocamlc`). Be aware that PPL requires the C++ wrapper `libgmpxx.a` library on top of GMP library, which should thus be installed. With the `-noautolink` option, the generation of the custom runtime executable should be done with

`ocamlc -I $APRON_PREFIX/lib -noautolink -make-runtime -o myrun -cc "g++"  bigarray.cma gmp.cma apron.cma ppl.cma -ccopt "-L$GMP_PREFIX/lib ..."  -cclib "-lap_ppl_caml -lap_ppl -lppl -lgmpxx -lapron_caml -lapron -lgmp_caml  -lmpfr -lgmp -lbigarray -lcamlidl"`

9.1.2 Native-code compilation

You compile and link with

`ocamlopt -I $APRON_PREFIX/lib -c X.ml` and

`ocamlopt -I $APRON_PREFIX/lib -o X -cc "g++" bigarray.cmxa gmp.cmxa apron.cmxa ppl.cmxa X.cmx`

Comments: Same as for bytecode compilation. Do not forget the `-cc "g++"` option. With the `-noautolink` option, the linking command becomes

`ocamlopt -I $APRON_PREFIX/lib -o X -cc "g++" bigarray.cmxa gmp.cmxa apron.cmxa ppl.cmxa -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lap_ppl_caml -lap_ppl -lppl -lgmpxx -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl" X.cmx`

Chapter 10

Module `PolkaGrid` : Reduced product of NewPolka polyhedra and PPL grids

`type 'a t`

Type of abstract values, where `'a` is `Polka.loose` or `Polka.strict`.

`val manager_alloc_loose : unit -> Polka.loose t Apron.Manager.t`

Create a PolkaGrid manager with loose convex polyhedra.

`val manager_alloc_strict : unit -> Polka.strict t Apron.Manager.t`

Create a PolkaGrid manager with strict convex polyhedra.

10.1 Compilation information

10.1.1 Bytecode compilation

To compile to bytecode, you should first generate a custom interpreter with a command which should look like:

`ocamlc -I $APRON_PREFIX/lib -make-runtime -o myrun -cc "g++" bigarray.cma gmp.cma apron.cma polkaGrid.cma`

and then you compile and link your example `X.ml` with

`ocamlc -I $APRON_PREFIX/lib -c X.ml` and

`ocamlc -I $APRON_PREFIX/lib -use-runtime myrun -o X bigarray.cma gmp.cma apron.cma polkaGrid.cma X.cmo`

Comments: The C libraries related to `gmp.cma`, `apron.cma` and `polkaGrid.cma` are automatically looked for (thanks to the auto-linking feature provided by `ocamlc`).

With the `-noautolink` option, the generation of the custom runtime executable should be done with

`ocamlc -I $APRON_PREFIX/lib -noautolink -make-runtime -o myrun bigarray.cma gmp.cma apron.cma polkaGrid.cma -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lpolkaGrid_caml -lap_pkgrid -lpolka -lap_ppl -lppl -lgmpxx -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl"`

10.1.2 Native-code compilation

You compile and link with

`ocamlopt -I $APRON_PREFIX/lib -c X.ml` and

`ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa polkaGrid.cmxa X.cmx`

Comments: Same as for bytecode compilation. With the `-noautolink` option, the linking command becomes

```
ocamlopt -I $APRON_PREFIX/lib -o X bigarray.cmxa gmp.cmxa apron.cmxa polka.cmxa ppl.cmxa
-cc "g++" -ccopt "-L$GMP_PREFIX/lib ..." -cclib "-lpolkaGrid_caml -lap_pkgrid -lpolka -
lap_ppl -lppl -lgmpxx -lapron_caml -lapron -lgmp_caml -lmpfr -lgmp -lbigarray -lcamlidl"
X.cmx
```

Part III

Level 1 of the interface

Chapter 11

Module `Var` : APRON Variables

```
type t
val of_string : string -> t
val compare : t -> t -> int
val to_string : t -> string
val hash : t -> int
```

APRON Variables

```
val print : Format.formatter -> t -> unit
```

Chapter 12

Module `Environment` : APRON Environments binding dimensions to names

```
type typvar =
  | INT
  | REAL
```

```
type t
```

APRON Environments binding dimensions to names

```
val make : Var.t array -> Var.t array -> t
```

Making an environment from a set of integer and real variables. Raise `Failure` in case of name conflict.

```
val add : t -> Var.t array -> Var.t array -> t
```

Adding to an environment a set of integer and real variables. Raise `Failure` in case of name conflict.

```
val remove : t -> Var.t array -> t
```

Remove from an environment a set of variables. Raise `Failure` in case of non-existing variables.

```
val rename : t -> Var.t array -> Var.t array -> t
```

Renaming in an environment a set of variables. Raise `Failure` in case of interferences with the variables that are not renamed.

```
val rename_perm : t -> Var.t array -> Var.t array -> t * Dim.perm
```

Similar to previous function, but returns also the permutation on dimensions induced by the renaming.

```
val lce : t -> t -> t
```

Compute the least common environment of 2 environment, that is, the environment composed of all the variables of the 2 environments. Raise `Failure` if the same variable has different types in the 2 environment.

```
val lce_change : t ->
  t -> t * Dim.change option * Dim.change option
```

Similar to the previous function, but returns also the transformations required to convert from `e1` (resp. `e2`) to the lce. If `None` is returned, this means that `e1` (resp. `e2`) is identic to the lce.

```
val dimchange : t -> t -> Dim.change
```

`dimchange e1 e2` computes the transformation for converting from an environment `e1` to a superenvironment `e2`. Raises `Failure` if `e2` is not a superenvironment.

```
val equal : t -> t -> bool
```

Test equality if two environments

```
val compare : t -> t -> int
```

Compare two environment. `compare env1 env2` return `-2` if the environements are not compatible (a variable has different types in the 2 environements), `-1` if `env1` is a subset of env2, `0` if equality, `+1` if env1 is a superset of env2, and `+2` otherwise (the lce exists and is a strict superset of both)

```
val hash : t -> int
```

Hashing function for environments

```
val dimension : t -> Dim.dimension
```

Return the dimension of the environment

```
val size : t -> int
```

Return the size of the environment

```
val mem_var : t -> Var.t -> bool
```

Return true if the variable is present in the environment.

```
val typ_of_var : t -> Var.t -> typvar
```

Return the type of variables in the environment. If the variable does not belong to the environment, raise a `Failure` exception.

```
val vars : t -> Var.t array * Var.t array
```

Return the (lexicographically ordered) sets of integer and real variables in the environment

```
val var_of_dim : t -> Dim.t -> Var.t
```

Return the variable corresponding to the given dimension in the environment. Raise `Failure` is the dimension is out of the range of the environment (greater than or equal to `dim env`)

```
val dim_of_var : t -> Var.t -> Dim.t
```

Return the dimension associated to the given variable in the environment. Raise `Failure` if the variable does not belong to the environment.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> t -> unit
```

Printing

Chapter 13

Module Linexpr1 : APRON Expressions of level 1

```
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable env : Environment.t ;
}
```

APRON Expressions of level 1

`val make : ?sparse:bool -> Environment.t -> t`

Build a linear expression defined on the given argument, which is sparse by default.

`val minimize : t -> unit`

In case of sparse representation, remove zero coefficients

`val copy : t -> t`

Copy

`val print : Format.formatter -> t -> unit`

Print the linear expression

`val set_list : t -> (Coeff.t * Var.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")] (Some cst)` assigns coefficients `c1` to variable `"x"`, coefficient `c2` to variable `"y"`, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

`val set_array : t -> (Coeff.t * Var.t) array -> Coeff.t option -> unit`

Set simultaneously a number of coefficients, as `set_list`.

`val iter : (Coeff.t -> Var.t -> unit) -> t -> unit`

Iter the function on the pair coefficient/variable of the linear expression

`val get_cst : t -> Coeff.t`

Get the constant

`val set_cst : t -> Coeff.t -> unit`

Set the constant

`val get_coeff : t -> Var.t -> Coeff.t`

Get the coefficient of the variable

`val set_coeff : t -> Var.t -> Coeff.t -> unit`

Set the coefficient of the variable

`val extend_environment : t -> Environment.t -> t`

Change the environment of the expression for a super-environement. Raise `Failure` if it is not the case

`val extend_environment_with : t -> Environment.t -> unit`

Side-effet version of the previous function

`val is_integer : t -> bool`

Does the linear expression depend only on integer variables ?

`val is_real : t -> bool`

Does the linear expression depend only on real variables ?

`val get_linexpr0 : t -> Linexpr0.t`

Get the underlying expression of level 0 (which is not a copy).

`val get_env : t -> Environment.t`

Get the environement of the expression

Chapter 14

Module `Lincons1` : APRON Constraints and array of constraints of level 1

```
type t = {
  mutable lincons0 : Lincons0.t ;
  mutable env : Environment.t ;
}
```

```
type earray = {
  mutable lincons0_array : Lincons0.t array ;
  mutable array_env : Environment.t ;
}
```

APRON Constraints and array of constraints of level 1

```
type typ = Lincons0.typ =
  | EQ
  | SUPEQ
  | SUP
  | DISEQ
  | EQMOD of Scalar.t
```

```
val make : Linexpr1.t -> typ -> t
```

Make a linear constraint. Modifying later the linear expression (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

```
val copy : t -> t
```

Copy (deep copy)

```
val string_of_typ : typ -> string
```

Convert a constraint type to a string (`=`,`>=`, or `>`)

```
val print : Format.formatter -> t -> unit
```

Print the linear constraint

```
val get_typ : t -> typ
```

Get the constraint type

```
val iter : (Coeff.t -> Var.t -> unit) -> t -> unit
```

Iter the function on the pair coefficient/variable of the underlying linear expression

`val get_cst : t -> Coeff.t`

Get the constant of the underlying linear expression

`val set_typ : t -> typ -> unit`

Set the constraint type

`val set_list : t -> (Coeff.t * Var.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")] (Some cst)` assigns coefficients `c1` to variable `"x"`, coefficient `c2` to variable `"y"`, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

`val set_array : t -> (Coeff.t * Var.t) array -> Coeff.t option -> unit`

Set simultaneously a number of coefficients, as `set_list`.

`val set_cst : t -> Coeff.t -> unit`

Set the constant of the underlying linear expression

`val get_coeff : t -> Var.t -> Coeff.t`

Get the coefficient of the variable in the underlying linear expression

`val set_coeff : t -> Var.t -> Coeff.t -> unit`

Set the coefficient of the variable in the underlying linear expression

`val make_unsat : Environment.t -> t`

Build the unsatisfiable constraint $-1 \geq 0$

`val is_unsat : t -> bool`

Is the constraint not satisfiable ?

`val extend_environment : t -> Environment.t -> t`

Change the environement of the constraint for a super-environement. Raise `Failure` if it is not the case

`val extend_environment_with : t -> Environment.t -> unit`

Side-effect version of the previous function

`val get_env : t -> Environment.t`

Get the environement of the linear constraint

`val get_linexpr1 : t -> Linexpr1.t`

Get the underlying linear expression. Modifying the linear expression (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

`val get_lincons0 : t -> Lincons0.t`

Get the underlying linear constraint of level 0. Modifying the constraint of level 0 (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

14.1 Type array

```
val array_make : Environment.t -> int -> earray
```

Make an array of linear constraints with the given size and defined on the given environement. The elements are initialized with the constraint 0=0.

