BDDAPRON, version 2.2.0, 30/08/12

Bertrand Jeannet

August 30, 2012

Contents

Part I

Module `Bdd` : Finite-type expressions/properties on top of CUDD

Chapter 1

Introduction

Higher-level interface relying on `Cudd` interface for manipulating BDDs/MTBDDs:

- `Bdd.Reg`[4]: manipulating arrays of BDDs as CPU register, including most ALU (Arithmetic and Logical Unit) operations;
- `Bdd.Int`[6]: signed/unsigned integers encoded as arrays of BDDs;
- `Bdd.Enum`[7]: enumerated types, with management of labels and types (requires `Bdd.Env`[5]);
- `Bdd.Output`[2]: outputing BDDs/MTBDDs (low-level)

Finite-type expressions and properties:

- `Bdd.Env`[5]: environment defining finite-type variables and user-defined enumerated types, and mapping them to BDDs indices;
- `Bdd.Expr0`[10]: general finite-type expressions;
- `Bdd.Domain0`[12]: Boolean formula seen as an (abstract) domain
- `Bdd.Expr1`[11], `Bdd.Domain1`[13]: extends `Bdd.Expr0`[10] and `Bdd.Domain0`[12] by incorporating normalized environments.

Exploits MLCuddIDL[`http://pop-art.inrialpes.fr/~bjeannet/mlxxxidl-forge/mlcuddidl/html/index.html`] (ultimately CUDD[`http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`]) and Camllib[`http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/camllib/html/index.html`] libraries.

Chapter 2

Module `Output`: Output of BDDs/MTBDDs

```
module Output :
  sig
```

2.1 Types

```
type bnode =
  | BIte of int * int * bool * int
```

BIte(idcond,idnodeThen,signElse,idnodeElse)

```
  | BTrue
```

Terminal case. Not needed in principle

BDD node

```
type 'a bdd = {
  cond : int PSette.t Pervasives.ref ;
```

Reachable conditions

```
  mutable bdef : (int, bnode) PMappe.t ;
```

Global BDDs graph

```
  bhash : ('a Cudd.Bdd.t, int) Hashhe.t ;
  mutable blastid : int ;
```

Hashtables and Counters for resp. first free BDD or IDD node

```
}
```

Database

```
type 'a vnode =
  | VIte of int * int * int
```

VIte(idcond,idnodeThen,idnodeElse)

```
  | VCst of 'a
```

Leaf

MTBDD node

```
type 'a vdd = {
  cond : int PSette.t Pervasives.ref ;
```

Reachable conditions

```
  mutable vdef : (int, 'a vnode) PMappe.t ;
```

Global MTBDDs graph

```
  lhash : ('a, unit) PHashhe.t ;
  vhash : ('a Cudd.Vdd.t, int) Hashhe.t ;
  mutable vlastid : int ;
```

Hashtables and Counters for MTBDD nodes.

```
}
```

Database

```
type anode =
  | AIte of int * int * int
```

AIte(idcond,idnodeThen,idnodeElse)

```
  | ACst of float
```

ADD node

```
type add = {
  cond : int PSette.t Pervasives.ref ;
  mutable adef : (int, anode) PMappe.t ;
  mutable lset : float Sette.t ;
  ahash : (Cudd.Add.t, int) Hashhe.t ;
  mutable alastid : int ;
}
```

Database

```
val make_bdd : cond:int PSette.t Pervasives.ref -> 'a bdd
```

2.2 Functions

Create a database for printing BDDs
`cond` allows to share the same set of conditions between BDDs and MTBDDs.

```
val signid_of_bdd : 'a bdd -> 'a Cudd.Bdd.t -> bool * int
```

Output the BDD and return its identifier

```
val make_vdd :
  compare:'a Cudd.PWeakke.compare ->
  cond:int PSette.t Pervasives.ref -> 'a vdd
val make_mtbdd :
  table:'a Cudd.Mtbdd.table ->
  cond:int PSette.t Pervasives.ref -> 'a Cudd.Mtbdd.unique vdd
val make_mtbddc :
  table:'a Cudd.Mtbddc.table ->
  cond:int PSette.t Pervasives.ref -> 'a Cudd.Mtbddc.unique vdd
```

Create a database for printing MTBDDs
`cond` allows to share the same set of conditions between BDDs and MTBDDs.

```
val id_of_vdd : 'a vdd -> 'a Cudd.Vdd.t -> int
```

Output the MTBDD and return its identifier

```
val iter_cond_ordered :
  int PSette.t -> 'a Cudd.Man.t -> (int -> unit) -> unit
```

Iterate the function on all the registered conditions, from level 0 to higher levels.

```
val iter_bdef_ordered : 'a bdd -> (int -> bnode -> unit) -> unit
```

Iterate on definitions of BDD identifiers, in a topological order.

```
val iter_vdef_ordered : 'a vdd -> (int -> 'a vnode -> unit) -> unit
```

Iterate on definitions of MTBDD identifiers, in a topological order.

```
end
```

Chapter 3

Module Normalform: Utility types and functions for normalization

```
module Normalform :
  sig
```

3.1 Types

```
type 'a conjunction =
  | Conjunction of 'a list
```

Conjunction of terms. Empty list means true.

```
  | Cfalse
```

Conjunction

```
type 'a disjunction =
  | Disjunction of 'a list
```

Disjunction of conjunctions. Empty list means false

```
  | Dtrue
```

Disjunction

```
type 'a cnf = 'a disjunction conjunction
```

CNF
DNF

```
type 'a dnf = 'a conjunction disjunction
type ('a, 'b) tree = ('a * bool) list * ('a, 'b) decision
```

Decision tree

```
type ('a, 'b) decision =
  | Leaf of 'b
  | Ite of 'a * ('a, 'b) tree * ('a, 'b) tree
```

3.2 Constants

```
val conjunction_false : 'a conjunction
val conjunction_true : 'a conjunction
val disjunction_false : 'a disjunction
val disjunction_true : 'a disjunction
val cnf_false : 'a cnf
val cnf_true : 'a cnf
val dnf_false : 'a dnf
val dnf_true : 'a dnf
```

3.3 Operations

```
val conjunction_and :
  ?merge:('a list -> 'a list -> 'a list) ->
  'a conjunction ->
  'a conjunction -> 'a conjunction
```

Default merge is `List.rev_append`

```
val disjunction_or :
  ?merge:('a list -> 'a list -> 'a list) ->
  'a disjunction ->
  'a disjunction -> 'a disjunction
```

Default merge is `List.rev_append`

```
val conjunction_and_term : 'a conjunction -> 'a -> 'a conjunction
```

”Merge” is list cons

```
val disjunction_or_term : 'a disjunction -> 'a -> 'a disjunction
```

”Merge” is list cons

```
val minterm_of_tree :
  ?compare:'b PHashhe.compare ->
  ('a, 'b) tree -> (('a * bool) dnf * 'b) list
```

Decompose a decision tree into a disjunction of pairs of a formula and a leaf. It is assumed that leaves can be put in hashtables using `compare` (default value: `PHashhe.compare`). There is no identical leaves (according to `compare`) in the resulting list.

3.3.1 Map functions

```
val rev_map_conjunction : ('a -> 'b) -> 'a conjunction -> 'b conjunction
val rev_map_disjunction : ('a -> 'b) -> 'a disjunction -> 'b disjunction
val rev_map2_conjunction :
  ('a -> 'b -> 'c) ->
  'a conjunction ->
  'b conjunction -> 'c conjunction
val rev_map2_disjunction :
  ('a -> 'b -> 'c) ->
  'a disjunction ->
  'b disjunction -> 'c disjunction
```

```
val map_conjunction : ('a -> 'b) -> 'a conjunction -> 'b conjunction
val map_disjunction : ('a -> 'b) -> 'a disjunction -> 'b disjunction
val map_cnf : ('a -> 'b) -> 'a cnf -> 'b cnf
val map_dnf : ('a -> 'b) -> 'a dnf -> 'b dnf
val map_tree : ('a -> 'c) ->
  ('b -> 'd) -> ('a, 'b) tree -> ('c, 'd) tree
```

3.4 Printing functions

```
val print_conjunction :
  ?firstconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastconj:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a conjunction -> unit
val print_disjunction :
  ?firstdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastdisj:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a disjunction -> unit
val print_cnf :
  ?firstconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastdisj:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a cnf -> unit
val print_dnf :
  ?firstdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastdisj:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepconj:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastconj:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a dnf -> unit
val print_tree_minterm :
  ?compare:'a PHashhe.compare ->
  (Format.formatter -> ('b * bool) dnf -> unit) ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> ('b, 'a) tree -> unit
val print_tree :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) tree -> unit
end
```

Chapter 4

Module Reg: Bounded unsigned integer expressions with BDDs

```
module Reg :
  sig
```

This module encode the classical arithmetic and logical operations on arrays of bits, each bit being defined by a BDD. It doesn't need any initialization, and there is no in-place modification.

The type handled by the module is an array of BDDs, which represent a processor-like register with the Least Significant Bit in first position.

This module requires the mlcuddidl library.

```
type 'a t = 'a Cudd.Bdd.t array
```

type of arrays of bits

```
type dt = Cudd.Man.d t
type vt = Cudd.Man.v t
val lnot : 'a t -> 'a t
```

4.1 Logical operations

Logical negation (for all bits).

```
val shift_left : 'a Cudd.Man.t -> int -> 'a t -> 'a t * 'a Cudd.Bdd.t
```

`shift_left man t n` shifts the register to the left by `n` bits. Returns the resulting register and the carry, which contains the last bit shifted out of the register. Assume, as for the following functions, that `n` is between `1` and the size of the register.

```
val shift_right : 'a Cudd.Man.t -> int -> 'a t -> 'a t * 'a Cudd.Bdd.t
```

`shift_right t n` shifts the register to the right by `n` bits. This an *arithmetical* shift: the sign is inserted as the new most significant bits. Returns the resulting register and the carry, which contains the last bit shifted out of the register.

```
val shift_right_logical :
  'a Cudd.Man.t -> int -> 'a t -> 'a t * 'a Cudd.Bdd.t
```

Same as `shift_right`, but here *logical* shift: a zero is always inserted.

```
val extend : 'a Cudd.Man.t -> signed:bool -> int -> 'a t -> 'a t
```

register extension or truncation. `extend ~signed:b n x` extends the register x by adding `n` most significant bits, if `n>0`, or truncate the `-n` most significant bits if `n<0`. `b` indicates whether the register is considered as a signed one or not.

4.2 Arithmetic operations

```
val succ : 'a Cudd.Man.t -> 'a t -> 'a t * 'a Cudd.Bdd.t
```

Successor operation; returns the new register and the carry.

```
val pred : 'a Cudd.Man.t -> 'a t -> 'a t * 'a Cudd.Bdd.t
```

Predecessor operation; returns the new register and the carry.

```
val add :
  'a Cudd.Man.t ->
  'a t -> 'a t -> 'a t * 'a Cudd.Bdd.t * 'a Cudd.Bdd.t
```

Addition; returns the new registerm, the carry, and the overflow (for signed integers).

```
val sub :
  'a Cudd.Man.t ->
  'a t -> 'a t -> 'a t * 'a Cudd.Bdd.t * 'a Cudd.Bdd.t
```

Substraction; returns the new register, the carry, and the overflow (for signed integers).

```
val neg : 'a t -> 'a t
```

Unary negation; be cautious, if the size of integer is `n`, the negation of `-2^(n-1)` is itself.

```
val scale : int -> 'a t -> 'a t
```

Multiplication by a positive constant.

```
val mul : 'a t -> 'a t -> 'a t
```

(Unsigned) multiplication

```
val ite : 'a Cudd.Bdd.t -> 'a t -> 'a t -> 'a t
```

if-then-else operation. Zero-size possible.

4.3 Predicates

```
val is_cst : 'a t -> bool
```

Tests whether it contains a constant value. Zero-size possible.

```
val zero : 'a Cudd.Man.t -> 'a t -> 'a Cudd.Bdd.t
```

Tests the register to zero.

```
val equal : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

Equality test.

```
val greatereq : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

Signed greater-or-equal test.

```
val greater : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

Signed strictly-greater-than test.

```
val highereq : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

Unsigned greater-or-equal test.

```
val higher : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

Unsigned strictly-greater-than test.

4.4 Constants: conversion and predicates

```
val min_size : int -> int
```

`min_size cst` computes the minimum number of bits required to represent the given constant. We have for example `min_size 0=0`, `min_size 1 = 1`, `min_size 3 = 2`, `min_size (-8) = 4`.

```
val of_int : 'a Cudd.Man.t -> int -> int -> 'a t
```

`of_int size cst` puts the constant integer `cst` in a constant register of size `size`. The fact that `size` is big enough is checked using the previous function, and a `Failure "..."` exception is raised in case of problem.

```
val to_int : signed:bool -> 'a t -> int
```

`to_int sign x` converts a constant register to an integer. `sign` indicates whether the register is to be interpreted as a signed or unsigned.

```
val equal_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val greatereq_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val greater_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val highereq_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val higher_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
```

Tests w.r.t. a constant register, the size of which is defined by the first given register.

4.5 Decomposition in guarded form

```
module Minterm :
  sig
```

```
    type t = Cudd.Man.tbool array
```

Type of a minterm: an array of Booleans extend with undefined value, indexed by variable indices.

```
    val is_indet : t -> bool
```

Is the minterm completely non-determinated ? (ie, contain only undefined values)

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

Convert a possibly negative integer into a minterm of size `size`

```
    val to_int : signed:bool -> t -> int
```

Convert a minterm to a (possibly signed) integer. Raise `Invalid_argument` if the minterm is not deterministic.

`val iter : (t -> unit) -> t -> unit`

Iterate the function on all determinated minterms represented by the argument minterm.

`val map : (t -> 'a) -> t -> 'a list`

Apply the function to all determinated minterms represented by the argument minterm and return the list of the results.

`end`

`val guard_of_minterm : 'a Cudd.Man.t -> 'a t -> Minterm.t -> 'a Cudd.Bdd.t`

Return the guard of the deterministic minterm in the BDD register.

`val guard_of_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t`

Return the guard of the integer value in the BDD register.

```
val guardints :
  'a Cudd.Man.t -> signed:bool -> 'a t -> ('a Cudd.Bdd.t * int) list
```

Return the list `g -> n` represented by the BDD register.

4.6 Evaluation

`val cofactor : 'a t -> 'a Cudd.Bdd.t -> 'a t`

`val restrict : 'a t -> 'a Cudd.Bdd.t -> 'a t`

`val tdrestrict : 'a t -> 'a Cudd.Bdd.t -> 'a t`

4.7 Printing

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

`print f fmt t` prints the register `t` using the formatter `fmt` and the function `f` to print BDDs indices.

```
val print_minterm :
  signed:bool ->
  (Format.formatter -> 'a Cudd.Bdd.t -> unit) ->
  Format.formatter -> 'a t -> unit
```

`print_minterm f fmt t` prints the register `t` using the formatter `fmt` and the function `f` to convert BDDs indices to names.

`val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t`

Permutation (scale `Cudd.Bdd.permute` and `Cudd.Bdd.permute_memo`)

`val varmap : 'a t -> 'a t`

Permutation (scale `Cudd.Bdd.varmap`)

`val vectorcompose : ?memo:Cudd.Memo.t -> 'a Cudd.Bdd.t array -> 'a t -> 'a t`

Composition (scale `Cudd.Bdd.vectorcompose` and `Cudd.Bdd.vectorcompose_memo`)

`end`

Chapter 5

Module Env: Normalized managers/environments

```
module Env :
  sig
```

5.1 Types

```
exception Bddindex
```

```
type 'a typdef = [ `Benum of 'a array ]
```

Type defintions. 'a is the type of symbols (typically, string).

```
type 'a typ = [ `Benum of 'a | `Bint of bool * int | `Bool ]
```

Types. 'a is the type of symbols (typically, string.

```
type 'a symbol = {
  compare : 'a -> 'a -> int ;
```

Total order

```
  marshal : 'a -> string ;
```

Conversion to string. The generated strings SHOULD NOT contain NULL character, as they may be converted to C strings.

```
  unmarshal : string -> 'a ;
```

Conversion from string

```
  mutable print : Format.formatter -> 'a -> unit ;
```

Printing

```
}
```

Manager for manipulating symbols.
DO NOT USE Marshal.to_string and Marshal.from_string, as they generate strings with NULL character, which is not handled properly when converted to C strings.
You may use instead Bdd.Env.marshal[5.3] and Bdd.Env.unmarshal[5.3].