```
val array_print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> earray -> unit
```

Print an array of constraints

```
val array_length : earray -> int
```

Get the size of the array

```
val array_get_env : earray -> Environment.t
```

Get the environment of the array

```
val array_get : earray -> int -> t
```

Get the element of the given index (which is not a copy)

```
val array_set : earray -> int -> t -> unit
```

Set the element of the given index (without any copy). The array and the constraint should be defined on the same environement; otherwise a `Failure` exception is raised.

```
val array_extend_environment : earray -> Environment.t -> earray
```

Change the environement of the array of constraints for a super-environement. Raise `Failure` if it is not the case

```
val array_extend_environment_with : earray -> Environment.t -> unit
```

Side-effect version of the previous function

Chapter 15

Module Generator1 : APRON Generators and array of generators of level 1

```
type t = {
  mutable generator0 : Generator0.t ;
  mutable env : Environment.t ;
}
```

```
type earray = {
  mutable generator0_array : Generator0.t array ;
  mutable array_env : Environment.t ;
}
```

APRON Generators and array of generators of level 1

```
type typ = Generator0.typ =
  | LINE
  | RAY
  | VERTEX
  | LINEMOD
  | RAYMOD
```

```
val make : Linexpr1.t -> Generator0.typ -> t
```

Make a generator. Modifying later the linear expression (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

```
val copy : t -> t
```

Copy (deep copy)

```
val print : Format.formatter -> t -> unit
```

Print the generator

```
val get_typ : t -> Generator0.typ
```

Get the generator type

```
val iter : (Coeff.t -> Var.t -> unit) -> t -> unit
```

Iter the function on the pair coefficient/variable of the underlying linear expression

```
val set_typ : t -> Generator0.typ -> unit
```

Set the generator type

```
val set_list : t -> (Coeff.t * Var.t) list -> unit
```

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")]` assigns coefficients `c1` to variable `"x"` and coefficient `c2` to variable `"y"`.

```
val set_array : t -> (Coeff.t * Var.t) array -> unit
```

Set simultaneously a number of coefficients, as `set_list`.

```
val get_coeff : t -> Var.t -> Coeff.t
```

Get the coefficient of the variable in the underlying linear expression

```
val set_coeff : t -> Var.t -> Coeff.t -> unit
```

Set the coefficient of the variable in the underlying linear expression

```
val extend_environment : t -> Environment.t -> t
```

Change the environement of the generator for a super-environement. Raise `Failure` if it is not the case

```
val extend_environment_with : t -> Environment.t -> unit
```

Side-effect version of the previous function

15.1 Type earray

```
val array_make : Environment.t -> int -> earray
```

Make an array of generators with the given size and defined on the given environement. The elements are initialized with the line 0.

```
val array_print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> earray -> unit
```

Print an array of generators

```
val array_length : earray -> int
```

Get the size of the array

```
val array_get : earray -> int -> t
```

Get the element of the given index (which is not a copy)

```
val array_set : earray -> int -> t -> unit
```

Set the element of the given index (without any copy). The array and the generator should be defined on the same environement; otherwise a `Failure` exception is raised.

```
val array_extend_environment : earray -> Environment.t -> earray
```

Change the environement of the array of generators for a super-environement. Raise `Failure` if it is not the case

```
val array_extend_environment_with : earray -> Environment.t -> unit
```

Side-effect version of the previous function

`val get_env : t -> Environment.t`

Get the environement of the generator

`val get_linexpr1 : t -> Linexpr1.t`

Get the underlying linear expression. Modifying the linear expression (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

`val get_generator0 : t -> Generator0.t`

Get the underlying generator of level 0. Modifying the generator of level 0 (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

Chapter 16

Module `Texpr1` : APRON Expressions of level 1

```
type t = {
  mutable texpr0 : Texpr0.t ;
  mutable env : Environment.t ;
}
```

APRON Expressions of level 1

```
type unop = Texpr0.unop =
  | Neg
  | Cast
  | Sqrt
```

Unary operators

```
type binop = Texpr0.binop =
  | Add
  | Sub
  | Mul
  | Div
  | Mod
```

Binary operators

```
type typ = Texpr0.typ =
  | Real
  | Int
  | Single
  | Double
  | Extended
  | Quad
```

Destination type for rounding

```
type round = Texpr0.round =
  | Near
  | Zero
  | Up
  | Down
  | Rnd
```

Rounding direction

```
type expr =
  | Cst of Coeff.t
  | Var of Var.t
  | Unop of unop * expr * typ * round
  | Binop of binop * expr * expr * typ * round
```

User type for tree expressions

16.1 Constructors and Destructor

```
val of_expr : Environment.t -> expr -> t
```

General constructor (actually the most efficient)

```
val copy : t -> t
```

Copy

```
val of_linexpr : Linexpr1.t -> t
```

Conversion

```
val to_expr : t -> expr
```

General destructor

16.1.1 Incremental constructors

```
val cst : Environment.t -> Coeff.t -> t
val var : Environment.t -> Var.t -> t
val unop : Texpr0.unop -> t -> Texpr0.typ -> Texpr0.round -> t
val binop : Texpr0.binop ->
  t -> t -> Texpr0.typ -> Texpr0.round -> t
```

16.2 Tests

```
val is_interval_cst : t -> bool
val is_interval_linear : t -> bool
val is_interval_polynomial : t -> bool
val is_interval_polyfrac : t -> bool
val is_scalar : t -> bool
```

16.3 Operations

```
val extend_environment : t -> Environment.t -> t
```

Change the environment of the expression for a super-environement. Raise `Failure` if it is not the case

```
val extend_environment_with : t -> Environment.t -> unit
```

Side-effet version of the previous function

```
val get_texpr0 : t -> Texpr0.t
```

Get the underlying expression of level 0 (which is not a copy).

```
val get_env : t -> Environment.t
```

Get the environement of the expression

16.4 Printing

```
val string_of_unop : unop -> string
val string_of_binop : binop -> string
val string_of_typ : typ -> string
val string_of_round : round -> string
val print_unop : Format.formatter -> unop -> unit
val print_binop : Format.formatter -> binop -> unit
val print_typ : Format.formatter -> typ -> unit
val print_round : Format.formatter -> round -> unit
val print_expr : Format.formatter -> expr -> unit
```

Print a tree expression

```
val print : Format.formatter -> t -> unit
```

Print an abstract tree expression

Chapter 17

Module `Tcons1` : APRON tree constraints and array of tree constraints of level 1

```
type t = {
  mutable tcons0 : Tcons0.t ;
  mutable env : Environment.t ;
}
```

```
type earray = {
  mutable tcons0_array : Tcons0.t array ;
  mutable array_env : Environment.t ;
}
```

APRON tree constraints and array of tree constraints of level 1

```
type typ = Lincons0.typ =
  | EQ
  | SUPEQ
  | SUP
  | DISEQ
  | EQMOD of Scalar.t
```

```
val make : Texpr1.t -> typ -> t
```

Make a tree expression constraint. Modifying later the linear expression (*not advisable*) modifies correspondingly the tree expression constraint and conversely, except for changes of environements

```
val copy : t -> t
```

Copy (deep copy)

```
val string_of_typ : typ -> string
```

Convert a constraint type to a string (`=`,`>=`, or `>`)

```
val print : Format.formatter -> t -> unit
```

Print the tree expression constraint

```
val get_typ : t -> typ
```

Get the constraint type

```
val set_typ : t -> typ -> unit
```

Set the constraint type

```
val extend_environment : t -> Environment.t -> t
```

Change the environement of the constraint for a super-environement. Raise `Failure` if it is not the case

```
val extend_environment_with : t -> Environment.t -> unit
```

Side-effect version of the previous function

```
val get_env : t -> Environment.t
```

Get the environement of the tree expression constraint

```
val get_texpr1 : t -> Texpr1.t
```

Get the underlying linear expression. Modifying the linear expression (*not advisable*) modifies correspondingly the tree expression constraint and conversely, except for changes of environements

```
val get_tcons0 : t -> Tcons0.t
```

Get the underlying tree expression constraint of level 0. Modifying the constraint of level 0 (*not advisable*) modifies correspondingly the tree expression constraint and conversely, except for changes of environements

17.1 Type array

```
val array_make : Environment.t -> int -> earray
```

Make an array of tree expression constraints with the given size and defined on the given environement. The elements are initialized with the constraint 0=0.

```
val array_print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> earray -> unit
```

Print an array of constraints

```
val array_length : earray -> int
```

Get the size of the array

```
val array_get_env : earray -> Environment.t
```

Get the environment of the array

```
val array_get : earray -> int -> t
```

Get the element of the given index (which is not a copy)

```
val array_set : earray -> int -> t -> unit
```

Set the element of the given index (without any copy). The array and the constraint should be defined on the same environement; otherwise a `Failure` exception is raised.

```
val array_extend_environment : earray -> Environment.t -> earray
```

Change the environement of the array of constraints for a super-environement. Raise `Failure` if it is not the case

```
val array_extend_environment_with : earray -> Environment.t -> unit
```

Side-effect version of the previous function

Chapter 18

Module `Abstract1` : APRON Abstract values of level 1

```
type 'a t = {
  mutable abstract0 : 'a Abstract0.t ;
  mutable env : Environment.t ;
}
```

APRON Abstract values of level 1

The type parameter `'a` allows to distinguish abstract values with different underlying abstract domains.

```
type box1 = {
  mutable interval_array : Interval.t array ;
  mutable box1_env : Environment.t ;
}
```

18.1 General management

18.1.1 Memory

```
val copy : 'a Manager.t -> 'a t -> 'a t
```

Copy a value

```
val size : 'a Manager.t -> 'a t -> int
```

Return the abstract size of a value

18.1.2 Control of internal representation

```
val minimize : 'a Manager.t -> 'a t -> unit
```

Minimize the size of the representation of the value. This may result in a later recomputation of internal information.

```
val canonicalize : 'a Manager.t -> 'a t -> unit
```

Put the abstract value in canonical form. (not yet clear definition)

```
val approximate : 'a Manager.t -> 'a t -> int -> unit
```

`approximate man abs alg` perform some transformation on the abstract value, guided by the argument `alg`. The transformation may lose information. The argument `alg` overrides the field algorithm of the structure of type `Manager.funopt` associated to ap_abstract0_approximate (commodity feature).

18.1.3 Printing

```
val fdump : 'a Manager.t -> 'a t -> unit
```

Dump on the `stdout` C stream the internal representation of an abstract value, for debugging purposes

```
val print : Format.formatter -> 'a t -> unit
```

Print as a set of constraints

18.1.4 Serialization

18.2 Constructor, accessors, tests and property extraction

18.2.1 Basic constructors

All these functions request explicitly an environment in their arguments.

```
val bottom : 'a Manager.t -> Environment.t -> 'a t
```

Create a bottom (empty) value defined on the given environment

```
val top : 'a Manager.t -> Environment.t -> 'a t
```

Create a top (universe) value defined on the given environment

```
val of_box :
  'a Manager.t ->
  Environment.t -> Var.t array -> Interval.t array -> 'a t
```

Abstract an hypercube.

`of_box man env tvar tinterval` abstracts an hypercube defined by the arrays `tvar` and `tinterval`. The result is defined on the environment `env`, which should contain all the variables in `tvar` (and defines their type)

18.2.2 Accessors

```
val manager : 'a t -> 'a Manager.t
val env : 'a t -> Environment.t
val abstract0 : 'a t -> 'a Abstract0.t
```

Return resp. the underlying manager, environment and abstract value of level 0

18.2.3 Tests

```
val is_bottom : 'a Manager.t -> 'a t -> bool
```

Emptiness test

```
val is_top : 'a Manager.t -> 'a t -> bool
```

Universality test

```
val is_leq : 'a Manager.t -> 'a t -> 'a t -> bool
```

Inclusion test. The 2 abstract values should be compatible.

`val is_eq : 'a Manager.t -> 'a t -> 'a t -> bool`

Equality test. The 2 abstract values should be compatible.

`val sat_lincons : 'a Manager.t -> 'a t -> Lincons1.t -> bool`

Does the abstract value satisfy the linear constraint ?

`val sat_tcons : 'a Manager.t -> 'a t -> Tcons1.t -> bool`

Does the abstract value satisfy the tree expression constraint ?

`val sat_interval : 'a Manager.t -> 'a t -> Var.t -> Interval.t -> bool`

Does the abstract value satisfy the constraint `dim in interval` ?

`val is_variable_unconstrained : 'a Manager.t -> 'a t -> Var.t -> bool`

Is the variable unconstrained in the abstract value ? If yes, this means that the existential quantification of the dimension does not change the value.

18.2.4 Extraction of properties

`val bound_variable : 'a Manager.t -> 'a t -> Var.t -> Interval.t`

Return the interval of variation of the variable in the abstract value.

`val bound_linexpr : 'a Manager.t -> 'a t -> Linexpr1.t -> Interval.t`

Return the interval of variation of the linear expression in the abstract value.

Implement a form of linear programming, where the argument linear expression is the one to optimize under the constraints induced by the abstract value.

`val bound_texpr : 'a Manager.t -> 'a t -> Texpr1.t -> Interval.t`

Return the interval of variation of the tree expression in the abstract value.

`val to_box : 'a Manager.t -> 'a t -> box1`

Convert the abstract value to an hypercube

`val to_lincons_array : 'a Manager.t -> 'a t -> Lincons1.earray`

Convert the abstract value to a conjunction of linear constraints.

Convert the abstract value to a conjunction of tree expressions constraints.

`val to_tcons_array : 'a Manager.t -> 'a t -> Tcons1.earray`

`val to_generator_array : 'a Manager.t -> 'a t -> Generator1.earray`

Convert the abstract value to a set of generators that defines it.

18.3 Operations

18.3.1 Meet and Join

`val meet : 'a Manager.t -> 'a t -> 'a t -> 'a t`

Meet of 2 abstract values.

`val meet_array : 'a Manager.t -> 'a t array -> 'a t`

Meet of a non empty array of abstract values.

```
val meet_lincons_array : 'a Manager.t -> 'a t -> Lincons1.earray -> 'a t
```

Meet of an abstract value with an array of linear constraints.

```
val meet_tcons_array : 'a Manager.t -> 'a t -> Tcons1.earray -> 'a t
```

Meet of an abstract value with an array of tree expressions constraints.

```
val join : 'a Manager.t -> 'a t -> 'a t -> 'a t
```

Join of 2 abstract values.

```
val join_array : 'a Manager.t -> 'a t array -> 'a t
```

Join of a non empty array of abstract values.

```
val add_ray_array : 'a Manager.t -> 'a t -> Generator1.earray -> 'a t
```

Add the array of generators to the abstract value (time elapse operator).

The generators should either lines or rays, not vertices.

18.3.1.0.1 Side-effect versions of the previous functions

```
val meet_with : 'a Manager.t -> 'a t -> 'a t -> unit
val meet_lincons_array_with : 'a Manager.t -> 'a t -> Lincons1.earray -> unit
val meet_tcons_array_with : 'a Manager.t -> 'a t -> Tcons1.earray -> unit
val join_with : 'a Manager.t -> 'a t -> 'a t -> unit
val add_ray_array_with : 'a Manager.t -> 'a t -> Generator1.earray -> unit
```

18.3.2 Assignement and Substitutions

```
val assign_linexpr_array :
  'a Manager.t ->
  'a t ->
  Var.t array -> Linexpr1.t array -> 'a t option -> 'a t
```

Parallel assignement of an array of dimensions by an array of same size of linear expressions

```
val substitute_linexpr_array :
  'a Manager.t ->
  'a t ->
  Var.t array -> Linexpr1.t array -> 'a t option -> 'a t
```

Parallel substitution of an array of dimensions by an array of same size of linear expressions

```
val assign_texpr_array :
  'a Manager.t ->
  'a t ->
  Var.t array -> Texpr1.t array -> 'a t option -> 'a t
```

Parallel assignement of an array of dimensions by an array of same size of tree expressions

```
val substitute_texpr_array :
  'a Manager.t ->
  'a t ->
  Var.t array -> Texpr1.t array -> 'a t option -> 'a t
```

Parallel substitution of an array of dimensions by an array of same size of tree expressions

18.3.2.0.2 Side-effect versions of the previous functions

```
val assign_linexpr_array_with :
  'a Manager.t ->
  'a t ->
  Var.t array -> Linexpr1.t array -> 'a t option -> unit
```

```
val substitute_linexpr_array_with :
  'a Manager.t ->
  'a t ->
  Var.t array -> Linexpr1.t array -> 'a t option -> unit
```

```
val assign_texpr_array_with :
  'a Manager.t ->
  'a t ->
  Var.t array -> Texpr1.t array -> 'a t option -> unit
```

```
val substitute_texpr_array_with :
  'a Manager.t ->
  'a t ->
  Var.t array -> Texpr1.t array -> 'a t option -> unit
```

18.3.3 Projections

These functions implements forgeting (existential quantification) of (array of) variables. Both functional and side-effect versions are provided. The Boolean, if true, adds a projection onto 0-plane.

```
val forget_array : 'a Manager.t -> 'a t -> Var.t array -> bool -> 'a t
```

```
val forget_array_with : 'a Manager.t -> 'a t -> Var.t array -> bool -> unit
```

18.3.4 Change and permutation of dimensions

```
val change_environment :
  'a Manager.t -> 'a t -> Environment.t -> bool -> 'a t
```

Change the environement of the abstract values.

Variables that are removed are first existentially quantified, and variables that are introduced are unconstrained. The Boolean, if true, adds a projection onto 0-plane for these ones.

```
val minimize_environment : 'a Manager.t -> 'a t -> 'a t
```

Remove from the environment of the abstract value and from the abstract value itself variables that are unconstrained in it.

```
val rename_array :
  'a Manager.t ->
  'a t -> Var.t array -> Var.t array -> 'a t
```

Parallel renaming of the environment of the abstract value.

The new variables should not interfere with the variables that are not renamed.

```
val change_environment_with :
  'a Manager.t -> 'a t -> Environment.t -> bool -> unit
```

```
val minimize_environment_with : 'a Manager.t -> 'a t -> unit
```

```
val rename_array_with :
  'a Manager.t -> 'a t -> Var.t array -> Var.t array -> unit
```

18.3.5 Expansion and folding of dimensions

These functions allows to expand one dimension into several ones having the same properties with respect to the other dimensions, and to fold several dimensions into one. Formally,

- expand P(x,y,z) z w = P(x,y,z) inter P(x,y,w) if z is expanded in z and w
- fold Q(x,y,z,w) z w = exists w:Q(x,y,z,w) union (exist z:Q(x,y,z,w))(z←w) if z and w are folded onto z

```
val expand : 'a Manager.t -> 'a t -> Var.t -> Var.t array -> 'a t
```

Expansion: `expand a var tvar` expands the variable `var` into itself and the additional variables in `tvar`, which are given the same type as `var`.

It results in (n+1) unrelated variables having same relations with other variables. The additional variables are added to the environment of the argument for making the environment of the result, so they should not belong to the initial environement.

```
val fold : 'a Manager.t -> 'a t -> Var.t array -> 'a t
```

Folding: `fold a tvar` fold the variables in the array `tvar` of size n≥1 and put the result in the first variable of the array. The other variables of the array are then removed, both from the environment and the abstract value.

```
val expand_with : 'a Manager.t -> 'a t -> Var.t -> Var.t array -> unit
val fold_with : 'a Manager.t -> 'a t -> Var.t array -> unit
```

18.3.6 Widening

```
val widening : 'a Manager.t -> 'a t -> 'a t -> 'a t
```

Widening

```
val widening_threshold :
  'a Manager.t ->
  'a t -> 'a t -> Lincons1.earray -> 'a t
```

18.3.7 Closure operation

```
val closure : 'a Manager.t -> 'a t -> 'a t
```

Closure: transform strict constraints into non-strict ones.