```
type ('a, 'b, 'c, 'd, 'e) t0 = {
  mutable cudd : 'd Cudd.Man.t ;
```

CUDD manager

```
  mutable typdef : ('a, 'c) PMappe.t ;
```

Named types definitions

```
  mutable vartyp : ('a, 'b) PMappe.t ;
```

Associate to a var/label its type

```
  mutable bddindex0 : int ;
```

First index for finite-type variables

```
  mutable bddsize : int ;
```

Number of indices dedicated to finite-type variables

```
  mutable bddindex : int ;
```

Next free index in BDDs used by `self#add_var`.

```
  mutable bddincr : int ;
```

Increment used by `add_var` for incrementing `bddindex`

```
  mutable idcondvar : (int, 'a) PMappe.t ;
```

Associates to a BDD index the variable involved by it

```
  mutable vartid : ('a, int array) PMappe.t ;
```

(Sorted) array of BDD indices associated to finite-type variables.

```
  mutable varset : ('a, 'd Cudd.Bdd.t) PMappe.t ;
```

Associates to enumerated variable the (care)set of possibled values.

```
  mutable print_external_idcondb : Format.formatter -> int * bool -> unit ;
```

Printing conditions not managed by the environment.. By default, `pp_print_int`.

```
  mutable ext : 'e ;
  symbol : 'a symbol ;
  copy_ext : 'e -> 'e ;
}
```

Environment

- 'a is the type of symbols;
- 'b is the type of variables type;
- 'c is the type of type definitions;
- 'd is the type of CUDD managers and BDDs (`Cudd.Man.d` or `Cudd.Man.v`);
- 'e is the type of "extension"

```
module O :
  sig
    type ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) t = ('a, [> 'a Bdd.Env.typ ] as 'b,
      Bdd.Env.t0
    val make :
      symbol:'a Bdd.Env.symbol ->
      copy_ext:('e -> 'e) ->
      ?bddindex0:int ->
      ?bddsize:int ->
      ?relational:bool ->
      'd Cudd.Man.t ->
      'e -> ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) t
    val print :
      (Format.formatter -> ([> 'a Bdd.Env.typ ] as 'b) -> unit) ->
      (Format.formatter -> ([> 'a Bdd.Env.typdef ] as 'c) -> unit) ->
      (Format.formatter -> 'e -> unit) ->
      Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
```

Print an environment

```
end
```

5.1.1 Opened signature

```
type ('a, 'd) t = ('a, 'a typ, 'a typdef, 'd, unit) O.t
val print_typ :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> [> 'a typ ] -> unit
```

5.2 Printing

Print a type

```
val print_typdef :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> [> 'a typdef ] -> unit
```

Print a type definition

```
val print_tid : Format.formatter -> int array -> unit
val print_idcondb :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  Format.formatter -> int * bool -> unit
val print_order :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  Format.formatter -> unit
```

Print the BDD variable ordering

```
val print :
  Format.formatter ->
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t -> unit
```

Print an environment

5.3 Constructors

```
val marshal : 'a -> string
```

Safe marshalling function, generating strings without NULL characters.
(Based on `Marshal.to_string` with `Marshal.No_sharing` option.)

```
val unmarshal : string -> 'a
```

Companion unmarshalling function

```
val make_symbol :
  ?compare:('a -> 'a -> int) ->
  ?marshal:('a -> string) ->
  ?unmarshal:(string -> 'a) ->
  (Format.formatter -> 'a -> unit) -> 'a symbol
```

Generic function for creating a manager for symbols. Default values are `Pervasives.compare`, `Bdd.Env.marshal`[5.3] and `Bdd.Env.unmarshal`[5.3].

DO NOT USE `Marshal.to_string` and `Marshal.from_string`, as they generate strings with NULL character, which is not handled properly when converted to C strings.

```
val string_symbol : string symbol
```

Standard manager for symbols of type `string`

```
val make :
  symbol:'a symbol ->
  ?bddindex0:int ->
  ?bddsize:int -> ?relational:bool -> 'd Cudd.Man.t -> ('a, 'd) t
```

Create a new environment.

- `symbol` is the manager for manipulating symbols;
- `bddindex0`: starting index in BDDs for finite-type variables;
- `bddsize`: number of indices booked for finite-type variables. If at some point, there is no such available index, a `Failure` exception is raised.
- `relational`: if true, primed indices (unprimed indices plus one) are booked together with unprimed indices. `bddincr` is initialized to 1 if `relational=false`, 2 otherwise.

Default values for `bddindex0,bddsize,relational` are `0,100,false`.

```
val make_string :
  ?bddindex0:int ->
  ?bddsize:int -> ?relational:bool -> 'd Cudd.Man.t -> (string, 'd) t
```

```
make_string XXX = make ~symbol:string_symbol XXX
```

```
val copy :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t
```

Copy

5.4 Accessors

```
val mem_typ : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a -> bool
```

Is the type defined in the database ?

```
val mem_var : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a -> bool
```

Is the label/var defined in the database ?

```
val mem_label : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a -> bool
```

Is the label a label defined in the database ?

```
val typdef_of_typ :
  ('a, [> 'a typ ], [> 'a typdef ] as 'b, 'd, 'e) O.t ->
  'a -> 'b
```

Return the definition of the type

```
val typ_of_var :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ], 'd, 'e) O.t ->
  'a -> 'b
```

Return the type of the label/variable

```
val vars : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a PSette.t
```

Return the list of variables (not labels)

```
val labels : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a PSette.t
```

Return the list of labels (not variables)

5.5 Adding types and variables

```
val add_typ_with :
  ('a, [> 'a typ ], [> 'a typdef ] as 'b, 'd, 'e) O.t ->
  'a -> 'b -> unit
```

Declaration of a new type

```
val add_vars_with :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ], 'd, 'e) O.t ->
  ('a * 'b) list -> int array option
```

Add the set of variables, possibly normalize the environment and return the applied permutation (that should also be applied to expressions defined in this environment)

```
val remove_vars_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a list -> int array option
```

Remove the set of variables, and possibly normalize the environment and return the applied permutation.

```
val rename_vars_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  ('a * 'a) list -> int array option
```

Rename the variables, possibly normalize the environment and return the applied permutation.

```
val add_typ :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> 'a -> 'c -> ('a, 'b, 'c, 'd, 'e) O.t
val add_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a * 'b) list -> ('a, 'b, 'c, 'd, 'e) O.t
val remove_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> 'a list -> ('a, 'b, 'c, 'd, 'e) O.t
val rename_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a * 'a) list -> ('a, 'b, 'c, 'd, 'e) O.t
```

Functional versions of the previous functions

```
val add_var_with :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ], 'd, 'e) O.t ->
  'a -> 'b -> unit
```

Addition without normalization (internal)

5.6 Operations

```
val iter_ordered :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  ('a -> int array -> unit) -> unit
```

Iter on all finite-state variables declared in the database

```
val is_leq :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t -> bool
```

Test inclusion of environments in terms of types and variables (but not in term of indexes)

```
val is_eq :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t -> bool
```

Test equality of environments in terms of types and variables (but not in term of indexes)

```
val shift :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> int -> ('a, 'b, 'c, 'd, 'e) O.t
```

Shift all the indices by the offset

```
val lce :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t ->
  ('a, 'b, 'c, 'd, 'e) O.t -> ('a, 'b, 'c, 'd, 'e) O.t
```

Least common environment

```
val permutation12 :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t -> int array
```

Permutation for going from a subenvironment to a superenvironment

```
val permutation21 :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t -> int array
```

Permutation from a superenvironment to a subenvironment

5.7 Precomputing change of environments

```
type 'a change = {
  intro : int array option ;
```

Permutation to apply for making space for new BDD variables

```
  remove : ('a Cudd.Bdd.t * int array) option ;
```

BDD variables to existentially quantify out, and permutation to apply

```
}
```

Contain the computed information to switch from one environment to another one.

```
val compute_change :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> ('a, 'b, 'c, 'd, 'e) O.t -> 'd change
```

5.8 Utilities

```
val notfound : ('a, Format.formatter, unit, 'b) Pervasives.format4 -> 'a
```

```
type ('a, 'b) value = {
  env : 'a ;
  val0 : 'b ;
}
```

Type of pairs `(environment, value)`

```
val make_value :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> 'f -> (('a, 'b, 'c, 'd, 'e) O.t, 'f) value
```

Constructor

```
val get_env : ('a, 'b) value -> 'a
```

```
val get_val0 : ('a, 'b) value -> 'b
```

```
val extend_environment :
  ('f -> int array -> 'f) ->
  (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
   O.t, 'f)
  value ->
  ('a, 'b, 'c, 'd, 'e) O.t ->
  (('a, 'b, 'c, 'd, 'e) O.t, 'f) value
```

`extend_environment permute value env` embed `value` in the new (super)environment `env`, by computing the permutation transformation and using `permute` to apply it to the value.

5.9 Internal functions

```
val compare_idb : int * bool -> int * bool -> int
```

Comparison

5.9.1 Normalisation

```
val permutation :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  int array
```

Compute the permutation for normalizing the environment

```
val permute_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  int array -> unit
```

Apply the given permutation to the environment

```
val normalize_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  int array
```

Combine the two previous functions, and return the permutation

```
val check_normalized : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t -> bool
```

Prints error message and returns `false` if not normalized

5.9.2 Permutations

```
val compose_permutation : int array -> int array -> int array
val compose_opermutation :
  int array option -> int array option -> int array option
val permutation_of_offset : int -> int -> int array
```

5.9.3 Used by level1 APIs

```
val check_var : ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a -> unit
val check_lvar :
  ('a, [> 'a typ ], [> 'a typdef ], 'd, 'e) O.t ->
  'a list -> unit
val check_value :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t -> (('a, 'b, 'c, 'd, 'e) O.t, 'f) value -> unit
val check_value2 :
  (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
   O.t, 'f)
  value -> (('a, 'b, 'c, 'd, 'e) O.t, 'g) value -> unit
val check_value3 :
  (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
   O.t, 'f)
  value ->
  (('a, 'b, 'c, 'd, 'e) O.t, 'g) value ->
  (('a, 'b, 'c, 'd, 'e) O.t, 'h) value -> unit
val check_lvarvalue :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t ->
  ('a * (('a, 'b, 'c, 'd, 'e) O.t, 'f) value) list ->
  ('a * 'f) list
val check_lvalue :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t ->
  (('a, 'b, 'c, 'd, 'e) O.t, 'f) value list -> 'f list
val check_ovalue :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
  O.t ->
  (('a, 'b, 'c, 'd, 'e) O.t, 'f) value option -> 'f option
val mapunop :
```

```
      ('f -> 'g) ->
      (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
       O.t, 'f)
      value -> (('a, 'b, 'c, 'd, 'e) O.t, 'g) value
    val mapbinop :
      ('f -> 'g -> 'h) ->
      (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
       O.t, 'f)
      value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'g) value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'h) value
    val mapbinope :
      (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
       O.t -> 'f -> 'g -> 'h) ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'f) value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'g) value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'h) value
    val mapterop :
      ('f -> 'g -> 'h -> 'i) ->
      (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd, 'e)
       O.t, 'f)
      value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'g) value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'h) value ->
      (('a, 'b, 'c, 'd, 'e) O.t, 'i) value
  end
```

Chapter 6

Module `Int`: Bounded integer expressions with BDDs

```
module Int :
  sig
```

This module is intended to encode operations based on bounded integers (or more generally, enumerated types), using BDDs. It represents a layer on top of `Bddarith`, which provides with arrays resizing and signs.

Let us give an example: suppose we have a variable `i: int[0..7]`; this variable will be encoded by 3 Boolean variables `i0,i1,i2`, starting from the least significant bit. Now, how to translate the expression `i<=5` ? Here the result will be `i2 and not i1 or not i2`

This module requires the mlcuddidl library and the `Bddreg` module.

The basic types are signed or unsigned n-bits integers. Integers are defined by a an array of BDDs, each BDD corresponding to a bit. The order in this array is going from the LSB (Least Significant Bit) to the MSB (Most Significant Bit).

```
type 'a t = {
  signed : bool ;
  reg : 'a Cudd.Bdd.t array ;
}
```

type of an enumerated variable

```
type dt = Cudd.Man.d t
```

```
type vt = Cudd.Man.v t
```

```
exception Typing of string
```

Raised when operands are of incompatible type (sign and size)

All the functions are basically wrapper around functions of `Bddarith`. Resizing and conversion from unsigned to signed are automatic. For instance, When adding or substracting integers, the smallest one is resized to the size of the larger one, possibly with one more bit if they are not of the same type (signed or unsigned).

6.1 Conversion of integers

```
val extend : 'a Cudd.Man.t -> int -> 'a t -> 'a t
```

6.2 Operations on integers

```
val neg : 'a t -> 'a t
val succ : 'a t -> 'a t
val pred : 'a t -> 'a t
val add : 'a t -> 'a t -> 'a t
val sub : 'a t -> 'a t -> 'a t
val mul : 'a t -> 'a t -> 'a t
val shift_left : int -> 'a t -> 'a t
val shift_right : int -> 'a t -> 'a t
val scale : int -> 'a t -> 'a t
val ite : 'a Cudd.Bdd.t -> 'a t -> 'a t -> 'a t
```

6.3 Predicates on integers

```
val is_cst : 'a t -> bool
val zero : 'a Cudd.Man.t -> 'a t -> 'a Cudd.Bdd.t
val equal : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
val greatereq : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
val greater : 'a Cudd.Man.t -> 'a t -> 'a t -> 'a Cudd.Bdd.t
```

6.4 Predicates involving constant integers

```
val of_int : 'a Cudd.Man.t -> bool -> int -> int -> 'a t
val to_int : 'a t -> int
val equal_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val greatereq_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
val greater_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
```

6.5 Decomposition in guarded form

```
module Minterm :
  sig
```

```
    val iter : signed:bool -> (int -> unit) -> Bdd.Reg.Minterm.t -> unit
```

Iterate the function on all the integer values represented by the argument minterm.

```
    val map : signed:bool -> (int -> 'a) -> Bdd.Reg.Minterm.t -> 'a list
```

Apply the function to all integer values represented by the argument minterm and return the list of the results.

```
  end
```

```
val guard_of_int : 'a Cudd.Man.t -> 'a t -> int -> 'a Cudd.Bdd.t
```

Return the guard of the integer value in the BDD register.

```
val guardints : 'a Cudd.Man.t -> 'a t -> ('a Cudd.Bdd.t * int) list
```

Return the list `g -> n` represented by the BDD register.

```
val cofactor : 'a t -> 'a Cudd.Bdd.t -> 'a t
```

6.6 Evaluation

```
val restrict : 'a t -> 'a Cudd.Bdd.t -> 'a t
val tdrestrict : 'a t -> 'a Cudd.Bdd.t -> 'a t
```

6.7 Printing

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

print f fmt t prints the register t using the formatter fmt and the function f to print BDDs indices.

```
val print_minterm :
  (Format.formatter -> 'a Cudd.Bdd.t -> unit) ->
  Format.formatter -> 'a t -> unit
```

print_minterm f fmt t prints the register t using the formatter fmt and the function f to convert BDDs indices to names.

```
val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t
```

Permutation (scale Cudd.Bdd.permute and Cudd.Bdd.permute_memo)

```
val varmap : 'a t -> 'a t
```

Permutation (scale Cudd.Bdd.varmap)

```
val vectorcompose : ?memo:Cudd.Memo.t -> 'a Cudd.Bdd.t array -> 'a t -> 'a t
```

Composition (scale Cudd.Bdd.vectorcompose and Cudd.Bdd.vectorcompose_memo)

```
end
```

Chapter 7

Module Enum: Enumerated expressions with BDDs

```
module Enum :
  sig
```

7.1 Types

```
type 'a typ = [ `Benum of 'a ]
```

A type is just a name

```
type 'a typdef = [ `Benum of 'a array ]
```

An enumerated type is defined by its (ordered) set of labels

7.1.1 Datatype representing a BDD register of enumerated type

```
type 'a t = {
  typ : string ;
```

Type of the value (refers to the database, see below)

```
  reg : 'a Bdd.Reg.t ;
```

Value itself

```
}
type dt = Cudd.Man.d t
type vt = Cudd.Man.v t
```

7.2 Constants and Operation(s)

```
val of_label :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> 'd t
```

Create a register of the type of the label containing the label

```
val is_cst : 'd t -> bool
```

Does the register contain a constant value ?

```
val to_code : 'd t -> int
```

Convert a constant register to its value as a code.

```
val to_label :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'a
```

Convert a constant register to its value as a label.

```
val equal_label :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'a -> 'd Cudd.Bdd.t
```

Under which condition the register is equal to the label ?

```
val equal :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd Cudd.Bdd.t
```

Under which condition the 2 registers are equal ?

```
val ite : 'd Cudd.Bdd.t -> 'd t -> 'd t -> 'd t
```

If-then-else operator. The types of the 2 branches should be the same.

7.3 Decomposition in guarded form

```
val guard_of_label :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'a -> 'd Cudd.Bdd.t
```

Return the guard of the label in the BDD register.

```
val guardlabels :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> ('d Cudd.Bdd.t * 'a) list
```

Return the list `g -> label` represented by the BDD register.

7.4 Evaluation

```
val cofactor : 'd t -> 'd Cudd.Bdd.t -> 'd t
val restrict : 'd t -> 'd Cudd.Bdd.t -> 'd t
val tdrestrict : 'd t -> 'd Cudd.Bdd.t -> 'd t
```

7.5 Printing

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

`print f fmt t` prints the register `t` using the formatter `fmt` and the function `f` to print BDDs indices.

```
val print_minterm :
  (Format.formatter -> 'd Cudd.Bdd.t -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> 'd t -> unit
```

print_minterm f fmt t prints the register t using the formatter fmt and the function f to convert BDDs indices to names.

7.6 Internal functions

```
val size_of_typ :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> int
```

Return the cardinality of a type (the number of its labels)

```
val maxcode_of_typ :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> int
```

Return the maximal integer corresponding to a label belonging to the type. Labels are indeed associated numbers from 0 to this number.

```
val mem_typcode :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> int -> bool
```

Does the integer code some label of the given type ?

```
val labels_of_typ :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> 'a array
```

Return the array of labels defining the type

```
val code_of_label :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> int
```

Return the code associated to the label

```
val label_of_typcode :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> int -> 'a
```

Return the label associated to the given code interpreted as of type the given type.

```
module Minterm :
  sig
    val iter :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'a -> ('a -> unit) -> Bdd.Reg.Minterm.t -> unit
```

Iter the function on all label of the given type contained in the minterm.

```
    val map :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'a -> ('a -> 'f) -> Bdd.Reg.Minterm.t -> 'f list
```

Apply the function to all label of the given type contained in the minterm and return the list of the results.

```
  end
```

```
val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t
```

Permutation (scale `Cudd.Bdd.permute` and `Cudd.Bdd.permute_memo`)

```
val varmap : 'a t -> 'a t
```

Permutation (scale `Cudd.Bdd.varmap`)

```
val vectorcompose : ?memo:Cudd.Memo.t -> 'a Cudd.Bdd.t array -> 'a t -> 'a t
```

Composition (scale `Cudd.Bdd.vectorcompose` and `Cudd.Bdd.vectorcompose_memo`)

```
end
```

Chapter 8

Module Cond: Normalized condition environments (base module)

```
module Cond :
  sig
```

8.1 Datatypes

```
type ('a, 'b, 'c, 'd) t = {
  symbol : 'a Bdd.Env.symbol ;
  compare_cond : 'c -> 'c -> int ;
  negate_cond : 'b -> 'c -> 'c ;
  support_cond : 'b -> 'c -> 'a PSette.t ;
  mutable print_cond : 'b -> Format.formatter -> 'c -> unit ;
  mutable cudd : 'd Cudd.Man.t ;
```

CUDD manager

```
  mutable bddindex0 : int ;
```

First index for conditions

```
  mutable bddsize : int ;
```

Number of indices dedicated to conditions

```
  mutable bddindex : int ;
```

Next free index in BDDs used by `Bdd.Cond.idb_of_cond`[8.5].

```
  mutable bddincr : int ;
  mutable condidb : ('c, int * bool) PDMappe.t ;
```

Two-way association between a condition and a pair of a BDD index and a polarity

```
  mutable supp : 'd Cudd.Bdd.t ;
```

Support of conditions

```
  mutable careset : 'd Cudd.Bdd.t ;
```

Boolean formula indicating which logical combination known as true could be exploited for simplification. For instance, `x>=1 => x>=0`.

```
}
```

8.2 Printing

```
val print : 'b -> Format.formatter -> ('a, 'b, 'c, 'd) t -> unit
```

8.3 Constructors

```
val make :
  symbol:'a Bdd.Env.symbol ->
  compare_cond:('c -> 'c -> int) ->
  negate_cond:('b -> 'c -> 'c) ->
  support_cond:('b -> 'c -> 'a PSette.t) ->
  print_cond:('b -> Format.formatter -> 'c -> unit) ->
  ?bddindex0:int ->
  ?bddsize:int -> 'd Cudd.Man.t -> ('a, 'b, 'c, 'd) t
val copy : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t
```

8.4 Internal functions

```
val permutation : ('a, 'b, 'c, 'd) t -> int array
```

Compute the permutation for normalizing the environment

```
val permute_with : ('a, 'b, 'c, 'd) t -> int array -> unit
```

Apply the given permutation to the environment

```
val normalize_with : ('a, 'b, 'c, 'd) t -> int array
```

Combine the two previous functions, and return the permutation

```
val reduce_with : ('a, 'b, 'c, 'd) t -> 'd Cudd.Bdd.t -> unit
```

Remove from the environment all conditions that do not belong to the given support. Does not perform normalization (so there may be ”holes” in the allocation of indices

```
val clear : ('a, 'b, 'c, 'd) t -> unit
```

Clear all the conditions (results in a normalized environments)

```
val check_normalized : 'b -> ('a, 'b, 'c, 'd) t -> bool
```

8.5 Operations

```
val cond_of_idb : ('a, 'b, 'c, 'd) t -> int * bool -> 'c
val idb_of_cond : 'a -> ('b, 'a, 'c, 'd) t -> 'c -> int * bool
val compute_careset : ('a, 'b, 'c, 'd) t -> normalized:bool -> unit
val is_leq : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t -> bool
val is_eq : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t -> bool
val shift : ('a, 'b, 'c, 'd) t -> int -> ('a, 'b, 'c, 'd) t
val lce : ('a, 'b, 'c, 'd) t ->
  ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t
val permutation12 : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t -> int array
val permutation21 : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) t -> int array
```

8.6 Level 2

```
  type ('a, 'b) value = {
    cond : 'a ;
    val1 : 'b ;
  }
  val make_value : 'a -> 'b -> ('a, 'b) value
  val get_cond : ('a, 'b) value -> 'a
  val get_val1 : ('a, 'b) value -> 'b
  val get_env : ('a, ('b, 'c) Bdd.Env.value) value -> 'b
  val get_val0 : ('a, ('b, 'c) Bdd.Env.value) value -> 'c
end
```

Chapter 9

Module Decompose: Separation of Boolean formula in purely Boolean/conditional parts

```
module Decompose :
  sig
```

```
    type vdd = bool Cudd.Vdd.t
```

```
    val vdd_of_bdd : Cudd.Bdd.vt -> bool Cudd.Vdd.t
```

```
    val bdd_of_vdd : bool Cudd.Vdd.t -> Cudd.Bdd.vt
```

```
    type typ =
      | Bool
      | Cond
      | Other
```

```
    type info = {
      mutable minlevelbool : int ;
      mutable maxlevelbool : int ;
      mutable minlevelcond : int ;
      mutable maxlevelcond : int ;
      varlevel : int array ;
      levelvar : int array ;
      vartyp : typ array ;
      leveltyp : typ array ;
    }
```

```
    val make_info :
      ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
      ('f, 'g, 'h, 'i) Bdd.Cond.t -> info
```

Builds a temporary record of type `info` which gathers various informations on environment and condition.

```
    val split_level : Cudd.Bdd.vt -> int -> (Cudd.Bdd.vt * Cudd.Bdd.vt) list
```

Decompose a BDD `f` into a disjunction `[(f1,g1);...;(fN,gN)]` such that

- all decisions in f1...fN have levels strictly less than `level`;
- all decisions in g1...gN have levels greater than or equal to `level`;
- f1...fN are pairwise disjoint.

```
    val splitpermutation_of_envcond :
```

```
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, 'i) Bdd.Cond.t ->
  [ `BoolCond | `CondBool ] -> int * (int array * int array) option
```

Two cases in `(level,operm)=splitpermutation_of_envcond ...`:

- `operm=None`: a BDD should be split according to `level`;
- `operm=Some(perm1,perm2)`: a BDD should be split by applying permutation `perm1`, splitting the result according to `level`, and applying to the resulting BDDs the inverse permutation `perm2`.

```
val split_bdd :
  ?memo1:Cudd.Memo.t ->
  ?memo2:Cudd.Memo.t ->
  int * (int array * int array) option ->
  Cudd.Bdd.vt -> (Cudd.Bdd.vt * Cudd.Bdd.vt) list
val cube_split :
  ('a, 'b, 'c, 'd) Bdd.Cond.t -> 'd Cudd.Bdd.t -> 'd Cudd.Bdd.t * 'd Cudd.Bdd.t
```

Split a cube into a cube of Booleans and a cube of conditions

```
val decompose_bdd_boolcond :
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, 'i) Bdd.Cond.t ->
  Cudd.Bdd.vt -> (Cudd.Bdd.vt * Cudd.Bdd.vt) list
```

Decompose a BDD `f` into a disjunction `[(f1,g1);...;(fN,gN)]` such that

- all decisions in f1...fN are Boolean variables;
- all decisions in g1...gN are (non-Boolean) conditions;
- f1...fN are pairwise disjoint.

```
val decompose_bdd_condbool :
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, 'i) Bdd.Cond.t ->
  Cudd.Bdd.vt -> (Cudd.Bdd.vt * Cudd.Bdd.vt) list
```

Dual version

```
val decompose_dd_treecondbool :
  ?careset:'a Cudd.Bdd.t ->
  topvar:('b -> int) ->
  support:('b -> 'c Cudd.Bdd.t) ->
  cofactor:('b -> 'a Cudd.Bdd.t -> 'b) ->
  ('d, 'e, 'f, 'g, 'h) Bdd.Env.t0 ->
  ('i, 'j, 'k, 'a) Bdd.Cond.t -> 'b -> (int, 'b) Bdd.Normalform.tree
```

Internal use, look at the code. Be cautious: support is supposed to return a support intersected with conditions

```
val decompose_bdd_treecondbool :
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, 'i) Bdd.Cond.t ->
  'i Cudd.Bdd.t -> (int, 'i Cudd.Bdd.t) Bdd.Normalform.tree
val decompose_vdd_treecondbool :
  ?careset:Cudd.Bdd.vt ->
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, Cudd.Man.v) Bdd.Cond.t ->
  'i Cudd.Vdd.t -> (int, 'i Cudd.Vdd.t) Bdd.Normalform.tree
```

Decompose a BDD/MTBDD into a tree with decisions on conditions, and purely Boolean BDDs on leaves

```
val decompose_tbdd_tvdd_treecondbool :
  ?careset:Cudd.Bdd.vt ->
  ('a, 'b, 'c, 'd, 'e) Bdd.Env.t0 ->
  ('f, 'g, 'h, Cudd.Man.v) Bdd.Cond.t ->
  Cudd.Bdd.vt array * 'i Cudd.Vdd.t array ->
  (int, Cudd.Bdd.vt array * 'i Cudd.Vdd.t array) Bdd.Normalform.tree
```

Same for an array of BDDs/VDDs

```
val conjunction_of_minterm :
  ?first:int ->
  ?last:int ->
  (int * bool -> 'a) -> Cudd.Man.tbool array -> 'a Bdd.Normalform.conjunction
```

`conjunction_of_minterm of_idb minterm` translates a minterm into an explicit conjunction of type `'a Normalform.conjunction`, using the function `of_idb` to produce elements of type `'a`.

The optional arguments `first` and `last` allows focusing only on a part of the minterm.

```
val dnf_of_bdd :
  ?first:int ->
  ?last:int -> (int * bool -> 'a) -> 'b Cudd.Bdd.t -> 'a Bdd.Normalform.dnf
```

Converts a BDD into a disjunctive normal form. The arguments are the same as in `Bdd.Decompose.conjunction_of_minterm`[9].

```
val descend :
  cudd:'c Cudd.Man.t ->
  maxdepth:int ->
  nocare:('a -> bool) ->
  cube_of_down:('a -> 'c Cudd.Bdd.t) ->
  cofactor:('a -> 'c Cudd.Bdd.t -> 'a) ->
  select:('a -> int) ->
  terminal:(depth:int ->
            newcube:'c Cudd.Bdd.t -> cube:'c Cudd.Bdd.t -> down:'a -> 'b option) ->
  ite:(depth:int ->
       newcube:'c Cudd.Bdd.t ->
       cond:int -> dthen:'b option -> delse:'b option -> 'b option) ->
  down:'a -> 'b option
val select_cond : 'd Cudd.Bdd.t -> int
val select_cond_bdd : ('a, 'b, 'c, 'd) Bdd.Cond.t -> 'd Cudd.Bdd.t -> int
val bdd_support_cond :
  ('a, 'b, 'c, 'd) Bdd.Cond.t -> 'd Cudd.Bdd.t -> 'd Cudd.Bdd.t
val vdd_support_cond :
  ('a, 'b, 'c, Cudd.Man.v) Bdd.Cond.t -> 'd Cudd.Vdd.t -> Cudd.Bdd.vt
val tbdd_tvdd_support_cond :
  ('a, 'b, 'c, Cudd.Man.v) Bdd.Cond.t ->
  Cudd.Bdd.vt array * 'd Cudd.Vdd.t array -> Cudd.Bdd.vt
val tbdd_tvdd_cofactor :
  Cudd.Bdd.vt array * 'a Cudd.Vdd.t array ->
  Cudd.Bdd.vt -> Cudd.Bdd.vt array * 'a Cudd.Vdd.t array
end
```

Chapter 10

Module Expr0: Finite-type expressions with BDDs

```
module Expr0 :
  sig
```

This module allows to manipulate structured BDDs, where variables involved in the Boolean formula are not only Boolean variables, but also of bounded integer or enumerated type (such types are encoded with several Boolean variables).

10.1 Expressions

```
type 'a t = [ `Benum of 'a Bdd.Enum.t | `Bint of 'a Bdd.Int.t | `Bool of 'a Cudd.Bdd.t ]
type 'a expr = 'a t
```

Type of general expressions

```
type dt = Cudd.Man.d t
type vt = Cudd.Man.v t
```

General expressions are described below, after Boolean, bounded integer and enumerated types expressions

10.1.1 Boolean expressions

```
module Bool :
  sig
    type 'a t = 'a Cudd.Bdd.t
    type dt = Cudd.Man.d t
    type vt = Cudd.Man.v t
    val of_expr : 'a Bdd.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bdd.Expr0.expr
    val dtrue : ('a, 'b) Bdd.Env.t -> 'b t
    val dfalse : ('a, 'b) Bdd.Env.t -> 'b t
    val of_bool : ('a, 'b) Bdd.Env.t -> bool -> 'b t
```

```
val var : ('a, 'b) Bdd.Env.t -> 'a -> 'b t
```

```
val dnot : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t
```

```
val dand : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

```
val dor : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

`not`, `and` and `or` (use of 'd' prefix because of conflict with OCaml keywords)

```
val xor : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

```
val nand : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

```
val nor : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

```
val nxor : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

Exclusive or, not and, nor or and not xor

```
val leq : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

Implication

```
val eq : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
```

Same as `nxor`

```
val ite : ('a, 'b) Bdd.Env.t ->
  'b t ->
  'b t -> 'b t -> 'b t
```

If-then-else

```
val is_true : ('a, 'b) Bdd.Env.t -> 'b t -> bool
```

```
val is_false : ('a, 'b) Bdd.Env.t -> 'b t -> bool
```

```
val is_cst : ('a, 'b) Bdd.Env.t -> 'b t -> bool
```

```
val is_eq : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> bool
```

```
val is_leq : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> bool
```

```
val is_and_false : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> bool
```

```
val exist : ('a, 'b) Bdd.Env.t -> 'a list -> 'b t -> 'b t
```

```
val forall : ('a, 'b) Bdd.Env.t -> 'a list -> 'b t -> 'b t
```

```
val cofactor : 'a t -> 'a t -> 'a t
```

```
val restrict : 'a t -> 'a t -> 'a t
```

```
val tdrestrict : 'a t -> 'a t -> 'a t
```

```
val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t
```

```
val varmap : 'a t -> 'a t
```

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t ->
  'b t -> ('a * 'a) list -> 'b t
```

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t ->
```

```
    'b t -> ('a * 'b Bdd.Expr0.expr) list -> 'b t
  val print :
    ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
    ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b t -> unit
end
```