```
val closure_with : 'a Manager.t -> 'a t -> unit
```

Side-effect version

18.4 Additional operations

```
val of_lincons_array :
  'a Manager.t -> Environment.t -> Lincons1.earray -> 'a t
val of_tcons_array : 'a Manager.t -> Environment.t -> Tcons1.earray -> 'a t
```

Abstract a conjunction of constraints

```
val assign_linexpr :
  'a Manager.t ->
  'a t ->
  Var.t -> Linexpr1.t -> 'a t option -> 'a t
val substitute_linexpr :
  'a Manager.t ->
  'a t ->
  Var.t -> Linexpr1.t -> 'a t option -> 'a t
val assign_texpr :
  'a Manager.t ->
  'a t ->
  Var.t -> Texpr1.t -> 'a t option -> 'a t
val substitute_texpr :
  'a Manager.t ->
  'a t ->
  Var.t -> Texpr1.t -> 'a t option -> 'a t
```

Assignement/Substitution of a single dimension by a single expression

```
val assign_linexpr_with :
  'a Manager.t ->
  'a t -> Var.t -> Linexpr1.t -> 'a t option -> unit
val substitute_linexpr_with :
  'a Manager.t ->
  'a t -> Var.t -> Linexpr1.t -> 'a t option -> unit
val assign_texpr_with :
  'a Manager.t ->
  'a t -> Var.t -> Texpr1.t -> 'a t option -> unit
val substitute_texpr_with :
  'a Manager.t ->
  'a t -> Var.t -> Texpr1.t -> 'a t option -> unit
```

Side-effect version of the previous functions

```
val unify : 'a Manager.t -> 'a t -> 'a t -> 'a t
```

Unification of 2 abstract values on their least common environment

```
val unify_with : 'a Manager.t -> 'a t -> 'a t -> unit
```

Side-effect version

Chapter 19

Module Parser : APRON Parsing of expressions

19.1 Introduction

This small module implements the parsing of expressions, constraints and generators. The allowed syntax is simple for linear expressions (no parenthesis) but supports interval expressions. The syntax is more flexible for tree expressions.

19.1.1 Syntax

```
lincons ::= linexpr (’>’ | ’>=’ | ’=’ | ’!=’ | ’=’ | ’<=’ | ’<’) linexpr | linexpr = linexpr ’mod’ scalar
gen ::= (’V:’ | ’R:’ | ’L:’ | ’RM:’ | ’LM:’) linexpr
linexpr ::= linexpr ’+’ linterm | linexpr ’-’ linterm | linterm
linterm ::= coeff [’*’] identifier | coeff | [’-’] identifier
tcons ::= texpr (’>’ | ’>=’ | ’=’ | ’!=’ | ’=’ | ’<=’ | ’<’) texpr | texpr = texpr ’mod’ scalar
texpr ::= coeff | identifier | unop texpr | texpr binop texpr | ’(’ texpr ’)’
binop ::= (’+’|’-’|’*’|’/’|’%’)[’_’(’i’|’f’|’d’|’l’|’q’)][’,’(’n’|’0’|’+oo’|’-oo’)]
unop ::= (’cast’ | ’sqrt’)[’_’(’i’|’f’|’d’|’l’|’q’)][’,’(’n’|’0’|’+oo’|’-oo’)]
coeff ::= scalar | [’-’] ’[’scalar ’;’ scalar ’]’
scalar ::= [’-’] (integer | rational | floating_point_number)
```

For tree expressions `texpr`, by default the operations have an exact arithmetic semantics in the real numbers (even if involved variables are of integer). The type qualifiers modify this default semantics. Their meaning is as follows:

- `i` integer semantics
- `f` IEEE754 32 bits floating-point semantics
- `d` IEEE754 64 bits floating-point semantics
- `l` IEEE754 80 bits floating-point semantics
- `q` IEEE754 129 bits floating-point semantics

By default, the rounding mode is ”any” (this applies only in non-real semantics), which allows to emulate all the following rounding modes:

- `n` nearest
- `0` towards zero
- `+oo` towards infinity
- `-oo` towards minus infinity
- `?` any

19.1.2 Examples

```
let (linexpr:Linexpr1.t) = Parser.linexpr1_of_string env "z+0.4x+2y"
let (tab:Lincons1.earray) = Parser.lincons1_of_lstring env ["1/2x+2/3y=1";"[1;2]<=z+2w";"z+2w<=4";"0<=u'
let (generator:Generator1.t) = Parser.generator1_of_string env "R:x+2y"
let (texpr:Texpr1.t) = Parser.texpr1_of_string env "a %_i,?  b +_f,0 c"
```

19.1.3 Remarks

There is the possibility to parse directly from a lexing buffer, or from a string (from which one can generate a buffer with the function `Lexing.from_string`.

This module uses the underlying modules `Apron_lexer` and `Apron_parser`.

19.2 Interface

```
exception Error of string
```

Raised by conversion functions

```
val linexpr1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Linexpr1.t
val lincons1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Lincons1.t
val generator1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Generator1.t
```

Conversion from lexing buffers to resp. linear expressions, linear constraints and generators, defined on the given environment.

```
val texpr1expr_of_lexbuf : Lexing.lexbuf -> Texpr1.expr
val texpr1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Texpr1.t
val tcons1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Tcons1.t
```

Conversion from lexing buffers to resp. tree expressions and constraints, defined on the given environment.

```
val linexpr1_of_string : Environment.t -> string -> Linexpr1.t
val lincons1_of_string : Environment.t -> string -> Lincons1.t
val generator1_of_string : Environment.t -> string -> Generator1.t
```

Conversion from strings to resp. linear expressions, linear constraints and generators, defined on the given environment.

```
val texpr1expr_of_string : string -> Texpr1.expr
val texpr1_of_string : Environment.t -> string -> Texpr1.t
val tcons1_of_string : Environment.t -> string -> Tcons1.t
```

Conversion from lexing buffers to resp. tree expressions and constraints, defined on the given environment.

```
val lincons1_of_lstring : Environment.t -> string list -> Lincons1.earray
val generator1_of_lstring : Environment.t -> string list -> Generator1.earray
```

Conversion from lists of strings to array of resp. linear constraints and generators, defined on the given environment.

```
val tcons1_of_lstring : Environment.t -> string list -> Tcons1.earray
```

Conversion from lists of strings to array of tree constraints.

```
val of_lstring :
  'a Manager.t -> Environment.t -> string list -> 'a Abstract1.t
```

Abstraction of lists of strings representing constraints to abstract values, on the abstract domain defined by the given manager.

Part IV

Level 0 of the interface

Chapter 20

Module `Dim` : APRON Dimensions and related types

```
type t = int
type change = {
  dim : int array ;
  intdim : int ;
  realdim : int ;
}
type perm = int array
type dimension = {
  intd : int ;
  reald : int ;
}
```

APRON Dimensions and related types

Chapter 21

Module `Linexpr0` : APRON Linear expressions of level 0

`type t`

APRON Linear expressions of level 0

`val make : int option -> t`

Create a linear expression. Its representation is sparse if `None` is provided, dense of size `size` if `Some size` is provided.

`val of_list : int option -> (Coeff.t * Dim.t) list -> Coeff.t option -> t`

Combines `Linexpr0.make`[21] and `Linexpr0.set_list`[21] (see below)

`val of_array : int option -> (Coeff.t * Dim.t) array -> Coeff.t option -> t`

Combines `Linexpr0.make`[21] and `Linexpr0.set_array`[21] (see below)

`val minimize : t -> unit`

In case of sparse representation, remove zero coefficients

`val copy : t -> t`

Copy

`val compare : t -> t -> int`

Comparison with lexicographic ordering using Coeff.cmp, terminating by constant

`val hash : t -> int`

Hashing function

`val get_size : t -> int`

Get the size of the linear expression (which may be sparse or dense)

`val get_cst : t -> Coeff.t`

Get the constant

`val get_coeff : t -> int -> Coeff.t`

Get the coefficient corresponding to the dimension

`val set_list : t -> (Coeff.t * Dim.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,1); (c2,2)] (Some cst)` assigns coefficients `c1` to dimension 1, coefficient `c2` to dimension 2, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

```
val set_array : t -> (Coeff.t * Dim.t) array -> Coeff.t option -> unit
```

Set simultaneously a number of coefficients, as `set_list`.

```
val set_cst : t -> Coeff.t -> unit
```

Set the constant

```
val set_coeff : t -> int -> Coeff.t -> unit
```

Set the coefficient corresponding to the dimension

Iter the function on the pairs coefficient/dimension of the linear expression

```
val iter : (Coeff.t -> Dim.t -> unit) -> t -> unit
```

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print a linear expression, using a function converting from dimensions to names

Chapter 22

Module `Lincons0` : APRON Linear constraints of level 0

```
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable typ : typ ;
}
```

```
type typ =
  | EQ
  | SUPEQ
  | SUP
  | DISEQ
  | EQMOD of Scalar.t
```

APRON Linear constraints of level 0

```
val make : Linexpr0.t -> typ -> t
```

Make a linear constraint. Modifying later the linear expression modifies correspondingly the linear constraint and conversely

```
val copy : t -> t
```

Copy a linear constraint (deep copy)

```
val string_of_typ : typ -> string
```

Convert a constraint type to a string (`=`,`>=`, or `>`)

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print a constraint

Chapter 23

Module `Generator0` : APRON Generators of level 0

```
type typ =
  | LINE
  | RAY
  | VERTEX
  | LINEMOD
  | RAYMOD
```

```
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable typ : typ ;
}
```

APRON Generators of level 0

```
val make : Linexpr0.t -> typ -> t
```

Making a generator. The constant coefficient of the linear expression is ignored. Modifying later the linear expression modifies correspondingly the generator and conversely.