10.1.2 Bounded integer expressions

```
module Bint :
  sig
    type 'a t = 'a Bdd.Int.t
    type dt = Cudd.Man.d t
    type vt = Cudd.Man.v t
    val of_expr : 'a Bdd.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bdd.Expr0.expr
    val of_int : ('a, 'b) Bdd.Env.t -> [ `Bint of bool * int ] -> int -> 'b t
    val var : ('a, 'b) Bdd.Env.t -> 'a -> 'b t
    val ite :
      ('a, 'b) Bdd.Env.t ->
      'b Bdd.Expr0.Bool.t ->
      'b t -> 'b t -> 'b t
    val neg : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t
    val succ : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t
    val pred : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t
    val add : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b t
    val sub : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b t
    val mul : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b t
    val shift_left : ('a, 'b) Bdd.Env.t -> int -> 'b t -> 'b t
    val shift_right : ('a, 'b) Bdd.Env.t -> int -> 'b t -> 'b t
    val scale : ('a, 'b) Bdd.Env.t -> int -> 'b t -> 'b t
    val zero : ('a, 'b) Bdd.Env.t -> 'b t -> 'b Bdd.Expr0.Bool.t
    val eq : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b Bdd.Expr0.Bool.t
    val eq_int : ('a, 'b) Bdd.Env.t -> 'b t -> int -> 'b Bdd.Expr0.Bool.t
    val supeq : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b Bdd.Expr0.Bool.t
    val supeq_int : ('a, 'b) Bdd.Env.t -> 'b t -> int -> 'b Bdd.Expr0.Bool.t
    val sup : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b Bdd.Expr0.Bool.t
    val sup_int : ('a, 'b) Bdd.Env.t -> 'b t -> int -> 'b Bdd.Expr0.Bool.t
    val cofactor : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val restrict : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val tdrestrict : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t
```

```
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, 'b) Bdd.Env.t ->
      'b t -> ('a * 'a) list -> 'b t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, 'b) Bdd.Env.t ->
      'b t -> ('a * 'b Bdd.Expr0.expr) list -> 'b t
    val guard_of_int : ('a, 'b) Bdd.Env.t -> 'b t -> int -> 'b Bdd.Expr0.Bool.t
```

Return the guard of the integer value.

```
    val guardints :
      ('a, 'b) Bdd.Env.t -> 'b t -> ('b Bdd.Expr0.Bool.t * int) list
```

Return the list `g -> n` of guarded values.

```
    val print :
      ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
      ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b t -> unit
  end
```

10.1.3 Enumerated expressions

```
module Benum :
  sig
    type 'a t = 'a Bdd.Enum.t
    type dt = Cudd.Man.d t
    type vt = Cudd.Man.v t
    val of_expr : 'a Bdd.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bdd.Expr0.expr
    val var : ('a, 'b) Bdd.Env.t -> 'a -> 'b t
    val ite :
      ('a, 'b) Bdd.Env.t ->
      'b Bdd.Expr0.Bool.t ->
      'b t -> 'b t -> 'b t
    val eq : ('a, 'b) Bdd.Env.t ->
      'b t -> 'b t -> 'b Bdd.Expr0.Bool.t
    val eq_label : ('a, 'b) Bdd.Env.t -> 'b t -> 'a -> 'b Bdd.Expr0.Bool.t
    val cofactor : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val restrict : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val tdrestrict : 'a t -> 'a Bdd.Expr0.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, 'b) Bdd.Env.t ->
      'b t -> ('a * 'a) list -> 'b t
    val substitute :
      ?memo:Cudd.Memo.t ->
```

```
      ('a, 'b) Bdd.Env.t ->
      'b t -> ('a * 'b Bdd.Expr0.expr) list -> 'b t
```

```
    val guard_of_label : ('a, 'b) Bdd.Env.t -> 'b t -> 'a -> 'b Bdd.Expr0.Bool.t
```

Return the guard of the label.

```
    val guardlabels :
      ('a, 'b) Bdd.Env.t -> 'b t -> ('b Bdd.Expr0.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

```
    val print :
      ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
      ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b t -> unit
  end
```

10.1.4 General (typed) expressions

The following operations raise a `Failure` exception in case of a typing error.

```
val typ_of_expr : ('a, 'b) Bdd.Env.t -> 'b t -> 'a Bdd.Env.typ
```

Type of an expression

```
val var : ('a, 'b) Bdd.Env.t -> 'a -> 'b t
```

Expression representing the litteral var

```
val ite : 'a Bool.t -> 'a t -> 'a t -> 'a t
```

If-then-else operation

```
val eq : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> 'b Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t -> 'b t -> ('a * 'a) list -> 'b t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t ->
  'b t list -> ('a * 'a) list -> 'b t list
```

Variable renaming.
The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t ->
  'b t -> ('a * 'b t) list -> 'b t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) Bdd.Env.t ->
  'b t list -> ('a * 'b t) list -> 'b t list
```

Parallel substitution of variables by expressions

```
val cofactor : 'a t -> 'a Cudd.Bdd.t -> 'a t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val restrict : 'a t -> 'a Cudd.Bdd.t -> 'a t
val tdrestrict : 'a t -> 'a Cudd.Bdd.t -> 'a t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val support : ('a, 'b) Bdd.Env.t -> 'b t -> 'a PSette.t
```

Support of the expression

```
val support_cond : 'a Cudd.Man.t -> 'a t -> 'a Cudd.Bdd.t
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

10.1.4.1 Miscellaneous

```
val cube_of_bdd : ('a, 'b) Bdd.Env.t -> 'b Cudd.Bdd.t -> 'b Cudd.Bdd.t
```

Same as `Cudd.Bdd.cube_of_bdd`, but keep only the the values of variables having a determinated value.

Example: the classical `Cudd.Bdd.cube_of_bdd` could return `b and (x=1 or x=3)`, whereas `cube_of_bdd` will return only `b` in such a case.

10.1.4.2 Printing

```
val print :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b t -> unit
```

Print an expression

```
val print_minterm :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> Cudd.Man.tbool array -> unit
```

Print a minterm

```
val print_bdd :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b Cudd.Bdd.t -> unit
```

Print a BDD

```
val print_idcondb :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> int * bool -> unit
```

Print the condition represented by the signed BDD index.

```
val print_idcond :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> int -> unit
```

Print the condition

10.2 Opened signature and Internal functions

We provide here the same functions and modules as before, but with opened types (this allows etxensions). The functions above are axtually derived from the functions below by just constraining their types. We provide here also more internal functions

```
module O :
  sig
    val tid_of_var :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'a -> int array
    val reg_of_expr : 'd Bdd.Expr0.expr -> 'd Cudd.Bdd.t array
```

10.2.1 Expressions

10.2.1.1 Boolean expressions

```
    module Bool :
      sig
        type 'd t = 'd Cudd.Bdd.t
        type dt = Cudd.Man.d t
        type vt = Cudd.Man.v t
        val of_expr : [> `Bool of 'd t ] -> 'd t
        val to_expr : 'd t -> [> `Bool of 'd t ]
        val dtrue :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t
        val dfalse :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t
        val of_bool :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          bool -> 'd t
        val var :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'a -> 'd t
        val dnot :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t -> 'd t
        val dand :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t -> 'd t -> 'd t
        val dor :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t -> 'd t -> 'd t
```

`not`, `and` and `or` (use of 'd' prefix because of conflict with OCaml keywords)

```
        val xor :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
          'd t -> 'd t -> 'd t
        val nand :
          ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
```

```
  'd t -> 'd t -> 'd t
val nor :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd t
val nxor :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd t
```

Exclusive or, not and, nor or and not xor

```
val leq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd t
```

Implication

```
val eq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd t
```

Same as `nxor`

```
val ite :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t ->
  'd t -> 'd t -> 'd t
```

If-then-else

```
val is_true :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> bool
val is_false :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> bool
val is_cst :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> bool
val is_eq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> bool
val is_leq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> bool
val is_and_false :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> bool
val exist :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a list -> 'd t -> 'd t
val forall :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a list -> 'd t -> 'd t
val cofactor : 'd t -> 'd t -> 'd t
val restrict : 'd t -> 'd t -> 'd t
val tdrestrict : 'd t -> 'd t -> 'd t
val permute : ?memo:Cudd.Memo.t ->
  'd t -> int array -> 'd t
```

```
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> ('a * 'a) list -> 'd t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t ->
      ('a * 'd Bdd.Expr0.expr) list -> 'd t
    val print :
      ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      Format.formatter -> 'd t -> unit
  end
```

10.2.1.2 Bounded integer expressions

```
module Bint :
  sig
    type 'd t = 'd Bdd.Int.t
    type dt = Cudd.Man.d t
    type vt = Cudd.Man.v t
    val of_expr : [> `Bint of 'd t ] -> 'd t
    val to_expr : 'd t -> [> `Bint of 'd t ]
    val of_int :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      [> `Bint of bool * int ] -> int -> 'd t
    val var :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'a -> 'd t
    val ite :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Expr0.O.Bool.t ->
      'd t -> 'd t -> 'd t
    val neg :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t
    val succ :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t
    val pred :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t
    val add :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t -> 'd t
    val sub :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t -> 'd t
    val mul :
```

```
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd t
val shift_left :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  int -> 'd t -> 'd t
val shift_right :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  int -> 'd t -> 'd t
val scale :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  int -> 'd t -> 'd t
val zero :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd Bdd.Expr0.O.Bool.t
val eq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd Bdd.Expr0.O.Bool.t
val eq_int :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> int -> 'd Bdd.Expr0.O.Bool.t
val supeq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd Bdd.Expr0.O.Bool.t
val supeq_int :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> int -> 'd Bdd.Expr0.O.Bool.t
val sup :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> 'd t -> 'd Bdd.Expr0.O.Bool.t
val sup_int :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> int -> 'd Bdd.Expr0.O.Bool.t
val cofactor : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
val restrict : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
val tdrestrict : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
val permute : ?memo:Cudd.Memo.t ->
  'd t -> int array -> 'd t
val varmap : 'a t -> 'a t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> ('a * 'a) list -> 'd t
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t ->
  ('a * 'd Bdd.Expr0.expr) list -> 'd t
val guard_of_int :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> int -> 'd Bdd.Expr0.O.Bool.t
```

Return the guard of the integer value.

```
    val guardints :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> ('d Bdd.Expr0.O.Bool.t * int) list
```

Return the list `g -> n` of guarded values.

```
    val print :
      ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      Format.formatter -> 'd t -> unit
  end
```

10.2.1.3 Enumerated expressions

```
module Benum :
  sig
    type 'd t = 'd Bdd.Enum.t
    type dt = Cudd.Man.d t
    type vt = Cudd.Man.v t
    val of_expr : [> `Benum of 'd t ] -> 'd t
    val to_expr : 'd t -> [> `Benum of 'd t ]
    val var :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'a -> 'd t
    val ite :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Expr0.O.Bool.t ->
      'd t -> 'd t -> 'd t
    val eq :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'd t -> 'd Bdd.Expr0.O.Bool.t
    val eq_label :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'a -> 'd Bdd.Expr0.O.Bool.t
    val cofactor : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
    val restrict : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
    val tdrestrict : 'd t -> 'd Bdd.Expr0.O.Bool.t -> 'd t
    val permute : ?memo:Cudd.Memo.t ->
      'd t -> int array -> 'd t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> ('a * 'a) list -> 'd t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t ->
      ('a * 'd Bdd.Expr0.expr) list -> 'd t
    val guard_of_label :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd t -> 'a -> 'd Bdd.Expr0.O.Bool.t
```

Return the guard of the label.

```
val guardlabels :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd t -> ('d Bdd.Expr0.O.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

```
val print :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> 'd t -> unit
end
```

10.2.1.4 General (typed) expressions

The following operations raise a `Failure` exception in case of a typing error.

```
val typ_of_expr :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t -> [> 'a Bdd.Env.typ ]
```

Type of an expression

```
val var :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a -> 'd Bdd.Expr0.t
```

Expression representing the litteral var

```
val ite : 'd Bool.t -> 'd Bdd.Expr0.t -> 'd Bdd.Expr0.t -> 'd Bdd.Expr0.t
```

If-then-else operation

```
val eq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t -> 'd Bdd.Expr0.t -> 'd Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t -> ('a * 'a) list -> 'd Bdd.Expr0.t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t list -> ('a * 'a) list -> 'd Bdd.Expr0.t list
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t -> ('a * 'd Bdd.Expr0.t) list -> 'd Bdd.Expr0.t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t list -> ('a * 'd Bdd.Expr0.t) list -> 'd Bdd.Expr0.t list
```

Parallel substitution of variables by expressions

```
val support :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Expr0.t -> 'a PSette.t
```

Support of the expression

```
val support_cond : 'd Cudd.Man.t -> 'd Bdd.Expr0.t -> 'd Cudd.Bdd.t
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

10.2.1.5 Miscellaneous

```
val cube_of_bdd :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Cudd.Bdd.t -> 'd Cudd.Bdd.t
```

Same as `Cudd.Bdd.cube_of_bdd`, but keep only the the values of variables having a determinated value.
Example: the classical `Cudd.Bdd.cube_of_bdd` could return `b and (x=1 or x=3)`, whereas `cube_of_bdd` will return only `b` in such a case.

```
val tbdd_of_texpr : 'd Bdd.Expr0.t array -> 'd Cudd.Bdd.t array
```

Concatenates in an array the BDDs involved in the expressions

```
val texpr_of_tbdd :
  'd Bdd.Expr0.t array -> 'd Cudd.Bdd.t array -> 'd Bdd.Expr0.t array
```

Inverse operation: rebuild an array of expressions from the old array of expressions (for the types) and the array of BDDs.

10.2.2 Printing

```
val print :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> [< 'd Bdd.Expr0.t ] -> unit
```

Print an expression

```
val print_minterm :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> Cudd.Man.tbool array -> unit
```

Print a minterm

```
val print_bdd :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> 'd Cudd.Bdd.t -> unit
```

Print a BDD

```
val print_idcondb :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> int * bool -> unit
```

Print the condition represented by the signed BDD index.

```
val print_idcond :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  Format.formatter -> int -> unit
```

Print the condition

10.2.3 Internal functions

10.2.3.1 Permutation and composition

```
val permutation_of_rename :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  ('a * 'a) list -> int array
val composition_of_lvarexpr :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  ('a * 'd Bdd.Expr0.t) list -> 'd Cudd.Bdd.t array
val composition_of_lvarlexpr :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a list -> 'd Bdd.Expr0.t list -> 'd Cudd.Bdd.t array
val bddsupport :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a list -> 'd Cudd.Bdd.t
val varmap : 'a Bdd.Expr0.t -> 'a Bdd.Expr0.t
val permute :
  ?memo:Cudd.Memo.t -> 'd Bdd.Expr0.t -> int array -> 'd Bdd.Expr0.t
val compose :
  ?memo:Cudd.Memo.t -> 'd Bdd.Expr0.t -> 'd Cudd.Bdd.t array -> 'd Bdd.Expr0.t
val permute_list :
  ?memo:Cudd.Memo.t -> 'd Bdd.Expr0.t list -> int array -> 'd Bdd.Expr0.t list
val compose_list :
  ?memo:Cudd.Memo.t ->
  'd Bdd.Expr0.t list -> 'd Cudd.Bdd.t array -> 'd Bdd.Expr0.t list
```

10.2.4 Conversion to expressions

```
module Expr :
  sig
```

Syntax tree for printing

```
    type 'a atom =
      | Tbool of 'a * bool
```

variable name and sign

```
      | Tint of 'a * int list
```

variable name and list of possible values

```
      | Tenum of 'a * 'a list
```

variable name, possibly primed, and list of possible labels

Atom

```
    type 'a term =
      | Tatom of 'a atom
      | Texternal of (int * bool)
```

Unregistered BDD identifier and a Boolean for possible negation

```
      | Tcst of bool
```

Boolean constant

Basic term

```
    val map_atom : ('a -> 'b) -> 'a atom -> 'b atom
    val map_term : ('a -> 'b) -> 'a term -> 'b term
    val term_of_vint : 'a -> 'd Bdd.Int.t -> Bdd.Reg.Minterm.t -> 'a term
```

```
      val term_of_venum :
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        'a -> 'd Bdd.Enum.t -> Bdd.Reg.Minterm.t -> 'a term
      val term_of_idcondb :
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        int * bool -> 'a term
      val bool_of_tbool : Cudd.Man.tbool -> bool
      val mand : 'a term list Pervasives.ref ->
        'a term -> unit
      val conjunction_of_minterm :
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        Cudd.Man.tbool array -> 'a term Bdd.Normalform.conjunction
      val dnf_of_bdd :
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        'd Cudd.Bdd.t -> 'a term Bdd.Normalform.dnf
      val print_term :
        ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        Format.formatter -> 'a term -> unit
      val print_conjunction :
        ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        Format.formatter ->
        'a term Bdd.Normalform.conjunction -> unit
      val print_dnf :
        ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
        ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
        Format.formatter -> 'a term Bdd.Normalform.dnf -> unit
    end

  end

end
```

Chapter 11

Module Expr1: Finite-type expressions with normalized environments

```
module Expr1 :
  sig
```

11.1 Expressions

```
type ('a, 'b) t = (('a, 'b) Bdd.Env.t, 'b Bdd.Expr0.t) Bdd.Env.value
type ('a, 'b) expr = ('a, 'b) t
```

Type of general expressions

```
type 'a dt = ('a, Cudd.Man.d) t
type 'a vt = ('a, Cudd.Man.v) t
```

General expressions are described below, after Boolean, bounded integer and enumerated types expressions

11.1.1 Boolean expressions

```
module Bool :
  sig
    type ('a, 'b) t = (('a, 'b) Bdd.Env.t, 'b Cudd.Bdd.t) Bdd.Env.value
    type 'a dt = ('a, Cudd.Man.d) t
    type 'a vt = ('a, Cudd.Man.v) t
    val of_expr0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.Bool.t -> ('a, 'b) t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
    val to_expr0 : ('a, 'b) t -> 'b Bdd.Expr0.Bool.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : ('a, 'b) Bdd.Expr1.expr -> ('a, 'b) t
    val to_expr : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.expr
```

Conversion from/to general expression

```
val extend_environment : ('a, 'b) t -> ('a, 'b) Bdd.Env.t -> ('a, 'b) t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val dtrue : ('a, 'b) Bdd.Env.t -> ('a, 'b) t
val dfalse : ('a, 'b) Bdd.Env.t -> ('a, 'b) t
val of_bool : ('a, 'b) Bdd.Env.t -> bool -> ('a, 'b) t
val var : ('a, 'b) Bdd.Env.t -> 'a -> ('a, 'b) t
```

11.1.1.1 Logical connectors

```
val dnot : ('a, 'b) t -> ('a, 'b) t
val dand : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
val dor : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
```

`not`, `and` and `or` (use of 'd' prefix because of conflict with OCaml keywords)

```
val xor : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
val nand : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
val nor : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
val nxor : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
```

Exclusive or, not and, nor or and not xor

```
val eq : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
```

Same as `nxor`

```
val leq : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
```

Implication

```
val ite : ('a, 'b) t ->
  ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
```

If-then-else

```
val is_true : ('a, 'b) t -> bool
val is_false : ('a, 'b) t -> bool
val is_cst : ('a, 'b) t -> bool
val is_eq : ('a, 'b) t -> ('a, 'b) t -> bool
val is_leq : ('a, 'b) t -> ('a, 'b) t -> bool
val is_inter_false : ('a, 'b) t -> ('a, 'b) t -> bool
val exist : 'a list -> ('a, 'b) t -> ('a, 'b) t
val forall : 'a list -> ('a, 'b) t -> ('a, 'b) t
val cofactor : ('a, 'b) t ->
```

```
    ('a, 'b) t -> ('a, 'b) t
  val restrict : ('a, 'b) t ->
    ('a, 'b) t -> ('a, 'b) t
  val tdrestrict : ('a, 'b) t ->
    ('a, 'b) t -> ('a, 'b) t
  val substitute_by_var :
    ?memo:Cudd.Memo.t ->
    ('a, 'b) t -> ('a * 'a) list -> ('a, 'b) t
  val substitute :
    ?memo:Cudd.Memo.t ->
    ('a, 'b) t ->
    ('a * ('a, 'b) Bdd.Expr1.expr) list -> ('a, 'b) t
  val print : Format.formatter -> ('a, 'b) t -> unit
end
```

11.1.2 Bounded integer expressions

```
module Bint :
  sig
    type ('a, 'b) t = (('a, 'b) Bdd.Env.t, 'b Bdd.Int.t) Bdd.Env.value
    type 'a dt = ('a, Cudd.Man.d) t
    type 'a vt = ('a, Cudd.Man.v) t
    val of_expr0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.Bint.t -> ('a, 'b) t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
    val to_expr0 : ('a, 'b) t -> 'b Bdd.Expr0.Bint.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : ('a, 'b) Bdd.Expr1.expr -> ('a, 'b) t
    val to_expr : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.expr
```

Conversion from/to general expression

```
    val extend_environment : ('a, 'b) t -> ('a, 'b) Bdd.Env.t -> ('a, 'b) t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
    val of_int :
      ('a, 'b) Bdd.Env.t ->
      [ `Bint of bool * int ] -> int -> ('a, 'b) t
    val var : ('a, 'b) Bdd.Env.t -> 'a -> ('a, 'b) t
    val neg : ('a, 'b) t -> ('a, 'b) t
    val succ : ('a, 'b) t -> ('a, 'b) t
    val pred : ('a, 'b) t -> ('a, 'b) t
    val add : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) t
    val sub : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) t
    val mul : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) t
```

```
    val shift_left : int -> ('a, 'b) t -> ('a, 'b) t
    val shift_right : int -> ('a, 'b) t -> ('a, 'b) t
    val scale : int -> ('a, 'b) t -> ('a, 'b) t
    val ite :
      ('a, 'b) Bdd.Expr1.Bool.t ->
      ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) t
    val zero : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
    val eq : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
    val supeq : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
    val sup : ('a, 'b) t ->
      ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
    val eq_int : ('a, 'b) t -> int -> ('a, 'b) Bdd.Expr1.Bool.t
    val supeq_int : ('a, 'b) t -> int -> ('a, 'b) Bdd.Expr1.Bool.t
    val sup_int : ('a, 'b) t -> int -> ('a, 'b) Bdd.Expr1.Bool.t
    val cofactor : ('a, 'b) t ->
      ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
    val restrict : ('a, 'b) t ->
      ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
    val tdrestrict : ('a, 'b) t ->
      ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, 'b) t -> ('a * 'a) list -> ('a, 'b) t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, 'b) t ->
      ('a * ('a, 'b) Bdd.Expr1.expr) list -> ('a, 'b) t
    val guard_of_int : ('a, 'b) t -> int -> ('a, 'b) Bdd.Expr1.Bool.t
```

Return the guard of the integer value.

```
    val guardints : ('a, 'b) t -> (('a, 'b) Bdd.Expr1.Bool.t * int) list
```

Return the list `g -> n` of guarded values.

```
    val print : Format.formatter -> ('a, 'b) t -> unit
  end
```

11.1.3 Enumerated expressions

```
module Benum :
  sig
    type ('a, 'b) t = (('a, 'b) Bdd.Env.t, 'b Bdd.Enum.t) Bdd.Env.value
    type 'a dt = ('a, Cudd.Man.d) t
    type 'a vt = ('a, Cudd.Man.v) t
    val of_expr0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.Benum.t -> ('a, 'b) t
```

Creation from an expression of level 0 (without environment)

```
val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
val to_expr0 : ('a, 'b) t -> 'b Bdd.Expr0.Benum.t
```

Extract resp. the environment and the underlying expression of level 0

```
val of_expr : ('a, 'b) Bdd.Expr1.expr -> ('a, 'b) t
val to_expr : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.expr
```

Conversion from/to general expression

```
val extend_environment : ('a, 'b) t ->
  ('a, 'b) Bdd.Env.t -> ('a, 'b) t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val var : ('a, 'b) Bdd.Env.t -> 'a -> ('a, 'b) t
val ite :
  ('a, 'b) Bdd.Expr1.Bool.t ->
  ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) t
val eq : ('a, 'b) t ->
  ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
val eq_label : ('a, 'b) t -> 'a -> ('a, 'b) Bdd.Expr1.Bool.t
val cofactor : ('a, 'b) t ->
  ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
val restrict : ('a, 'b) t ->
  ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
val tdrestrict : ('a, 'b) t ->
  ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t -> ('a * 'a) list -> ('a, 'b) t
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t ->
  ('a * ('a, 'b) Bdd.Expr1.expr) list -> ('a, 'b) t
val guard_of_label : ('a, 'b) t -> 'a -> ('a, 'b) Bdd.Expr1.Bool.t
```

Return the guard of the label.

```
val guardlabels : ('a, 'b) t -> (('a, 'b) Bdd.Expr1.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

```
val print : Format.formatter -> ('a, 'b) t -> unit
end
```

11.1.4 General expressions

```
val typ_of_expr : ('a, 'b) t -> 'a Bdd.Env.typ
```

Type of an expression

```
val make : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.t -> ('a, 'b) t
val of_expr0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.t -> ('a, 'b) t
```

Creation from an expression of level 0 (without environment) (make should be considered as obsolete)

```
val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
val to_expr0 : ('a, 'b) t -> 'b Bdd.Expr0.t
```

Extract resp. the environment and the underlying expression of level 0

```
val extend_environment : ('a, 'b) t -> ('a, 'b) Bdd.Env.t -> ('a, 'b) t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val var : ('a, 'b) Bdd.Env.t -> 'a -> ('a, 'b) t
```

Expression representing the litteral var

```
val ite : ('a, 'b) Bool.t ->
  ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

If-then-else operation

```
val eq : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t -> ('a * 'a) list -> ('a, 'b) t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t list -> ('a * 'a) list -> ('a, 'b) t list
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t ->
  ('a * ('a, 'b) t) list -> ('a, 'b) t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a, 'b) t list ->
  ('a * ('a, 'b) t) list -> ('a, 'b) t list
```

Parallel substitution of variables by expressions

```
val support : ('a, 'b) t -> 'a PSette.t
```

Support of the expression

```
val support_cond : ('a, 'b) t -> 'b Cudd.Bdd.t
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

```
val cofactor : ('a, 'b) t -> ('a, 'b) Bool.t -> ('a, 'b) t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val restrict : ('a, 'b) t -> ('a, 'b) Bool.t -> ('a, 'b) t
val tdrestrict : ('a, 'b) t -> ('a, 'b) Bool.t -> ('a, 'b) t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val print : Format.formatter -> ('a, 'b) t -> unit
```

11.1.5 List of expressions

```
module List :
  sig
    type ('a, 'b) t = (('a, 'b) Bdd.Env.t, 'b Bdd.Expr0.t list) Bdd.Env.value
    type 'a dt = ('a, Cudd.Man.d) t
    type 'a vt = ('a, Cudd.Man.v) t
    val of_lexpr0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Expr0.t list -> ('a, 'b) t
    val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
    val to_lexpr0 : ('a, 'b) t -> 'b Bdd.Expr0.t list
    val of_lexpr :
      ('a, 'b) Bdd.Env.t ->
      ('a, 'b) Bdd.Expr1.expr list -> ('a, 'b) t
    val to_lexpr : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.expr list
    val extend_environment : ('a, 'b) t -> ('a, 'b) Bdd.Env.t -> ('a, 'b) t
    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 -> ('a, 'b) t -> unit
  end
```

11.2 Opened signature and Internal functions

We provide here the same functions and modules as before, but with opened types (this allows etxensions). The functions above are actually derived from the functions below by just constraining their types. We provide here also more internal functions

```
module O :
  sig
```

11.2.1 Expressions

```
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'f) Bdd.Env.O.t, 'c) t = (('a, [> '
       Bdd.Env.O.t, 'c Bdd.Expr0.t)
      Bdd.Env.value
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t, 'c) expr = ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'd, [> 'a Bdd.Env.typdef ] as 'e, 'c, 'b)
       Bdd.Env.O.t, 'c)
      t
```

Type of general expressions

```
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t) dt = ('a
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.d,
        'b)
       Bdd.Env.O.t, Cudd.Man.d)
      t
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t) vt = ('a
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.v,
        'b)
       Bdd.Env.O.t, Cudd.Man.v)
      t
```

11.2.1.1 Boolean expressions

```
module Bool :
  sig
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'f) Bdd.Env.O.t, 'c) t = (('a,
       Bdd.Env.O.t, 'c Cudd.Bdd.t)
      Bdd.Env.value
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t) dt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.d,
        'b)
       Bdd.Env.O.t, Cudd.Man.d)
      t
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t) vt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.v,
        'b)
       Bdd.Env.O.t, Cudd.Man.v)
      t
    val of_expr0 :
      ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
      Bdd.Env.O.t ->
      'c Bdd.Expr0.Bool.t ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val get_env :
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
    val to_expr0 :
      ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
       'c)
      t -> 'c Bdd.Expr0.Bool.t
    val of_expr :
      (('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, [> `Bool of 'c Cudd.Bdd.t ])
      Bdd.Env.value ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val to_expr :
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t ->
      (('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, [> `Bool of 'c Cudd.Bdd.t ]) Bdd.Env.value
    val extend_environment :
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t ->
      ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val dtrue :
      ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
      Bdd.Env.O.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val dfalse :
```

```
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val of_bool :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  bool -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val var :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

11.2.1.2 Logical connectors

```
val dnot :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val dand :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val dor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

`not`, `and` and `or` (use of 'd' prefix because of conflict with OCaml keywords)

```
val xor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val nand :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val nor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
```

```
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
val nxor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Exclusive or, not and, nor or and not xor

```
val eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Same as `nxor`

```
val leq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Implication

```
val ite :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

If-then-else

```
val is_true :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> bool
val is_false :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> bool
val is_cst :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> bool
val is_eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t -> bool
```

```
val is_leq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t -> bool
val is_inter_false :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t -> bool
val exist :
  'a list ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val forall :
  'a list ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val cofactor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val restrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t ->
      ('a * ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.expr) list ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val print :
      Format.formatter ->
      ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
       'c)
      t -> unit
  end
```

11.2.1.3 Bounded integer expressions

```
module Bint :
  sig
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'f) Bdd.Env.O.t, 'c) t = (('a,
       Bdd.Env.O.t, 'c Bdd.Int.t)
      Bdd.Env.value
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t, 'c) d
       ('a, [> 'a Bdd.Env.typ ] as 'd, [> 'a Bdd.Env.typdef ] as 'e, Cudd.Man.d,
        'b)
       Bdd.Env.O.t, Cudd.Man.d)
      t
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t, 'c) v
       ('a, [> 'a Bdd.Env.typ ] as 'd, [> 'a Bdd.Env.typdef ] as 'e, Cudd.Man.v,
        'b)
       Bdd.Env.O.t, Cudd.Man.v)
      t
    val of_expr0 :
      ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
      Bdd.Env.O.t ->
      'c Bdd.Expr0.Bint.t ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val get_env :
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
    val to_expr0 :
      ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
       'c)
      t -> 'c Bdd.Expr0.Bint.t
    val of_expr :
      (('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, [> `Bint of 'c Bdd.Int.t ])
      Bdd.Env.value ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val to_expr :
      ('a,
```

```
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  (('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, [> `Bint of 'c Bdd.Int.t ]) Bdd.Env.value
val extend_environment :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val of_int :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  [ `Bint of bool * int ] ->
  int -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val var :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val neg :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val succ :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val pred :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val add :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val sub :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val mul :
  ('a,
```

```
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val shift_left :
  int ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val shift_right :
  int ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val scale :
  int ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val ite :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val zero :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val supeq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val sup :
```

```
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val eq_int :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  int -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val supeq_int :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  int -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val sup_int :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  int -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val cofactor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val restrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute :
```

```
    ?memo:Cudd.Memo.t ->
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    ('a * ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.expr) list ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val guard_of_int :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    int -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
```

Return the guard of the integer value.

```
  val guardints :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    (('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t * int) list
```

Return the list `g -> n` of guarded values.

```
  val print :
    Format.formatter ->
    ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
     'c)
    t -> unit
end
```

11.2.1.4 Enumerated expressions

```
module Benum :
  sig
```

```
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'f) Bdd.Env.O.t, 'c) t = (('a,
       Bdd.Env.O.t, 'c Bdd.Enum.t)
      Bdd.Env.value
```

```
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t) dt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.d,
        'b)
       Bdd.Env.O.t, Cudd.Man.d)
      t
```

```
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t) vt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.v,
        'b)
       Bdd.Env.O.t, Cudd.Man.v)
      t
```

```
    val of_expr0 :
      ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
      Bdd.Env.O.t ->
      'c Bdd.Expr0.Benum.t ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
    val get_env :
      ('a,
```

```
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
val to_expr0 :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'c Bdd.Expr0.Benum.t
val of_expr :
  (('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, [> `Benum of 'c Bdd.Enum.t ])
  Bdd.Env.value ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val to_expr :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  (('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, [> `Benum of 'c Bdd.Enum.t ])
  Bdd.Env.value
val extend_environment :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val var :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val ite :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val eq_label :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val cofactor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
```

```
     Bdd.Env.O.t, 'c)
    t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val restrict :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val tdrestrict :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val substitute_by_var :
    ?memo:Cudd.Memo.t ->
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    ('a * 'a) list ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val substitute :
    ?memo:Cudd.Memo.t ->
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    ('a * ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.expr) list ->
    ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

```
  val guard_of_label :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
```

Return the guard of the label.