```
val copy : t -> t
```

Copy a generator (deep copy)

```
val string_of_typ : typ -> string
```

Convert a generator type to a string (`LIN`,`RAY`, or `VTX`)

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print a generator

Chapter 24

Module `Texpr0`

```
type t
type unop =
  | Neg
  | Cast
  | Sqrt
```

Unary operators

```
type binop =
  | Add
  | Sub
  | Mul
  | Div
  | Mod
```

Binary operators

```
type typ =
  | Real
  | Int
  | Single
  | Double
  | Extended
  | Quad
```

Destination type for rounding

```
type round =
  | Near
  | Zero
  | Up
  | Down
  | Rnd
```

Rounding direction

APRON tree expressions of level 0

```
type expr =
  | Cst of Coeff.t
  | Dim of Dim.t
```

```
  | Unop of unop * expr * typ * round
  | Binop of binop * expr * expr * typ * round
```

User type for tree expressions

24.1 Constructors and Destructor

```
val of_expr : expr -> t
```

General constructor (actually the most efficient

```
val copy : t -> t
```

Copy

```
val of_linexpr : Linexpr0.t -> t
```

Conversion

```
val to_expr : t -> expr
```

General destructor

24.1.1 Incremental constructors

```
val cst : Coeff.t -> t
val dim : Dim.t -> t
val unop : unop -> t -> typ -> round -> t
val binop : binop ->
  typ -> round -> t -> t -> t
```

24.2 Tests

```
val is_interval_cst : t -> bool
val is_interval_linear : t -> bool
val is_interval_polynomial : t -> bool
val is_interval_polyfrac : t -> bool
val is_scalar : t -> bool
```

24.3 Printing

```
val string_of_unop : unop -> string
val string_of_binop : binop -> string
val string_of_typ : typ -> string
val string_of_round : round -> string
val print_unop : Format.formatter -> unop -> unit
val print_binop : Format.formatter -> binop -> unit
val print_typ : Format.formatter -> typ -> unit
val print_round : Format.formatter -> round -> unit
val print_expr : (Dim.t -> string) -> Format.formatter -> expr -> unit
```

Print a tree expression, using a function converting from dimensions to names

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print an abstract tree expression, using a function converting from dimensions to names

24.4 Internal usage for level 1

```
val print_sprint_unop : unop -> typ -> round -> string
val print_sprint_binop : binop -> typ -> round -> string
val print_precedence_of_unop : unop -> int
val print_precedence_of_binop : binop -> int
```

Chapter 25

Module `Tcons0` : APRON tree expressions constraints of level 0

```
type t = {
  mutable texpr0 : Texpr0.t ;
  mutable typ : Lincons0.typ ;
}
```

APRON tree expressions constraints of level 0

```
type typ = Lincons0.typ =
  | EQ
  | SUPEQ
  | SUP
  | DISEQ
  | EQMOD of Scalar.t
```

```
val make : Texpr0.t -> typ -> t
```

Make a tree expression constraint. Modifying later the tree expression expression modifies correspondingly the tree expression constraint and conversely

```
val copy : t -> t
```

Copy a tree expression constraint (deep copy)

```
val string_of_typ : typ -> string
```

Convert a constraint type to a string (`=`,`>=`, or `>`)

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print a constraint

Chapter 26

Module `Abstract0` : APRON Abstract value of level 0

`type 'a t`

APRON Abstract value of level 0

The type parameter `'a` allows to distinguish abstract values with different underlying abstract domains.

`val set_gc : int -> unit`

TO BE DOCUMENTED

26.1 General management

26.1.1 Memory

`val copy : 'a Manager.t -> 'a t -> 'a t`

Copy a value

`val size : 'a Manager.t -> 'a t -> int`

Return the abstract size of a value

26.1.2 Control of internal representation

`val minimize : 'a Manager.t -> 'a t -> unit`

Minimize the size of the representation of the value. This may result in a later recomputation of internal information.

`val canonicalize : 'a Manager.t -> 'a t -> unit`

Put the abstract value in canonical form. (not yet clear definition)

`val approximate : 'a Manager.t -> 'a t -> int -> unit`

`approximate man abs alg` perform some transformation on the abstract value, guided by the argument `alg`. The transformation may lose information. The argument `alg` overrides the field algorithm of the structure of type `Manager.funopt` associated to ap_abstract0_approximate (commodity feature).

26.1.3 Printing

`val fdump : 'a Manager.t -> 'a t -> unit`

Dump on the `stdout` C stream the internal representation of an abstract value, for debugging purposes

`val print : (int -> string) -> Format.formatter -> 'a t -> unit`

Print as a set of constraints

26.1.4 Serialization

26.2 Constructor, accessors, tests and property extraction

26.2.1 Basic constructors

`val bottom : 'a Manager.t -> int -> int -> 'a t`

Create a bottom (empty) value with the given number of integer and real variables

`val top : 'a Manager.t -> int -> int -> 'a t`

Create a top (universe) value with the given number of integer and real variables

`val of_box : 'a Manager.t -> int -> int -> Interval.t array -> 'a t`

Abstract an hypercube.

`of_box man intdim realdim array` abstracts an hypercube defined by the array of intervals of size `intdim+realdim`

26.2.2 Accessors

`val dimension : 'a Manager.t -> 'a t -> Dim.dimension`

`val manager : 'a t -> 'a Manager.t`

26.2.3 Tests

`val is_bottom : 'a Manager.t -> 'a t -> bool`

Emptiness test

`val is_top : 'a Manager.t -> 'a t -> bool`

Universality test

`val is_leq : 'a Manager.t -> 'a t -> 'a t -> bool`

Inclusion test. The 2 abstract values should be compatible.

`val is_eq : 'a Manager.t -> 'a t -> 'a t -> bool`

Equality test. The 2 abstract values should be compatible.

`val sat_lincons : 'a Manager.t -> 'a t -> Lincons0.t -> bool`

Does the abstract value satisfy the linear constraint ?

`val sat_tcons : 'a Manager.t -> 'a t -> Tcons0.t -> bool`

Does the abstract value satisfy the tree expression constraint ?

`val sat_interval : 'a Manager.t -> 'a t -> Dim.t -> Interval.t -> bool`

Does the abstract value satisfy the constraint `dim in interval` ?

`val is_dimension_unconstrained : 'a Manager.t -> 'a t -> Dim.t -> bool`

Is the dimension unconstrained in the abstract value ? If yes, this means that the existential quantification of the dimension does not change the value.

26.2.4 Extraction of properties

`val bound_dimension : 'a Manager.t -> 'a t -> Dim.t -> Interval.t`

Return the interval of variation of the dimension in the abstract value.

`val bound_linexpr : 'a Manager.t -> 'a t -> Linexpr0.t -> Interval.t`

Return the interval of variation of the linear expression in the abstract value.

Implement a form of linear programming, where the argument linear expression is the one to optimize under the constraints induced by the abstract value.

`val bound_texpr : 'a Manager.t -> 'a t -> Texpr0.t -> Interval.t`

Return the interval of variation of the tree expression in the abstract value.

`val to_box : 'a Manager.t -> 'a t -> Interval.t array`

Convert the abstract value to an hypercube

`val to_lincons_array : 'a Manager.t -> 'a t -> Lincons0.t array`

Convert the abstract value to a conjunction of linear constraints.

`val to_tcons_array : 'a Manager.t -> 'a t -> Tcons0.t array`

Convert the abstract value to a conjunction of tree expression constraints.

`val to_generator_array : 'a Manager.t -> 'a t -> Generator0.t array`

Convert the abstract value to a set of generators that defines it.

26.3 Operations

26.3.1 Meet and Join

`val meet : 'a Manager.t -> 'a t -> 'a t -> 'a t`

Meet of 2 abstract values.

`val meet_array : 'a Manager.t -> 'a t array -> 'a t`

Meet of a non empty array of abstract values.

`val meet_lincons_array : 'a Manager.t -> 'a t -> Lincons0.t array -> 'a t`

Meet of an abstract value with an array of linear constraints.

`val meet_tcons_array : 'a Manager.t -> 'a t -> Tcons0.t array -> 'a t`

Meet of an abstract value with an array of tree expression constraints.

`val join : 'a Manager.t -> 'a t -> 'a t -> 'a t`

Join of 2 abstract values.

```
val join_array : 'a Manager.t -> 'a t array -> 'a t
```

Join of a non empty array of abstract values.

```
val add_ray_array : 'a Manager.t -> 'a t -> Generator0.t array -> 'a t
```

Add the array of generators to the abstract value (time elapse operator).

The generators should either lines or rays, not vertices.

26.3.1.0.3 Side-effect versions of the previous functions

```
val meet_with : 'a Manager.t -> 'a t -> 'a t -> unit
val meet_lincons_array_with :
  'a Manager.t -> 'a t -> Lincons0.t array -> unit
val meet_tcons_array_with : 'a Manager.t -> 'a t -> Tcons0.t array -> unit
val join_with : 'a Manager.t -> 'a t -> 'a t -> unit
val add_ray_array_with : 'a Manager.t -> 'a t -> Generator0.t array -> unit
```

26.3.2 Assignements and Substitutions

```
val assign_linexpr_array :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Linexpr0.t array -> 'a t option -> 'a t
```

Parallel assignement of an array of dimensions by an array of same size of linear expressions

```
val substitute_linexpr_array :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Linexpr0.t array -> 'a t option -> 'a t
```

Parallel substitution of an array of dimensions by an array of same size of linear expressions

```
val assign_texpr_array :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Texpr0.t array -> 'a t option -> 'a t
```

Parallel assignement of an array of dimensions by an array of same size of tree expressions

```
val substitute_texpr_array :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Texpr0.t array -> 'a t option -> 'a t
```

Parallel substitution of an array of dimensions by an array of same size of tree expressions

26.3.2.0.4 Side-effect versions of the previous functions

```
val assign_linexpr_array_with :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Linexpr0.t array -> 'a t option -> unit
val substitute_linexpr_array_with :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Linexpr0.t array -> 'a t option -> unit
```

```
val assign_texpr_array_with :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Texpr0.t array -> 'a t option -> unit
val substitute_texpr_array_with :
  'a Manager.t ->
  'a t ->
  Dim.t array -> Texpr0.t array -> 'a t option -> unit
```

26.3.3 Projections

These functions implements forgeting (existential quantification) of (array of) dimensions. Both functional and side-effect versions are provided. The Boolean, if true, adds a projection onto 0-plane.

```
val forget_array : 'a Manager.t -> 'a t -> Dim.t array -> bool -> 'a t
val forget_array_with : 'a Manager.t -> 'a t -> Dim.t array -> bool -> unit
```

26.3.4 Change and permutation of dimensions

```
val add_dimensions : 'a Manager.t -> 'a t -> Dim.change -> bool -> 'a t
val remove_dimensions : 'a Manager.t -> 'a t -> Dim.change -> 'a t
val permute_dimensions : 'a Manager.t -> 'a t -> Dim.perm option -> 'a t
```

26.3.4.0.5 Side-effect versions of the previous functions

```
val add_dimensions_with : 'a Manager.t -> 'a t -> Dim.change -> bool -> unit
val remove_dimensions_with : 'a Manager.t -> 'a t -> Dim.change -> unit
val permute_dimensions_with : 'a Manager.t -> 'a t -> Dim.perm option -> unit
```

26.3.5 Expansion and folding of dimensions

These functions allows to expand one dimension into several ones having the same properties with respect to the other dimensions, and to fold several dimensions into one. Formally,

- expand P(x,y,z) z w = P(x,y,z) inter P(x,y,w) if z is expanded in z and w
- fold Q(x,y,z,w) z w = exists w:Q(x,y,z,w) union (exist z:Q(x,y,z,w))(z←w) if z and w are folded onto z

```
val expand : 'a Manager.t -> 'a t -> Dim.t -> int -> 'a t
```

Expansion: `expand a dim n` expands the dimension `dim` into itself + `n` additional dimensions. It results in (n+1) unrelated dimensions having same relations with other dimensions. The (n+1) dimensions are put as follows:

- original dimension `dim`
- if the dimension is integer, the n additional dimensions are put at the end of integer dimensions; if it is real, at the end of the real dimensions.