```
  val guardlabels :
    ('a,
     ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
     Bdd.Env.O.t, 'c)
    t ->
    (('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

```
  val print :
    Format.formatter ->
    ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
     'c)
    t -> unit
end
```

11.2.1.5 General expressions

```
val typ_of_expr :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'a Bdd.Env.typ
```

Type of an expression

```
val make :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'c Bdd.Expr0.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) expr
val of_expr0 :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'c Bdd.Expr0.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) expr
```

Creation from an expression without environment

```
val get_env :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
val to_expr0 :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'c Bdd.Expr0.t
val extend_environment :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val var :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t -> 'a -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Expression representing the litteral var

```
val ite :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

If-then-else operation

```
val eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a * 'a) list -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t list ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t list
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a * ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t) list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t list ->
  ('a * ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t) list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t list
```

Parallel substitution of variables by expressions

```
val support :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'a PSette.t
```

Support of the expression

```
val support_cond :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'c Cudd.Bdd.t
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

```
val cofactor :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val restrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val print :
  Format.formatter ->
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> unit
```

11.2.1.6 List of expressions

```
module List :
  sig
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'f) Bdd.Env.O.t, 'c) t = (('a,
       Bdd.Env.O.t, 'c Bdd.Expr0.t list)
      Bdd.Env.value
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t) dt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.d,
        'b)
       Bdd.Env.O.t, Cudd.Man.d)
      t
    type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t) vt =
       ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.v,
        'b)
       Bdd.Env.O.t, Cudd.Man.v)
      t
    val of_lexpr0 :
      ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
      Bdd.Env.O.t ->
      'c Bdd.Expr0.t list ->
      ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
    val get_env :
      ('a,
       ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
       Bdd.Env.O.t, 'c)
      t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
    val to_lexpr0 :
      ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
       'c)
```

```
        t -> 'c Bdd.Expr0.t list
      val of_lexpr :
        ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
        Bdd.Env.O.t ->
        ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.expr list ->
        ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
      val to_lexpr :
        ('a,
         ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
         Bdd.Env.O.t, 'c)
        t ->
        ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.expr list
      val extend_environment :
        ('a,
         ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
         Bdd.Env.O.t, 'c)
        t ->
        ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
        ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
      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 ->
        ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
         'c)
        t -> unit
    end

  end

end
```

Chapter 12

Module Domain0: Boolean (abstract) domain

```
module Domain0 :
  sig
```

12.1 Abstract domain

```
type 'a t = 'a Bdd.Expr0.Bool.t
```

Abstract value

```
type dt = Cudd.Man.d t
type vt = Cudd.Man.v t
val size : 'a t -> int
```

Size of an abstract value (number of nodes)

```
val print :
  ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
  ('a, 'b) Bdd.Env.t -> Format.formatter -> 'b t -> unit
val bottom : ('a, 'b) Bdd.Env.t -> 'b t
val top : ('a, 'b) Bdd.Env.t -> 'b t
```

Constructors

```
val is_bottom : ('a, 'b) Bdd.Env.t -> 'b t -> bool
val is_top : ('a, 'b) Bdd.Env.t -> 'b t -> bool
val is_leq : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> bool
val is_eq : ('a, 'b) Bdd.Env.t -> 'b t -> 'b t -> bool
val is_variable_unconstrained : ('a, 'b) Bdd.Env.t -> 'b t -> 'a -> bool
```

Tests

```
val meet : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
val join : ('a, 'b) Bdd.Env.t ->
  'b t -> 'b t -> 'b t
val meet_condition :
  ('a, 'b) Bdd.Env.t ->
  'b t -> 'b Bdd.Expr0.Bool.t -> 'b t
```

Lattice operations

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) Bdd.Env.t ->
  'b t -> 'a list -> 'b Bdd.Expr0.expr list -> 'b t
```

Assignement

If `nodependency=true`, which means that no expression depends on the assigned variables, it uses an optimized algorithm.

If `relational=true`, it is assumed that `env#bddincr=2` (checked), starting from a pair index. It is also advised to have paired variables in groups.

`rel=true` is most probably much better for assignements of a few variables.

```
val substitute_lexpr :
  ('a, 'b) Bdd.Env.t ->
  'b t -> 'a list -> 'b Bdd.Expr0.expr list -> 'b t
```

Substitution

```
val forget_list : ('a, 'b) Bdd.Env.t -> 'b t -> 'a list -> 'b t
```

Eliminating variables

12.2 Opened signature and Internal functions

We provide here the same functions and modules as before, but with opened types (this allows etxensions). The functions above are axtually derived from the functions below by just constraining their types. We provide here also more internal functions

```
module O :
  sig
    val print :
      ?print_external_idcondb:(Format.formatter -> int * bool -> unit) ->
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      Format.formatter -> 'd Bdd.Domain0.t -> unit
    val bottom :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Domain0.t
    val top :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Domain0.t
```

Constructors

```
    val is_bottom :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Domain0.t -> bool
    val is_top :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Domain0.t -> bool
    val is_leq :
      ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
      'd Bdd.Domain0.t -> 'd Bdd.Domain0.t -> bool
```

```
val is_eq :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'd Bdd.Domain0.t -> bool
val is_variable_unconstrained :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'a -> bool
```

Tests

```
val meet :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'd Bdd.Domain0.t -> 'd Bdd.Domain0.t
val join :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'd Bdd.Domain0.t -> 'd Bdd.Domain0.t
val meet_condition :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'd Bdd.Expr0.Bool.t -> 'd Bdd.Domain0.t
```

Lattice operations

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'a list -> 'd Bdd.Expr0.expr list -> 'd Bdd.Domain0.t
```

Assignement

If `nodependency=true`, which means that no expression depends on the assigned variables, it uses an optimized algorithm.

If `rel=true`, it is assumed that `env#bddincr=2` (checked), starting from a pair index. It is also advised to have paired variables in groups.

`rel=true` is most probably much better for assignements of a few variables.

```
val substitute_lexpr :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'a list -> 'd Bdd.Expr0.expr list -> 'd Bdd.Domain0.t
```

Substitution

```
val forget_list :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'd Bdd.Domain0.t -> 'a list -> 'd Bdd.Domain0.t
```

Eliminating variables

```
module Asssub :
  sig
    val sort :
      int array -> 'a Cudd.Bdd.t array -> int array * 'a Cudd.Bdd.t array
    val is_equal : 'a Cudd.Bdd.t array -> 'a Cudd.Bdd.t array -> bool
    val post : 'a Cudd.Bdd.t -> int array -> 'a Cudd.Bdd.t array -
    > 'a Cudd.Bdd.t
    val postcondition : 'a Cudd.Bdd.t -> 'a Cudd.Bdd.t array -> 'a Cudd.Bdd.t
  end
val relation_supp_compose_of_lvarlexpr :
  ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'd, 'e) Bdd.Env.O.t ->
  'a list ->
  'd Bdd.Expr0.expr list -> 'd Cudd.Bdd.t * 'd Cudd.Bdd.t * 'd Cudd.Bdd.t array
val apply_change : 'a Bdd.Domain0.t -> 'a Bdd.Env.change -> 'a Bdd.Domain0.t
```

```
    end

end
```

Chapter 13

Module Domain1: Boolean (abstract) domain with normalized environment

```
module Domain1 :
  sig
```

13.1 Abstract domain

```
type ('a, 'b) t = ('a, 'b) Bdd.Expr1.Bool.t
```

Abstract value

```
type 'a dt = ('a, Cudd.Man.d) t
type 'a vt = ('a, Cudd.Man.v) t
val of_domain0 : ('a, 'b) Bdd.Env.t -> 'b Bdd.Domain0.t -> ('a, 'b) t
val get_env : ('a, 'b) t -> ('a, 'b) Bdd.Env.t
val to_domain0 : ('a, 'b) t -> 'b Bdd.Domain0.t
```

Conversion operations

```
val of_expr1 : ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
val to_expr1 : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t
```

Conversion operations

```
val size : ('a, 'b) t -> int
```

Size of an abstract value (number of nodes)

```
val print : Format.formatter -> ('a, 'b) t -> unit
val bottom : ('a, 'b) Bdd.Env.t -> ('a, 'b) t
val top : ('a, 'b) Bdd.Env.t -> ('a, 'b) t
```

Constructors

```
val is_bottom : ('a, 'b) t -> bool
val is_top : ('a, 'b) t -> bool
val is_leq : ('a, 'b) t -> ('a, 'b) t -> bool
val is_eq : ('a, 'b) t -> ('a, 'b) t -> bool
val is_variable_unconstrained : ('a, 'b) t -> 'a -> bool
```

Tests

```
val meet : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val join : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val meet_condition : ('a, 'b) t -> ('a, 'b) Bdd.Expr1.Bool.t -> ('a, 'b) t
```

Lattice operations

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) t ->
  'a list -> ('a, 'b) Bdd.Expr1.t list -> ('a, 'b) t
val assign_listexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) t ->
  'a list -> ('a, 'b) Bdd.Expr1.List.t -> ('a, 'b) t
```

Assignement

If `nodependency=true`, which means that no expression depends on the assigned variables, it uses an optimized algorithm.

If `rel=true`, it is assumed that `env#bddincr=2` (checked), starting from a pair index. It is also advised to have paired variables in groups.

`rel=true` is most probably much better for assignements of a few variables.

```
val substitute_lexpr :
  ('a, 'b) t ->
  'a list -> ('a, 'b) Bdd.Expr1.t list -> ('a, 'b) t
val substitute_listexpr :
  ('a, 'b) t ->
  'a list -> ('a, 'b) Bdd.Expr1.List.t -> ('a, 'b) t
```

Substitution

```
val forget_list : ('a, 'b) t -> 'a list -> ('a, 'b) t
```

Eliminating variables

```
val change_environment : ('a, 'b) t -> ('a, 'b) Bdd.Env.t -> ('a, 'b) t
val rename : ('a, 'b) t -> ('a * 'a) list -> ('a, 'b) t
```

Change of environments

13.2 Opened signature and Internal functions

We provide here the same functions and modules as before, but with opened types (this allows etxensions). The functions above are axtually derived from the functions below by just constraining their types. We provide here also more internal functions

```
module O :
  sig

    val check_value :
      ('a -> int array -> 'a) ->
      (('b, [> 'b Bdd.Env.typ ] as 'c, [> 'b Bdd.Env.typdef ] as 'd, 'e, 'f)
       Bdd.Env.O.t, 'a)
```

```
  Bdd.Env.value -> ('b, 'c, 'd, 'e, 'f) Bdd.Env.O.t -> 'a
val check_lvalue :
  ('a -> int array -> 'a) ->
  (('b, [> 'b Bdd.Env.typ ] as 'c, [> 'b Bdd.Env.typdef ] as 'd, 'e, 'f)
   Bdd.Env.O.t, 'a)
  Bdd.Env.value list -> ('b, 'c, 'd, 'e, 'f) Bdd.Env.O.t -> 'a list
type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t, 'c) t = ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'd, [> 'a Bdd.Env.typdef ] as 'e, 'c, 'b)
   Bdd.Env.O.t, 'c)
  Bdd.Expr1.O.Bool.t
type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.d, 'b) Bdd.Env.O.t) dt = ('a
   ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.d,
    'b)
   Bdd.Env.O.t, Cudd.Man.d)
  t
type ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], Cudd.Man.v, 'b) Bdd.Env.O.t) vt = ('a
   ('a, [> 'a Bdd.Env.typ ] as 'c, [> 'a Bdd.Env.typdef ] as 'd, Cudd.Man.v,
    'b)
   Bdd.Env.O.t, Cudd.Man.v)
  t
val of_domain0 :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t ->
  'c Bdd.Domain0.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val get_env :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t -> ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t
val to_domain0 :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'c Bdd.Domain0.t
val of_expr1 :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val to_expr1 :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t
val size :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> int
val print :
  Format.formatter ->
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
```

```
  t -> unit
val bottom :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val top :
  ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
  Bdd.Env.O.t -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Constructors

```
val is_bottom :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> bool
val is_top :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> bool
val is_leq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t -> bool
val is_eq :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t -> bool
val is_variable_unconstrained :
  ('a, ('a, [> 'a Bdd.Env.typ ], [> 'a Bdd.Env.typdef ], 'c, 'b) Bdd.Env.O.t,
   'c)
  t -> 'a -> bool
```

Tests

```
val meet :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val join :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val meet_condition :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
```

```
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Lattice operations

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.t list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val assign_listexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.List.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Assignement

If `rel=true`, it is assumed that `env#bddincr=2` (checked), starting from a pair index. It is also advised to have paired variables in groups.

`rel=true` is most probably much better for assignements of a few variables.

```
val substitute_lexpr :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.t list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
val substitute_listexpr :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a list ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) Bdd.Expr1.O.List.t ->
  ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Substitution

```
val forget_list :
  ('a,
   ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
   Bdd.Env.O.t, 'c)
  t ->
  'a list -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Eliminating variables

```
val change_environment :
```

```
          ('a,
           ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
           Bdd.Env.O.t, 'c)
          t ->
          ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t ->
          ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
        val rename :
          ('a,
           ('a, [> 'a Bdd.Env.typ ] as 'b, [> 'a Bdd.Env.typdef ] as 'd, 'c, 'e)
           Bdd.Env.O.t, 'c)
          t ->
          ('a * 'a) list -> ('a, ('a, 'b, 'd, 'c, 'e) Bdd.Env.O.t, 'c) t
```

Change of environments

```
    end

end
```

Part II

Module `Bddapron` : Finite \& numerical expressions/properties on top of CUDD \& APRON

Chapter 14

Introduction

Expressions and abstract domains combining finite-type and numerical types.

- `Bddapron.Env`[16]: environment defining finite-type variables and user-defined enumerated types, and mapping them to BDDs indices;
- `Bddapron.Cond`[17]: environment defining numerical conditions, and mapping them to BDDs indices;
- `Bddapron.Expr0`[21]: general finite-type and numerical expressions;
- `Bddapron.Domain0`[28]: abstract domains combining `Cudd` BDDs and `Apron` numerical abstract domains; two implemented combinations: `Bddapron.Bdddomain0`[27], `Bddapron.Mtbdddomain0`[25];
- `Bddapron.Expr1`[22], `Bddapron.Domain1`[32], `Bddapron.Bdddomain1`[31], `Bddapron.Mtbdddomain1`[30]: incorporate normalized environments;
- `Bddapron.Expr2`[23]: incorporates in addition normalized condition environment
- `Bddapron.Syntax`[35]: Abstract Syntax Tree for expressions
- `Bddapron.Parser`[38]: converting strings to expressions

`Bddapron`[II] extends `Bdd`[I] (manipulation of finite-type formula using BDDs) and `Apron` (numerical abstract domain library). It enables formula of numerical type, and it allows decisions on numerical constraints in formula.
It can be seen as an extension of the APRON[`http://apron.cri.ensmp.fr/library/`] abstract domain for dealing with Boolean and finite-type variables, in addition to numerical variables.

Chapter 15

Module Apronexpr: Purely arithmetic expressions (internal)

```
module Apronexpr :
  sig
```

Types of numerical variables (distinction is exploited when negating constraints)

```
    type 'a symbol = 'a Bdd.Env.symbol = {
      compare : 'a -> 'a -> int ;
      marshal : 'a -> string ;
      unmarshal : string -> 'a ;
      mutable print : Format.formatter -> 'a -> unit ;
    }
    type typ = [ `Int | `Real ]
    type ('a, [> typ ]) typ_of_var = 'a -> ([> typ ] as 'b)
```

15.1 Expressions

15.1.1 Linear expressions

```
    module Lin :
      sig
        type 'a term = Mpqf.t * 'a
        type 'a t = {
          cst : Mpqf.t ;
          lterm : 'a term list ;
        }
        val normalize : 'a Bddapron.Apronexpr.symbol ->
          'a t -> 'a t
        val compare_lterm :
          'a Bddapron.Apronexpr.symbol ->
          'a term list ->
          'a term list -> int
        val compare : 'a Bddapron.Apronexpr.symbol ->
          'a t -> 'a t -> int
        val var : 'a -> 'a t
        val zero : 'a t
```

```
    val one : 'a t
    val cst : Mpqf.t -> 'a t
    val add : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val sub : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val scale : Mpqf.t -> 'a t -> 'a t
    val negate : 'a t -> 'a t
    val support : 'a Bddapron.Apronexpr.symbol -> 'a t -> 'a PSette.t
    val substitute_by_var :
      'a Bddapron.Apronexpr.symbol ->
      'a t ->
      ('a, 'a) PMappe.t -> 'a t
    val normalize_as_constraint : 'a t -> 'a t
    val print :
      'a Bddapron.Apronexpr.symbol ->
      Format.formatter -> 'a t -> unit
    val of_linexpr0 :
      'a Bddapron.Apronexpr.symbol ->
      Apron.Environment.t -> Apron.Linexpr0.t -> 'a t
    val of_linexpr1 : 'a Bddapron.Apronexpr.symbol ->
      Apron.Linexpr1.t -> 'a t
    val to_linexpr0 :
      'a Bddapron.Apronexpr.symbol ->
      Apron.Environment.t -> 'a t -> Apron.Linexpr0.t
    val to_linexpr1 :
      'a Bddapron.Apronexpr.symbol ->
      Apron.Environment.t -> 'a t -> Apron.Linexpr1.t
  end
```

15.1.2 Polynomial expressions

```
module Poly :
  sig
    type 'a varexp = 'a * int
    type 'a monomial = 'a varexp list
    type 'a term = Mpqf.t * 'a monomial
    type 'a t = 'a term list
    val compare_varexp :
      'a Bddapron.Apronexpr.symbol ->
      'a varexp -> 'a varexp -> int
    val compare_monomial :
      'a Bddapron.Apronexpr.symbol ->
      'a monomial ->
      'a monomial -> int
    val normalize_monomial :
      'a Bddapron.Apronexpr.symbol ->
```

```
      'a monomial -> 'a monomial
    val normalize : 'a Bddapron.Apronexpr.symbol ->
      'a t -> 'a t
    val normalize_full : 'a Bddapron.Apronexpr.symbol ->
      'a t -> 'a t
    val compare : 'a Bddapron.Apronexpr.symbol ->
      'a t -> 'a t -> int
    val cst : Mpqf.t -> 'a t
    val var : 'a -> 'a t
    val add : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val sub : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val scale :
      'a Bddapron.Apronexpr.symbol ->
      Mpqf.t * 'a monomial ->
      'a t -> 'a t
    val mul : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val div : 'a Bddapron.Apronexpr.symbol ->
      'a t ->
      'a t -> 'a t
    val negate : 'a t -> 'a t
    val support : 'a Bddapron.Apronexpr.symbol -> 'a t -> 'a PSette.t
    val substitute_by_var :
      'a Bddapron.Apronexpr.symbol ->
      'a t ->
      ('a, 'a) PMappe.t -> 'a t
    val normalize_as_constraint : 'a t -> 'a t
    val print :
      'a Bddapron.Apronexpr.symbol ->
      Format.formatter -> 'a t -> unit
  end
```

15.1.3 Tree expressions

```
module Tree :
  sig
    type unop = Apron.Texpr1.unop =
      | Neg
      | Cast
      | Sqrt
    type binop = Apron.Texpr1.binop =
      | Add
      | Sub
      | Mul
      | Div
```

```
      | Mod
      | Pow
    type typ = Apron.Texpr1.typ =
      | Real
      | Int
      | Single
      | Double
      | Extended
      | Quad
    type round = Apron.Texpr1.round =
      | Near
      | Zero
      | Up
      | Down
      | Rnd
    type 'a t =
      | Cst of Apron.Coeff.t
      | Var of 'a
      | Unop of unop * 'a t
       * typ * round
      | Binop of binop * 'a t
       * 'a t * typ
       * round
    val support : 'a Bddapron.Apronexpr.symbol -> 'a t -> 'a PSette.t
    val substitute_by_var : 'a t ->
      ('a, 'a) PMappe.t -> 'a t
    val print :
      'a Bddapron.Apronexpr.symbol ->
      Format.formatter -> 'a t -> unit
    val compare : 'a Bddapron.Apronexpr.symbol ->
      'a t -> 'a t -> int
    val of_expr : 'a Bddapron.Apronexpr.symbol ->
      Apron.Texpr1.expr -> 'a t
    val to_expr : 'a Bddapron.Apronexpr.symbol ->
      'a t -> Apron.Texpr1.expr
  end
```

15.1.4 Conversions

```
val lin_of_poly : 'a symbol ->
  'a Poly.t -> 'a Lin.t
val lin_of_tree : 'a symbol ->
  'a Tree.t -> 'a Lin.t
val poly_of_tree : 'a symbol ->
  'a Tree.t -> 'a Poly.t
val tree_of_lin : 'a Lin.t -> 'a Tree.t
val tree_of_poly : 'a Poly.t -> 'a Tree.t
```

15.2 General expressions and operations

```
type 'a t =
```

```
    | Lin of 'a Lin.t
    | Poly of 'a Poly.t
    | Tree of 'a Tree.t

type 'a expr = 'a t

val var : 'a symbol ->
  ('a, [> typ ]) typ_of_var ->
  'a -> 'a t

val zero : 'a t

val one : 'a t

val cst : Apron.Coeff.t -> 'a t

val add :
  'a symbol ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t -> 'a t

val sub :
  'a symbol ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t -> 'a t

val mul :
  'a symbol ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t -> 'a t

val div :
  'a symbol ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t -> 'a t

val gmod :
  'a symbol ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t -> 'a t

val negate : 'a t -> 'a t

val cast :
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t

val sqrt :
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t

val support : 'a symbol -> 'a t -> 'a PSette.t

val substitute_by_var : 'a symbol ->
  'a t -> ('a, 'a) PMappe.t -> 'a t

val normalize : 'a symbol ->
  'a t -> 'a t

val equal : 'a symbol ->
  'a t -> 'a t -> bool

val hash : 'a symbol -> 'a t -> int
```

```
val compare : 'a symbol ->
  'a t -> 'a t -> int
val normalize_as_constraint : 'a t -> 'a t
val is_dependent_on_integer_only :
  ('a, [> typ ]) typ_of_var ->
  'a t -> bool
val typ_of_expr : ('a, [> typ ]) typ_of_var ->
  'a t -> [ `Int | `Real ]
val print : 'a symbol ->
  Format.formatter -> 'a t -> unit
val print_typ : Format.formatter -> [> typ ] -> unit
val of_linexpr0 :
  'a symbol ->
  Apron.Environment.t -> Apron.Linexpr0.t -> 'a t
val of_linexpr1 : 'a symbol -> Apron.Linexpr1.t -> 'a t
val to_linexpr0 :
  'a symbol ->
  Apron.Environment.t -> 'a t -> Apron.Linexpr0.t
val to_linexpr1 :
  'a symbol ->
  Apron.Environment.t -> 'a t -> Apron.Linexpr1.t
val of_texpr0 : 'a symbol ->
  Apron.Environment.t -> Apron.Texpr0.t -> 'a t
val of_texpr1 : 'a symbol -> Apron.Texpr1.t -> 'a t
val to_texpr0 : 'a symbol ->
  Apron.Environment.t -> 'a t -> Apron.Texpr0.t
val to_texpr1 : 'a symbol ->
  Apron.Environment.t -> 'a t -> Apron.Texpr1.t
val to_apron0 :
  'a symbol ->
  Apron.Environment.t ->
  'a t ->
  [ `Lin of Apron.Linexpr0.t | `Tree of Apron.Texpr0.t ]
val to_apron1 :
  'a symbol ->
  Apron.Environment.t ->
  'a t ->
  [ `Lin of Apron.Linexpr1.t | `Tree of Apron.Texpr1.t ]
```

15.3 Constraints

```
module Condition :
  sig
    type typ = Apron.Tcons1.typ =
      | EQ
      | SUPEQ
      | SUP
      | DISEQ
      | EQMOD of Apron.Scalar.t
    type 'a t = typ * 'a Bddapron.Apronexpr.expr
```

```
  val make :
    ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    typ ->
    'a Bddapron.Apronexpr.expr ->
    [ `Bool of bool | `Cond of 'a t ]
  val negate : ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    'a t -> 'a t
  val support : 'a Bddapron.Apronexpr.symbol ->
    'a t -> 'a PSette.t
  val print :
    'a Bddapron.Apronexpr.symbol ->
    Format.formatter -> 'a t -> unit
  val compare : 'a Bddapron.Apronexpr.symbol ->
    'a t -> 'a t -> int
  val of_lincons0 :
    'a Bddapron.Apronexpr.symbol ->
    ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    Apron.Environment.t ->
    Apron.Lincons0.t ->
    [ `Bool of bool | `Cond of 'a t ]
  val of_lincons1 :
    'a Bddapron.Apronexpr.symbol ->
    ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    Apron.Lincons1.t ->
    [ `Bool of bool | `Cond of 'a t ]
  val of_tcons0 :
    'a Bddapron.Apronexpr.symbol ->
    ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    Apron.Environment.t ->
    Apron.Tcons0.t ->
    [ `Bool of bool | `Cond of 'a t ]
  val of_tcons1 :
    'a Bddapron.Apronexpr.symbol ->
    ('a, [> typ ]) Bddapron.Apronexpr.typ_of_var ->
    Apron.Tcons1.t ->
    [ `Bool of bool | `Cond of 'a t ]
  val to_tcons0 :
    'a Bddapron.Apronexpr.symbol ->
    Apron.Environment.t -> 'a t -> Apron.Tcons0.t
  val to_tcons1 :
    'a Bddapron.Apronexpr.symbol ->
    Apron.Environment.t -> 'a t -> Apron.Tcons1.t
  val to_apron0 :
    'a Bddapron.Apronexpr.symbol ->
    Apron.Environment.t ->
    'a t ->
    [ `Lin of Apron.Lincons0.t | `Tree of Apron.Tcons0.t ]
  val to_apron1 :
    'a Bddapron.Apronexpr.symbol ->
    Apron.Environment.t ->
    'a t ->
    [ `Lin of Apron.Lincons1.t | `Tree of Apron.Tcons1.t ]
end
```

```
end
```

Chapter 16

Module Env: Normalized variable managers/environments

```
module Env :
  sig
```

16.1 Types

```
type 'a typdef = 'a Bdd.Env.typdef
```

Type definitions. 'a is the type of symbols (typically, string).

```
type 'a typ = [ `Benum of 'a | `Bint of bool * int | `Bool | `Int | `Real ]
```

Types. 'a is the type of symbols (typically, string).

```
type 'a symbol = 'a Bdd.Env.symbol = {
  compare : 'a -> 'a -> int ;
```

Total order

```
  marshal : 'a -> string ;
```

Conversion to string. The generated strings SHOULD NOT contain NULL character, as they may be converted to C strings.

```
  unmarshal : string -> 'a ;
```

Conversion from string

```
  mutable print : Format.formatter -> 'a -> unit ;
```

Printing

```
}
```

Manager for manipulating symbols.

DO NOT USE Marshal.to_string and Marshal.from_string, as they generate strings with NULL character, which is not handled properly when converted to C strings.

You may use instead Bddapron.Env.marshal[16.3] and Bddapron.Env.unmarshal[16.3].

```
type ('a, 'b) ext = {
  mutable table : 'a Bddapron.Apronexpr.t Cudd.Mtbdd.table ;
  mutable eapron : Apron.Environment.t ;
  mutable aext : 'b ;
}
```

Environment extension. 'a is the type of symbols, 'b is the type of further extension.

```
type ('a, 'b, 'c, 'd) t0 = ('a, 'b, 'c, Cudd.Man.v, ('a, 'd) ext) Bdd.Env.t0
```

Environment.

- 'a is the type of symbols;
- 'b is the type of variables type;
- 'c is the type of type definitions;
- 'd is the type of further extension

See Bdd.Env.t0[5.1] for more (internal) details.

```
module O :
  sig
    type ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd) t = ('a, [> 'a Bd-
    dapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.t0
    val make :
      symbol:'a Bddapron.Env.symbol ->
      copy_aext:('d -> 'd) ->
      ?bddindex0:int ->
      ?bddsize:int ->
      ?relational:bool ->
      Cudd.Man.vt ->
      'd ->
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      t
        Create a new database. Default values for bddindex0,bddsize,relational are
        0,100,false. bddincr is initialized to 1 if relational=false, 2 otherwise.

    val print :
      (Format.formatter -> ([> 'a Bddapron.Env.typ ] as 'b) -> unit) ->
      (Format.formatter -> ([> 'a Bddapron.Env.typdef ] as 'c) -> unit) ->
      (Format.formatter -> 'd -> unit) ->
      Format.formatter -> ('a, 'b, 'c, 'd) t -> unit
        Print an environment

  end
```

16.1.1 Opened signature

```
type 'a t = ('a, 'a typ, 'a typdef, unit) O.t
```

16.2 Printing

```
val print_typ :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> [> 'a typ ] -> unit
```

Print a type

```
val print_typdef :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> [> 'a typdef ] -> unit
```

Print a type definition

```
val print_idcondb :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> Format.formatter -> int * bool -> unit
val print_order :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> Format.formatter -> unit
```

Print the BDD variable ordering

```
val print :
  Format.formatter ->
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> unit
```

Print an environment

16.3 Constructors

```
val marshal : 'a -> string
```

Safe marshalling function, generating strings without NULL characters.
(Based on `Marshal.to_string` with `Marshal.No_sharing` option.)

```
val unmarshal : string -> 'a
```

Companion unmarshalling function

```
val make_symbol :
  ?compare:('a -> 'a -> int) ->
  ?marshal:('a -> string) ->
  ?unmarshal:(string -> 'a) ->
  (Format.formatter -> 'a -> unit) -> 'a symbol
```

Generic function for creating a manager for symbols Default values are `Pervasives.compare`, `Bddapron.Env.marshal`[16.3] and `Bddapron.Env.unmarshal`[16.3].
DO NOT USE `Marshal.to_string` and `Marshal.from_string`, as they generate strings with NULL character, which is not handled properly when converted to C strings.

```
val string_symbol : string symbol
```

Standard manager for symbols of type `string`

```
val make :
  symbol:'a symbol ->
  ?bddindex0:int ->
  ?bddsize:int -> ?relational:bool -> Cudd.Man.vt -> 'a t
```

Create a new environment.

- `symbol` is the manager for manipulating symbols;
- `bddindex0`: starting index in BDDs for finite-type variables;
- `bddsize`: number of indices booked for finite-type variables. If at some point, there is no such available index, a `Failure` exception is raised.
- `relational`: if true, primed indices (unprimed indices plus one) are booked together with unprimed indices. `bddincr` is initialized to 1 if `relational=false`, 2 otherwise.

Default values for `bddindex0,bddsize,relational` are `0,100,false`.

```
val make_string :
  ?bddindex0:int ->
  ?bddsize:int -> ?relational:bool -> Cudd.Man.vt -> string t
```

make_string XXX = make ~symbol:string_symbol XXX

```
val copy :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a, 'b, 'c, 'd) O.t
```

Copy

16.4 Accessors

```
val mem_typ : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a -> bool
```

Is the type defined in the database ?

```
val mem_var : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a -> bool
```

Is the label/var defined in the database ?

```
val mem_label : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a -> bool
```

Is the label a label defined in the database ?

```
val typdef_of_typ :
  ('a, [> 'a typ ], [> 'a typdef ] as 'b, 'd)
  O.t -> 'a -> 'b
```

Return the definition of the type

```
val typ_of_var :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ], 'd)
  O.t -> 'a -> 'b
```

Return the type of the label/variable

```
val vars : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a PSette.t
```

Return the list of variables (not labels)

```
val labels : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a PSette.t
```

Return the list of labels (not variables)

```
val apron :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> Apron.Environment.t
```

return the APRON sub-environment

16.5 Adding types and variables

```
val add_typ_with :
  ('a, [> 'a typ ], [> 'a typdef ] as 'b, 'd)
  O.t -> 'a -> 'b -> unit
```

Declaration of a new type

```
val add_vars_with :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ], 'd)
  O.t -> ('a * 'b) list -> int array option
```

Add the set of variables, possibly normalize the environment and return the applied permutation (that should also be applied to expressions defined in this environment)

```
val remove_vars_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a list -> int array option
```

Remove the set of variables, as well as all constraints, and possibly normalize the environment and return the applied permutation.

```
val rename_vars_with :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t ->
  ('a * 'a) list -> int array option * Apron.Dim.perm option
```

Rename the variables, and remove all constraints,possibly normalize the environment and return the applied permutation.

```
val add_typ :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> 'a -> 'c -> ('a, 'b, 'c, 'd) O.t
val add_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a * 'b) list -> ('a, 'b, 'c, 'd) O.t
val remove_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> 'a list -> ('a, 'b, 'c, 'd) O.t
val rename_vars :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a * 'a) list -> ('a, 'b, 'c, 'd) O.t
```

Functional versions of the previous functions

16.6 Operations

```
val is_leq :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a, 'b, 'c, 'd) O.t -> bool
```

Test inclusion of environments in terms of types and variables (but not in term of indexes)

```
val is_eq :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a, 'b, 'c, 'd) O.t -> bool
```

Test equality of environments in terms of types and variables (but not in term of indexes)

```
val lce :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t ->
  ('a, 'b, 'c, 'd) O.t -> ('a, 'b, 'c, 'd) O.t
```

Least common environment

16.7 Precomputing change of environments

```
type change = {
  cbdd : Cudd.Man.v Bdd.Env.change ;
  capron : Apron.Dim.change2 ;
}
val compute_change :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t -> ('a, 'b, 'c, 'd) O.t -> change
```

16.8 Utilities

```
type ('a, 'b) value = ('a, 'b) Bdd.Env.value = {
  env : 'a ;
  val0 : 'b ;
}
```

Type of pairs `(environment, value)`

```
val make_value :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t ->
  'e -> (('a, 'b, 'c, 'd) O.t, 'e) value
```