```
val fold : 'a Manager.t -> 'a t -> Dim.t array -> 'a t
```

Folding: `fold a tdim` fold the dimensions in the array `tdim` of size n≥1 and put the result in the first dimension of the array. The other dimensions of the array are then removed (using ap_abstract0_permute_remove_dimensions).

```
val expand_with : 'a Manager.t -> 'a t -> Dim.t -> int -> unit
val fold_with : 'a Manager.t -> 'a t -> Dim.t array -> unit
```

26.3.6 Widening

```
val widening : 'a Manager.t -> 'a t -> 'a t -> 'a t
```

Widening

```
val widening_threshold :
  'a Manager.t ->
  'a t -> 'a t -> Lincons0.t array -> 'a t
```

26.3.7 Closure operation

```
val closure : 'a Manager.t -> 'a t -> 'a t
```

Closure: transform strict constraints into non-strict ones.

```
val closure_with : 'a Manager.t -> 'a t -> unit
```

Side-effect version

26.4 Additional operations

```
val of_lincons_array : 'a Manager.t -> int -> int -> Lincons0.t array -> 'a t
val of_tcons_array : 'a Manager.t -> int -> int -> Tcons0.t array -> 'a t
```

Abstract a conjunction of constraints

```
val assign_linexpr :
  'a Manager.t ->
  'a t ->
  Dim.t -> Linexpr0.t -> 'a t option -> 'a t
val substitute_linexpr :
  'a Manager.t ->
  'a t ->
  Dim.t -> Linexpr0.t -> 'a t option -> 'a t
val assign_texpr :
  'a Manager.t ->
  'a t ->
  Dim.t -> Texpr0.t -> 'a t option -> 'a t
val substitute_texpr :
  'a Manager.t ->
  'a t ->
  Dim.t -> Texpr0.t -> 'a t option -> 'a t
```

Assignement/Substitution of a single dimension by a single expression

```
val assign_linexpr_with :
  'a Manager.t ->
  'a t -> Dim.t -> Linexpr0.t -> 'a t option -> unit
val substitute_linexpr_with :
  'a Manager.t ->
  'a t -> Dim.t -> Linexpr0.t -> 'a t option -> unit
val assign_texpr_with :
  'a Manager.t ->
  'a t -> Dim.t -> Texpr0.t -> 'a t option -> unit
val substitute_texpr_with :
  'a Manager.t ->
  'a t -> Dim.t -> Texpr0.t -> 'a t option -> unit
```

Side-effect version of the previous functions

```
val print_array :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a array -> unit
```

General use

Part V

MLGmpIDL modules

Chapter 27

Module `Mpz` : GMP multi-precision integers

```
type t
```

GMP multi-precision integers

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like `set`, `add`, ...) the first parameter of type `t` is an out-parameter and holds the result when the function returns. For instance, `add x y z` adds the values of `y` and `z` and stores the result in `x`.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation, unlike the corresponding functions in the module `Mpzf`[32].

27.1 Pretty printing

```
val print : Format.formatter -> t -> unit
```

27.2 Initialization Functions

```
val init : unit -> t
val init2 : int -> t
val realloc2 : t -> int -> unit
```

27.3 Assignement Functions

The first parameter holds the result.

```
val set : t -> t -> unit
val set_si : t -> int -> unit
val set_d : t -> float -> unit
val _set_str : t -> string -> int -> unit
val set_str : t -> string -> base:int -> unit
val swap : t -> t -> unit
```

27.4 Combined Initialization and Assignment Functions

```
val init_set : t -> t
```

```
val init_set_si : int -> t
val init_set_d : float -> t
val _init_set_str : string -> int -> t
val init_set_str : string -> base:int -> t
```

27.5 Conversion Functions

```
val get_si : t -> nativeint
val get_int : t -> int
val get_d : t -> float
val get_d_2exp : t -> float * int
val _get_str : int -> t -> string
val get_str : base:int -> t -> string
```

27.6 User Conversions

These functions are additions to or renaming of functions offered by the C library.

```
val to_string : t -> string
val to_float : t -> float
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
```

27.7 Arithmetic Functions

The first parameter holds the result.

```
val add : t -> t -> t -> unit
val add_ui : t -> t -> int -> unit
val sub : t -> t -> t -> unit
val sub_ui : t -> t -> int -> unit
val ui_sub : t -> int -> t -> unit
val mul : t -> t -> t -> unit
val mul_si : t -> t -> int -> unit
val addmul : t -> t -> t -> unit
val addmul_ui : t -> t -> int -> unit
val submul : t -> t -> t -> unit
val submul_ui : t -> t -> int -> unit
val mul_2exp : t -> t -> int -> unit
val neg : t -> t -> unit
val abs : t -> t -> unit
```

27.8 Division Functions

`c` stands for ceiling, `f` for floor, and `t` for truncate (rounds toward 0).

27.8.1 Ceiling division

`val cdiv_q : t -> t -> t -> unit`

The first parameter holds the quotient.

`val cdiv_r : t -> t -> t -> unit`

The first parameter holds the remainder.

`val cdiv_qr : t -> t -> t -> t -> unit`

The two first parameters hold resp. the quotient and the remainder).

`val cdiv_q_ui : t -> t -> int -> int`

The first parameter holds the quotient.

`val cdiv_r_ui : t -> t -> int -> int`

The first parameter holds the remainder.

`val cdiv_qr_ui : t -> t -> t -> int -> int`

The two first parameters hold resp. the quotient and the remainder).

`val cdiv_ui : t -> int -> int`

`val cdiv_q_2exp : t -> t -> int -> unit`

The first parameter holds the quotient.

`val cdiv_r_2exp : t -> t -> int -> unit`

The first parameter holds the remainder.

27.8.2 Floor division

```
val fdiv_q : t -> t -> t -> unit
val fdiv_r : t -> t -> t -> unit
val fdiv_qr : t -> t -> t -> t -> unit
val fdiv_q_ui : t -> t -> int -> int
val fdiv_r_ui : t -> t -> int -> int
val fdiv_qr_ui : t -> t -> t -> int -> int
val fdiv_ui : t -> int -> int
val fdiv_q_2exp : t -> t -> int -> unit
val fdiv_r_2exp : t -> t -> int -> unit
```

27.8.3 Truncate division

```
val tdiv_q : t -> t -> t -> unit
val tdiv_r : t -> t -> t -> unit
val tdiv_qr : t -> t -> t -> t -> unit
val tdiv_q_ui : t -> t -> int -> int
val tdiv_r_ui : t -> t -> int -> int
val tdiv_qr_ui : t -> t -> t -> int -> int
val tdiv_ui : t -> int -> int
val tdiv_q_2exp : t -> t -> int -> unit
val tdiv_r_2exp : t -> t -> int -> unit
```

27.8.4 Other division-related functions

```
val gmod : t -> t -> t -> unit
val gmod_ui : t -> t -> int -> int
val divexact : t -> t -> t -> unit
val divexact_ui : t -> t -> int -> unit
val divisible_p : t -> t -> bool
val divisible_ui_p : t -> int -> bool
val divisible_2exp_p : t -> int -> bool
val congruent_p : t -> t -> t -> bool
val congruent_ui_p : t -> int -> int -> bool
val congruent_2exp_p : t -> t -> int -> bool
```

27.9 Exponentiation Functions

```
val _powm : t -> t -> t -> t -> unit
val _powm_ui : t -> t -> int -> t -> unit
val powm : t -> t -> t -> modulo:t -> unit
val powm_ui : t -> t -> int -> modulo:t -> unit
val pow_ui : t -> t -> int -> unit
val ui_pow_ui : t -> int -> int -> unit
```

27.10 Root Extraction Functions

```
val root : t -> t -> int -> bool
val sqrt : t -> t -> unit
val _sqrtrem : t -> t -> t -> unit
val sqrtrem : t -> remainder:t -> t -> unit
val perfect_power_p : t -> bool
val perfect_square_p : t -> bool
```

27.11 Number Theoretic Functions

```
val probab_prime_p : t -> int -> int
val nextprime : t -> t -> unit
val gcd : t -> t -> t -> unit
val gcd_ui : t option -> t -> int -> int
val _gcdext : t -> t -> t -> t -> t -> unit
val gcdext : gcd:t -> alpha:t -> beta:t -> t -> t -> unit
val lcm : t -> t -> t -> unit
val lcm_ui : t -> t -> int -> unit
val invert : t -> t -> t -> bool
val jacobi : t -> t -> int
val legendre : t -> t -> int
val kronecker : t -> t -> int
val kronecker_si : t -> int -> int
```

```
val si_kronecker : int -> t -> int
val remove : t -> t -> t -> int
val fac_ui : t -> int -> unit
val bin_ui : t -> t -> int -> unit
val bin_uiui : t -> int -> int -> unit
val fib_ui : t -> int -> unit
val fib2_ui : t -> t -> int -> unit
val lucnum_ui : t -> int -> unit
val lucnum2_ui : t -> t -> int -> unit
```

27.12 Comparison Functions

```
val cmp : t -> t -> int
val cmp_d : t -> float -> int
val cmp_si : t -> int -> int
val cmpabs : t -> t -> int
val cmpabs_d : t -> float -> int
val cmpabs_ui : t -> int -> int
val sgn : t -> int
```

27.13 Logical and Bit Manipulation Functions

```
val gand : t -> t -> t -> unit
val ior : t -> t -> t -> unit
val xor : t -> t -> t -> unit
val com : t -> t -> unit
val popcount : t -> int
val hamdist : t -> t -> int
val scan0 : t -> int -> int
val scan1 : t -> int -> int
val setbit : t -> int -> unit
val clrbit : t -> int -> unit
val tstbit : t -> int -> bool
```

27.14 Input and Output Functions: not interfaced

27.15 Random Number Functions: see Gmp_random[31] module

27.16 Integer Import and Export Functions

```
val _import :
  t ->
  (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t ->
  int -> int -> unit
val _export :
  t ->
```

```
  int -> int -> (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t
```

```
val import :
  dest:t ->
  (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t ->
  order:int -> endian:int -> unit
```

```
val export :
  t ->
  order:int ->
  endian:int -> (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t
```

27.17 Miscellaneous Functions

```
val fits_int_p : t -> bool
```

```
val odd_p : t -> bool
```

```
val even_p : t -> bool
```

```
val size : t -> int
```

```
val sizeinbase : t -> int -> int
```

```
val fits_ulong_p : t -> bool
```

```
val fits_slong_p : t -> bool
```

```
val fits_uint_p : t -> bool
```

```
val fits_sint_p : t -> bool
```

```
val fits_ushort_p : t -> bool
```

```
val fits_sshort_p : t -> bool
```

Chapter 28

Module `Mpq` : GMP multiprecision rationals

`type t`

GMP multiprecision rationals

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like `set`, `add`, ...) the first parameter of type `t` is an out-parameter and holds the result when the function returns. For instance, `add x y z` adds the values of `y` and `z` and stores the result in `x`.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation, unlike the corresponding functions in the module `Mpqf`[33].

`val canonicalize : t -> unit`

28.1 Pretty printing

`val print : Format.formatter -> t -> unit`

28.2 Initialization and Assignment Functions

`val init : unit -> t`

`val set : t -> t -> unit`

`val set_z : t -> Mpz.t -> unit`

`val set_si : t -> int -> int -> unit`

`val _set_str : t -> string -> int -> unit`

`val set_str : t -> string -> base:int -> unit`

`val swap : t -> t -> unit`

28.3 Additional Initialization and Assignements functions

These functions are additions to or renaming of functions offered by the C library.

`val init_set : t -> t`

`val init_set_z : Mpz.t -> t`

`val init_set_si : int -> int -> t`

`val init_set_str : string -> base:int -> t`

`val init_set_d : float -> t`

28.4 Conversion Functions

```
val get_d : t -> float
val set_d : t -> float -> unit
val get_z : Mpz.t -> t -> unit
val _get_str : int -> t -> string
val get_str : base:int -> t -> string
```

28.5 User Conversions

These functionss are additions to or renaming of functions offeered by the C library.

```
val to_string : t -> string
val to_float : t -> float
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
val of_frac : int -> int -> t
val of_mpz : Mpz.t -> t
val of_mpz2 : Mpz.t -> Mpz.t -> t
```

28.6 Arithmetic Functions

```
val add : t -> t -> t -> unit
val sub : t -> t -> t -> unit
val mul : t -> t -> t -> unit
val mul_2exp : t -> t -> int -> unit
val div : t -> t -> t -> unit
val div_2exp : t -> t -> int -> unit
val neg : t -> t -> unit
val abs : t -> t -> unit
val inv : t -> t -> unit
```

28.7 Comparison Functions

```
val cmp : t -> t -> int
val cmp_si : t -> int -> int -> int
val sgn : t -> int
val equal : t -> t -> bool
```

28.8 Applying Integer Functions to Rationals

```
val get_num : Mpz.t -> t -> unit
val get_den : Mpz.t -> t -> unit
val set_num : t -> Mpz.t -> unit
val set_den : t -> Mpz.t -> unit
```

28.9 Input and Output Functions: not interfaced

Chapter 29

Module `Mpf` : GMP multiprecision floating-point numbers

`type t`

GMP multiprecision floating-point numbers

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like `set`, `add`, ...) the first parameter of type `t` is an out-parameter and holds the result when the function returns. For instance, `add x y z` adds the values of `y` and `z` and stores the result in `x`.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation.

29.1 Pretty printing

```
val print : Format.formatter -> t -> unit
```

29.2 Initialization and Assignment Functions

```
val set_default_prec : int -> unit
val get_default_prec : unit -> int
val init : unit -> t
val init2 : int -> t
val get_prec : t -> int
val set_prec : t -> int -> unit
val set_prec_raw : t -> int -> unit
val set : t -> t -> unit
val set_si : t -> int -> unit
val set_d : t -> float -> unit
val set_z : t -> Mpz.t -> unit
val set_q : t -> Mpq.t -> unit
val _set_str : t -> string -> int -> unit
val set_str : t -> string -> base:int -> unit
val swap : t -> t -> unit
val init_set : t -> t
val init_set_si : int -> t
```

```
val init_set_d : float -> t
val _init_set_str : string -> int -> t
val init_set_str : string -> base:int -> t
```

29.3 Conversion Functions

```
val get_d : t -> float
val get_d_2exp : t -> float * int
val get_si : t -> nativeint
val get_int : t -> int
val get_z : Mpz.t -> t -> unit
val get_q : Mpq.t -> t -> unit
val _get_str : int -> int -> t -> string * int
val get_str : base:int -> digits:int -> t -> string * int
```

29.4 User Conversions

These functionss are additions to or renaming of functions offered by the C library.

```
val to_string : t -> string
val to_float : t -> float
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
val of_mpz : Mpz.t -> t
val of_mpq : Mpq.t -> t
val is_integer : t -> bool
```

29.5 Arithmetic Functions

```
val add : t -> t -> t -> unit
val add_ui : t -> t -> int -> unit
val sub : t -> t -> t -> unit
val ui_sub : t -> int -> t -> unit
val sub_ui : t -> t -> int -> unit
val mul : t -> t -> t -> unit
val mul_ui : t -> t -> int -> unit
val mul_2exp : t -> t -> int -> unit
val div : t -> t -> t -> unit
val ui_div : t -> int -> t -> unit
val div_ui : t -> t -> int -> unit
val div_2exp : t -> t -> int -> unit
val sqrt : t -> t -> unit
val pow_ui : t -> t -> int -> unit
val neg : t -> t -> unit
val abs : t -> t -> unit
```

29.6 Comparison Functions

```
val cmp : t -> t -> int
val cmp_d : t -> float -> int
val cmp_si : t -> int -> int
val sgn : t -> int
val _equal : t -> t -> int -> bool
val equal : t -> t -> bits:int -> bool
val reldiff : t -> t -> t -> unit
```

29.7 Input and Output Functions: not interfaced

29.8 Random Number Functions: see `Gmp_random`[31] module

29.9 Miscellaneous Float Functions

```
val ceil : t -> t -> unit
val floor : t -> t -> unit
val trunc : t -> t -> unit
val integer_p : t -> bool
val fits_int_p : t -> bool
val fits_ulong_p : t -> bool
val fits_slong_p : t -> bool
val fits_uint_p : t -> bool
val fits_sint_p : t -> bool
val fits_ushort_p : t -> bool
val fits_sshort_p : t -> bool
```

Chapter 30

Module `Mpfr` : MPFR multiprecision floating-point numbers

```
type t
type round =
  | Near
  | Zero
  | Up
  | Down
```

MPFR multiprecision floating-point numbers

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like `set`, `add`, ...) the first parameter of type `t` is an out-parameter and holds the result when the function returns. For instance, `add x y z` adds the values of `y` and `z` and stores the result in `x`.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation.

30.1 Pretty printing

```
val print : Format.formatter -> t -> unit
val print_round : Format.formatter -> round -> unit
val string_of_round : round -> string
```

30.2 Rounding Modes

```
val set_default_rounding_mode : round -> unit
val get_default_rounding_mode : unit -> round
val round_prec : t -> round -> int -> int
```

30.3 Exceptions

```
val get_emin : unit -> int
val get_emax : unit -> int
val set_emin : int -> unit
val set_emax : int -> unit
val check_range : t -> int -> round -> int
```

```
val clear_underflow : unit -> unit
val clear_overflow : unit -> unit
val clear_nanflag : unit -> unit
val clear_inexflag : unit -> unit
val clear_flags : unit -> unit
val underflow_p : unit -> bool
val overflow_p : unit -> bool
val nanflag_p : unit -> bool
val inexflag_p : unit -> bool
```

30.4 Initialization and Assignment Functions

```
val set_default_prec : int -> unit
val get_default_prec : unit -> int
val init : unit -> t
val init2 : int -> t
val get_prec : t -> int
val set_prec : t -> int -> unit
val set_prec_raw : t -> int -> unit
val set : t -> t -> round -> int
val set_si : t -> int -> round -> int
val set_d : t -> float -> round -> int
val set_z : t -> Mpz.t -> round -> int
val set_q : t -> Mpq.t -> round -> int
val _set_str : t -> string -> int -> round -> unit
val set_str : t -> string -> base:int -> round -> unit
val set_f : t -> Mpf.t -> round -> int
val set_inf : t -> int -> unit
val set_nan : t -> unit
val swap : t -> t -> unit
val init_set : t -> round -> int * t
val init_set_si : int -> round -> int * t
val init_set_d : float -> round -> int * t
val init_set_f : Mpf.t -> round -> int * t
val init_set_z : Mpz.t -> round -> int * t
val init_set_q : Mpq.t -> round -> int * t
val _init_set_str : string -> int -> round -> t
val init_set_str : string -> base:int -> round -> t
```

30.5 Conversion Functions

```
val get_d : t -> round -> float
val get_d1 : t -> float
val get_z_exp : Mpz.t -> t -> int
val get_z : Mpz.t -> t -> round -> unit
val _get_str : int -> int -> t -> round -> string * int
val get_str : base:int -> digits:int -> t -> round -> string * int
```

30.6 User Conversions

These functionss are additions to or renaming of functions offered by the C library.

```
val to_string : t -> string
val to_float : ?round:round -> t -> float
val to_mpq : t -> Mpq.t
val of_string : string -> round -> t
val of_float : float -> round -> t
val of_int : int -> round -> t
val of_frac : int -> int -> round -> t
val of_mpz : Mpz.t -> round -> t
val of_mpz2 : Mpz.t -> Mpz.t -> round -> t
val of_mpq : Mpq.t -> round -> t
```

30.7 Basic Arithmetic Functions

```
val add : t -> t -> t -> round -> int
val add_ui : t -> t -> int -> round -> int
val add_z : t -> t -> Mpz.t -> round -> int
val add_q : t -> t -> Mpq.t -> round -> int
val sub : t -> t -> t -> round -> int
val ui_sub : t -> int -> t -> round -> int
val sub_ui : t -> t -> int -> round -> int
val sub_z : t -> t -> Mpz.t -> round -> int
val sub_q : t -> t -> Mpq.t -> round -> int
val mul : t -> t -> t -> round -> int
val mul_ui : t -> t -> int -> round -> int
val mul_z : t -> t -> Mpz.t -> round -> int
val mul_q : t -> t -> Mpq.t -> round -> int
val mul_2ui : t -> t -> int -> round -> int
val mul_2si : t -> t -> int -> round -> int
val mul_2exp : t -> t -> int -> round -> int
val div : t -> t -> t -> round -> int
val ui_div : t -> int -> t -> round -> int
val div_ui : t -> t -> int -> round -> int
val div_z : t -> t -> Mpz.t -> round -> int
val div_q : t -> t -> Mpq.t -> round -> int
val div_2ui : t -> t -> int -> round -> int
val div_2si : t -> t -> int -> round -> int
val div_2exp : t -> t -> int -> round -> int
val sqrt : t -> t -> round -> bool
val sqrt_ui : t -> int -> round -> bool
val pow_ui : t -> t -> int -> round -> bool
```

```
val pow_si : t -> t -> int -> round -> bool
val ui_pow_ui : t -> int -> int -> round -> bool
val ui_pow : t -> int -> t -> round -> bool
val pow : t -> t -> t -> round -> bool
val neg : t -> t -> round -> int
val abs : t -> t -> round -> int
```

30.8 Comparison Functions

```
val cmp : t -> t -> int
val cmp_si : t -> int -> int
val cmp_si_2exp : t -> int -> int -> int
val sgn : t -> int
val _equal : t -> t -> int -> bool
val equal : t -> t -> bits:int -> bool
val nan_p : t -> bool
val inf_p : t -> bool
val number_p : t -> bool
val reldiff : t -> t -> t -> round -> unit
```

30.9 Special Functions

```
val log : t -> t -> round -> int
val log2 : t -> t -> round -> int
val log10 : t -> t -> round -> int
val exp : t -> t -> round -> int
val exp2 : t -> t -> round -> int
val exp10 : t -> t -> round -> int
val cos : t -> t -> round -> int
val sin : t -> t -> round -> int
val tan : t -> t -> round -> int
val sec : t -> t -> round -> int
val csc : t -> t -> round -> int
val cot : t -> t -> round -> int
val sin_cos : t -> t -> t -> round -> bool
val acos : t -> t -> round -> int
val asin : t -> t -> round -> int
val atan : t -> t -> round -> int
val atan2 : t -> t -> t -> round -> int
val cosh : t -> t -> round -> int
val sinh : t -> t -> round -> int
val tanh : t -> t -> round -> int
val sech : t -> t -> round -> int
val csch : t -> t -> round -> int
val coth : t -> t -> round -> int
```

```
val acosh : t -> t -> round -> int
val asinh : t -> t -> round -> int
val atanh : t -> t -> round -> int
val fac_ui : t -> int -> round -> int
val log1p : t -> t -> round -> int
val expm1 : t -> t -> round -> int
val eint : t -> t -> round -> int
val gamma : t -> t -> round -> int
val lngamma : t -> t -> round -> int
val zeta : t -> t -> round -> int
val erf : t -> t -> round -> int
val erfc : t -> t -> round -> int
val fma : t -> t -> t -> t -> round -> int
val agm : t -> t -> t -> round -> int
val hypot : t -> t -> t -> round -> int
val const_log2 : t -> round -> int
val const_pi : t -> round -> int
val const_euler : t -> round -> int
val const_catalan : t -> round -> int
```

30.10 Input and Output Functions: not interfaced

30.11 Miscellaneous Float Functions

```
val rint : t -> t -> round -> int
val ceil : t -> t -> int
val floor : t -> t -> int
val round : t -> t -> int
val trunc : t -> t -> int
val integer_p : t -> bool
```

Chapter 31

Module `Gmp_random` : GMP random generation functions

`type state`

GMP random generation functions

31.1 Random State Initialization

```
val init_default : unit -> state
val init_lc_2exp : Mpz.t -> int -> int -> state
val init_lc_2exp_size : int -> state
```

31.2 Random State Seeding

```
val seed : state -> Mpz.t -> unit
val seed_ui : state -> int -> unit
```

31.3 Random Number Functions

```
module Mpz :
  sig

     val urandomb : Mpz.t -> Gmp_random.state -> int -> unit
     val urandomm : Mpz.t -> Gmp_random.state -> Mpz.t -> unit
     val rrandomb : Mpz.t -> Gmp_random.state -> int -> unit
  end

module Mpf :
  sig

     val urandomb : Mpf.t -> Gmp_random.state -> int -> unit
  end
```

Chapter 32

Module `Mpzf` : GMP multi-precision integers, functional version

Functions in this module has a functional semantics, unlike the corresponding functions in `Mpz`[27]. These functions are less efficients, due to the additional memory allocation neded for the result.

This module could be extended to offer more functions with a functional semantics.

`type t`

multi-precision integer

`val to_mpz : t -> Mpz.t`

`val of_mpz : Mpz.t -> t`

Safe conversion from and to Mpz.t.

There is no sharing between the argument and the result.

`val mpz : t -> Mpz.t`

`val mpzf : Mpz.t -> t`

Unsafe conversion from and to Mpz.t.

The argument and the result actually share the same number: be cautious !

32.1 Constructors

`val of_string : string -> t`

`val of_float : float -> t`

`val of_int : int -> t`

32.2 Conversions and Printing

`val to_string : t -> string`

`val to_float : t -> float`

`val print : Format.formatter -> t -> unit`

32.3 Arithmetic Functions

```
val add : t -> t -> t
val add_int : t -> int -> t
val sub : t -> t -> t
val sub_int : t -> int -> t
val mul : t -> t -> t
val mul_int : t -> int -> t
val cdiv_q : t -> t -> t
val cdiv_r : t -> t -> t
val cdiv_qr : t -> t -> t * t
val fdiv_q : t -> t -> t
val fdiv_r : t -> t -> t
val fdiv_qr : t -> t -> t * t
val tdiv_q : t -> t -> t
val tdiv_r : t -> t -> t
val tdiv_qr : t -> t -> t * t
val divexact : t -> t -> t
val gmod : t -> t -> t
val gcd : t -> t -> t
val lcm : t -> t -> t
val neg : t -> t
val abs : t -> t
```

32.4 Comparison Functions

```
val cmp : t -> t -> int
val cmp_int : t -> int -> int
val sgn : t -> int
```

Chapter 33

Module `Mpqf` : GMP multi-precision rationals, functional version

Functions in this module has a functional semantics, unlike the corresponding functions in `Mpq`[28]. These functions are less efficients, due to the additional memory allocation neded for the result.

`type t`

multi-precision rationals

```
val to_mpq : t -> Mpq.t
val of_mpq : Mpq.t -> t
```

Safe conversion from and to Mpq.t.

There is no sharing between the argument and the result.

```
val mpq : t -> Mpq.t
val mpqf : Mpq.t -> t
```

Unsafe conversion from and to Mpq.t.

The argument and the result actually share the same number: be cautious !

33.1 Constructors

```
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
val of_frac : int -> int -> t
val of_mpz : Mpz.t -> t
val of_mpz2 : Mpz.t -> Mpz.t -> t
val of_mpzf : Mpzf.t -> t
val of_mpzf2 : Mpzf.t -> Mpzf.t -> t
```

33.2 Conversions and Printing

```
val to_string : t -> string
val to_float : t -> float
val to_mpzf2 : t -> Mpzf.t * Mpzf.t
val print : Format.formatter -> t -> unit
```

33.3 Arithmetic Functions

```
val add : t -> t -> t
val sub : t -> t -> t
val mul : t -> t -> t
val div : t -> t -> t
val neg : t -> t
val abs : t -> t
val inv : t -> t
val equal : t -> t -> bool
```

33.4 Comparison Functions

```
val cmp : t -> t -> int
val cmp_int : t -> int -> int
val cmp_frac : t -> int -> int -> int
val sgn : t -> int
```

33.5 Extraction Functions

```
val get_num : t -> Mpzf.t
val get_den : t -> Mpzf.t
```

Chapter 34

Module `Mpfrf` : MPFR multi-precision floating-point version, functional version

Functions in this module has a functional semantics, unlike the corresponding functions in `Mpfr`[30]. These functions do not return the rounding information and are less efficients, due to the additional memory allocation neded for the result.

`type t = Mpfr.t`

multi-precision floating-point numbers

`val to_mpfr : t -> Mpfr.t`

`val of_mpfr : Mpfr.t -> t`

Safe conversion from and to Mpfr.t.

There is no sharing between the argument and the result.

`val mpfr : t -> Mpfr.t`

`val mpfrf : Mpfr.t -> t`

Unsafe conversion from and to Mpfr.t.

The argument and the result actually share the same number: be cautious !

Conversion from and to Mpz.t, Mpq.t and Mpfr.t There is no sharing between the argument and the result.

34.1 Constructors

```
val of_string : string -> Mpfr.round -> t
val of_float : float -> Mpfr.round -> t
val of_int : int -> Mpfr.round -> t
val of_frac : int -> int -> Mpfr.round -> t
val of_mpz : Mpz.t -> Mpfr.round -> t
val of_mpz2 : Mpz.t -> Mpz.t -> Mpfr.round -> t
val of_mpzf : Mpzf.t -> Mpfr.round -> t
val of_mpzf2 : Mpzf.t -> Mpzf.t -> Mpfr.round -> t
val of_mpq : Mpq.t -> Mpfr.round -> t
val of_mpqf : Mpqf.t -> Mpfr.round -> t
```

34.2 Conversions and Printing

```
val to_string : t -> string
val to_float : ?round:Mpfr.round -> t -> float
val to_mpqf : t -> Mpqf.t
val print : Format.formatter -> t -> unit
```

34.3 Arithmetic Functions

```
val add : t -> t -> Mpfr.round -> t
val add_int : t -> int -> Mpfr.round -> t
val sub : t -> t -> Mpfr.round -> t
val sub_int : t -> int -> Mpfr.round -> t
val mul : t -> t -> Mpfr.round -> t
val mul_ui : t -> int -> Mpfr.round -> t
val ui_div : int -> t -> Mpfr.round -> t
val div : t -> t -> Mpfr.round -> t
val div_ui : t -> int -> Mpfr.round -> t
val sqrt : t -> Mpfr.round -> t
val ui_pow : int -> t -> Mpfr.round -> t
val pow : t -> t -> Mpfr.round -> t
val pow_int : t -> int -> Mpfr.round -> t
val neg : t -> Mpfr.round -> t
val abs : t -> Mpfr.round -> t
```

34.4 Comparison Functions

```
val equal : t -> t -> bits:int -> bool
val cmp : t -> t -> int
val cmp_int : t -> int -> int
val sgn : t -> int
val nan_p : t -> bool
val inf_p : t -> bool
val number_p : t -> bool
```

Index