Constructor

```
val get_env : ('a, 'b) value -> 'a
val get_val0 : ('a, 'b) value -> 'b
val check_var : ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a -> unit
val check_lvar :
  ('a, [> 'a typ ], [> 'a typdef ], 'd)
  O.t -> 'a list -> unit
val check_value :
  ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
  O.t ->
  (('a, 'b, 'c, 'd) O.t, 'e) value -> unit
val check_value2 :
  (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
   O.t, 'e)
  value ->
  (('a, 'b, 'c, 'd) O.t, 'f) value -> unit
val check_value3 :
  (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
   O.t, 'e)
  value ->
```

```
      (('a, 'b, 'c, 'd) O.t, 'f) value ->
      (('a, 'b, 'c, 'd) O.t, 'g) value -> unit
  val check_lvarvalue :
    ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
    O.t ->
    ('a * (('a, 'b, 'c, 'd) O.t, 'e) value) list ->
    ('a * 'e) list
  val check_lvalue :
    ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
    O.t ->
    (('a, 'b, 'c, 'd) O.t, 'e) value list -> 'e list
  val check_ovalue :
    ('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
    O.t ->
    (('a, 'b, 'c, 'd) O.t, 'e) value option ->
    'e option
  val mapunop :
    ('e -> 'f) ->
    (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
     O.t, 'e)
    value ->
    (('a, 'b, 'c, 'd) O.t, 'f) value
  val mapbinop :
    ('e -> 'f -> 'g) ->
    (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
     O.t, 'e)
    value ->
    (('a, 'b, 'c, 'd) O.t, 'f) value ->
    (('a, 'b, 'c, 'd) O.t, 'g) value
  val mapbinope :
    (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
     O.t -> 'e -> 'f -> 'g) ->
    (('a, 'b, 'c, 'd) O.t, 'e) value ->
    (('a, 'b, 'c, 'd) O.t, 'f) value ->
    (('a, 'b, 'c, 'd) O.t, 'g) value
  val mapterop :
    ('e -> 'f -> 'g -> 'h) ->
    (('a, [> 'a typ ] as 'b, [> 'a typdef ] as 'c, 'd)
     O.t, 'e)
    value ->
    (('a, 'b, 'c, 'd) O.t, 'f) value ->
    (('a, 'b, 'c, 'd) O.t, 'g) value ->
    (('a, 'b, 'c, 'd) O.t, 'h) value
  val var_of_aprondim :
    ('a, [> 'a typ ], [> 'a typdef ], 'd)
    O.t -> Apron.Dim.t -> 'a
  val aprondim_of_var :
    ('a, [> 'a typ ], [> 'a typdef ], 'd)
    O.t -> 'a -> Apron.Dim.t
  val string_of_aprondim :
    ('a, [> 'a typ ], [> 'a typdef ], 'd)
    O.t -> Apron.Dim.t -> string
end
```

Chapter 17

Module Cond: Normalized condition environments

```
module Cond :
  sig
```

17.1 Types

```
type 'a cond = [ `Apron of 'a Bddapron.Apronexpr.Condition.t ]
```

Conditions

```
val print_cond :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> Format.formatter -> [< 'a cond ] -> unit
val compare_cond : 'a Bdd.Env.symbol ->
  [< 'a cond ] -> [< 'a cond ] -> int
val negate_cond :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> 'a cond -> 'a cond
val support_cond :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> [< 'a cond ] -> 'a PSette.t
module O :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t) t = ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'e)
       Bddapron.Env.O.t, 'a Bddapron.Cond.cond, Cudd.Man.v)
      Bdd.Cond.t
    val make :
      symbol:'a Bdd.Env.symbol ->
      ?bddindex0:int ->
      ?bddsize:int ->
      Cudd.Man.vt ->
      ('a,
       ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
       Bddapron.Env.O.t)
      t
```

```
    end

  type 'a t = ('a, 'a Bddapron.Env.t) O.t
  val make :
    symbol:'a Bdd.Env.symbol ->
    ?bddindex0:int -> ?bddsize:int -> Cudd.Man.vt -> 'a t
  val copy : 'a t -> 'a t
  val print : 'a Bddapron.Env.t -> Format.formatter -> 'a t -> unit
```

17.2 Level 2

```
  type ('a, 'b) value = ('a, 'b) Bdd.Cond.value = {
    cond : 'a ;
    val1 : 'b ;
  }
  val make_value : 'a -> 'b -> ('a, 'b) value
  val get_cond : ('a, 'b) value -> 'a
  val get_val1 : ('a, 'b) value -> 'b
  val get_env : ('a, ('b, 'c) Bddapron.Env.value) value -> 'b
  val get_val0 : ('a, ('b, 'c) Bddapron.Env.value) value -> 'c
end
```

Chapter 18

Module ApronexprDD: DDs on top of arithmetic expressions (internal)

```
module ApronexprDD :
  sig
    type 'a t = 'a Bddapron.Apronexpr.t Cudd.Mtbdd.t
    val of_expr : [> `Apron of 'a t ] -> 'a t
    val to_expr : 'a t -> [> `Apron of 'a t ]
    val of_apronexpr :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t -> 'a Bddapron.Apronexpr.t -> 'a t
    val print :
      (Format.formatter -> Cudd.Bdd.vt -> unit) ->
      'a Bddapron.Apronexpr.symbol ->
      Format.formatter -> 'a t -> unit
    val is_zero :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t -> 'a Bddapron.Apronexpr.t -> bool
    val is_one :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t -> 'a Bddapron.Apronexpr.t -> bool
    val absorbant_zero :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t ->
      'a Bddapron.Apronexpr.t Cudd.Mtbdd.unique ->
      'a Bddapron.Apronexpr.t Cudd.Mtbdd.unique option
    val absorbant_one :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t ->
      'a Bddapron.Apronexpr.t Cudd.Mtbdd.unique ->
      'a Bddapron.Apronexpr.t Cudd.Mtbdd.unique option
    val cst :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t -> Apron.Coeff.t -> 'a t
    val var :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t -> 'a -> 'a t
    val add :
```

```
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val sub :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val mul :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val div :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val gmod :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val negate :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> 'a t -> 'a t
val cast :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
val sqrt :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
val support_leaf :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> 'a t -> 'a PSette.t
val support_cond :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
```

```
    Bddapron.Env.O.t -> 'a t -> Cudd.Bdd.vt
  val substitute_linexpr :
    ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
    Bddapron.Env.O.t ->
    'a Bddapron.Apronexpr.Lin.t ->
    ('a, [> `Apron of 'a t ]) PMappe.t ->
    'a t
  val substitute_polyexpr :
    ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
    Bddapron.Env.O.t ->
    'a Bddapron.Apronexpr.Poly.t ->
    ('a, [> `Apron of 'a t ]) PMappe.t ->
    'a t
  val substitute_treeexpr :
    ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
    Bddapron.Env.O.t ->
    'a Bddapron.Apronexpr.Tree.t ->
    ('a, [> `Apron of 'a t ]) PMappe.t ->
    'a t
  val substitute :
    ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
    Bddapron.Env.O.t ->
    'a Bddapron.Apronexpr.t ->
    ('a, [> `Apron of 'a t ]) PMappe.t ->
    'a t
  module Condition :
    sig
      val of_apronexpr :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a Bddapron.Apronexpr.Condition.t -> Cudd.Bdd.vt
      val of_condition :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        [< `Bool of bool | `Cond of 'a Bddapron.Apronexpr.Condition.t ] ->
        Cudd.Bdd.vt
      val make :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        Bddapron.Apronexpr.Condition.typ -> 'a Bddapron.ApronexprDD.t -
      > Cudd.Bdd.vt
      val supeq :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a Bddapron.ApronexprDD.t -> Cudd.Bdd.vt
      val sup :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
```

```
        'a Bddapron.ApronexprDD.t -> Cudd.Bdd.vt
      val eq :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a Bddapron.ApronexprDD.t -> Cudd.Bdd.vt
      val substitute :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a Bddapron.Apronexpr.Condition.t ->
        ('a, [> `Apron of 'a Bddapron.ApronexprDD.t ]) PMappe.t -> Cudd.Bdd.vt
    end

end
```

Chapter 19

Module Common: Functions common to the two implementations of Combined Boolean/Numerical domain

```
module Common :
  sig
```

```
    val tcons0_array_of_cubecond :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Cudd.Bdd.vt -> Apron.Tcons0.t array
```

Converts a cube of conditions into an array of APRON constraints

```
    val lvar_split :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t -> 'a list -> Cudd.Man.v Cudd.Bdd.t * Apron.Dim.t array
```

Split the list of variable into a positive cube (support) of Boolean variables and an array of APRON dimensions

```
    val condition_of_tcons0 :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t ->
      Apron.Tcons0.t ->
      [ `Bool of bool | `Cond of 'a Bddapron.Apronexpr.Condition.t ]
    val bdd_of_tcons0 :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Apron.Tcons0.t -> Cudd.Bdd.vt
    val bdd_of_tcons0_array :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Apron.Tcons0.t array -> Cudd.Bdd.vt
  end
```

Chapter 20

Module ApronDD: DDs on top of Apron abstract values (internal)

```
module ApronDD :
  sig
    type 'a leaf = 'a Apron.Abstract0.t
    type 'a t = 'a leaf Cudd.Mtbddc.t
    type 'a table = 'a leaf Cudd.Mtbddc.table
    type 'a leaf_u = 'a leaf Cudd.Mtbddc.unique
    type 'a global = {
      op_is_leq : ('a leaf_u, 'a leaf_u) Cudd.User.test2 ;
      op_join : ('a leaf_u, 'a leaf_u,
       'a leaf_u)
      Cudd.User.op2 ;
      op_meet : ('a leaf_u, 'a leaf_u,
       'a leaf_u)
      Cudd.User.op2 ;
      op_exist : 'a leaf_u Cudd.User.exist ;
    }
    type 'a man = {
      apron : 'a Apron.Manager.t ;
      table : 'a table ;
      oglobal : 'a global option ;
    }
    val make_table : 'a Apron.Manager.t -> 'a table
    val neutral_join : 'a Apron.Abstract0.t -> bool
    val special_is_leq : 'a Apron.Manager.t ->
      'a t -> 'a t -> bool option
    val special_join : 'a Apron.Manager.t ->
      'a t ->
      'a t -> 'a t option
    val special_meet : 'a Apron.Manager.t ->
      'a t ->
      'a t -> 'a t option
    val make_global : 'a Apron.Manager.t -> 'a table -> 'a global
    val make_man : ?global:bool -> 'a Apron.Manager.t -> 'a man
    val make_op_join :
```

```
  'a man ->
  ('a leaf_u, 'a leaf_u,
   'a leaf_u)
  Cudd.User.op2
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'a Apron.Abstract0.t -> unit) ->
  (Format.formatter -> Cudd.Bdd.vt -> unit) ->
  (int -> string) -> Format.formatter -> 'a t -> unit
val cst : cudd:Cudd.Man.vt ->
  'a man -> 'a Apron.Abstract0.t -> 'a t
val bottom : cudd:Cudd.Man.vt ->
  'a man -> Apron.Dim.dimension -> 'a t
val top : cudd:Cudd.Man.vt ->
  'a man -> Apron.Dim.dimension -> 'a t
val is_bottom : 'a man -> 'a t -> bool
val is_top : 'a man -> 'a t -> bool
val is_eq : 'a man ->
  'a t -> 'a t -> bool
val is_leq : 'a man ->
  'a t -> 'a t -> bool
val join : 'a man ->
  'a t -> 'a t -> 'a t
val meet : 'a man ->
  'a t -> 'a t -> 'a t
val widening : 'a man ->
  'a t -> 'a t -> 'a t
val widening_threshold :
  'a man ->
  'a t ->
  'a t -> Apron.Lincons0.t array -> 'a t
val meet_tcons_array : 'a man ->
  'a t -> Apron.Tcons0.t array -> 'a t
val forget_array : 'a man ->
  'a t -> Apron.Dim.t array -> 'a t
val permute_dimensions : 'a man ->
  'a t -> Apron.Dim.perm -> 'a t
val add_dimensions : 'a man ->
  'a t -> Apron.Dim.change -> bool -> 'a t
val remove_dimensions : 'a man ->
  'a t -> Apron.Dim.change -> 'a t
val apply_dimchange2 : 'a man ->
  'a t -> Apron.Dim.change2 -> bool -> 'a t
type asssub =
  | Assign
  | Substitute
val asssub_texpr_array :
  ?asssub_bdd:(Cudd.Bdd.vt -> Cudd.Bdd.vt) ->
  asssub ->
  'b Bdd.Env.symbol ->
  'a man ->
```

```
      Apron.Environment.t ->
      'a t ->
      Apron.Dim.t array ->
      'b Bddapron.ApronexprDD.t array ->
      'a t option -> 'a t
    val assign_texpr_array :
      'b Bdd.Env.symbol ->
      'a man ->
      Apron.Environment.t ->
      'a t ->
      Apron.Dim.t array ->
      'b Bddapron.ApronexprDD.t array ->
      'a t option -> 'a t
    val substitute_texpr_array :
      'b Bdd.Env.symbol ->
      'a man ->
      Apron.Environment.t ->
      'a t ->
      Apron.Dim.t array ->
      'b Bddapron.ApronexprDD.t array ->
      'a t option -> 'a t
    val exist : 'a man ->
      supp:Cudd.Bdd.vt -> 'a t -> 'a t
    val existand :
      'a man ->
      bottom:'a Apron.Abstract0.t Cudd.Mtbddc.unique ->
      supp:Cudd.Bdd.vt ->
      Cudd.Bdd.vt -> 'a t -> 'a t
  end
```

Chapter 21

Module Expr0: Finite-type and arithmetical expressions

```
module Expr0 :
  sig
```

Important remark:

The following functions may require the creation of new external conditions in the conditional environment.

- the various `substitute` and `substitute_by_var` functions
- `Apron.condition`, `Apron.sup`, `Apron.supeq`, `Apron.eq` functions

21.1 Expressions

```
type 'a t = [ `Apron of 'a Bddapron.ApronexprDD.t
  | `Benum of Cudd.Man.v Bdd.Enum.t
  | `Bint of Cudd.Man.v Bdd.Int.t
  | `Bool of Cudd.Man.v Cudd.Bdd.t ]
type 'a expr = 'a t
```

Type of general expressions

General expressions are described below, after Boolean, bounded integer and enumerated types expressions

21.1.1 Boolean expressions

```
module Bool :
  sig
    type 'a t = Cudd.Bdd.vt
    val of_expr : 'a Bddapron.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr0.expr
    val dtrue : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t
    val dfalse : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t
    val of_bool : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> bool -> 'a t
    val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
```

```
val ite :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t ->
  'a t -> 'a t
val dnot : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a t
val dand :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val dor :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val xor :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nand :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nor :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nxor :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val leq :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val eq :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val is_true : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t -> bool
val is_false : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t -> bool
val is_cst : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t -> bool
val is_leq :
  'a Bddapron.Env.t ->
```

```
      'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val is_eq :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val is_and_false :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val exist :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a list -> 'a t -> 'a t
    val forall :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a list -> 'a t -> 'a t
    val cofactor : 'a t ->
      'a t -> 'a t
    val restrict : 'a t ->
      'a t -> 'a t
    val tdrestrict : 'a t ->
      'a t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

21.1.2 Bounded integer expressions

```
module Bint :
  sig
    type 'a t = Cudd.Man.v Bdd.Int.t
    val of_expr : 'a Bddapron.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr0.expr
    val of_int :
```

```
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  [ `Bint of bool * int ] -> int -> 'a t
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
val ite :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a Bddapron.Expr0.Bool.t ->
  'a t ->
  'a t -> 'a t
val neg : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a t
val succ : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a t
val pred : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a t
val add :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val sub :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val mul :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val shift_left :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val shift_right :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val scale :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val zero :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr0.Bool.t
val eq :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a Bddapron.Expr0.Bool.t
val eq_int :
  'a Bddapron.Env.t ->
```

```
      'a Bddapron.Cond.t ->
      'a t -> int -> 'a Bddapron.Expr0.Bool.t
    val supeq :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.Bool.t
    val supeq_int :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> int -> 'a Bddapron.Expr0.Bool.t
    val sup :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.Bool.t
    val sup_int :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> int -> 'a Bddapron.Expr0.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

21.1.3 Enumerated expressions

```
module Benum :
  sig
    type 'a t = Cudd.Man.v Bdd.Enum.t
    val of_expr : 'a Bddapron.Expr0.expr -> 'a t
```

```
    val to_expr : 'a t -> 'a Bddapron.Expr0.expr
    val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
    val ite :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a Bddapron.Expr0.Bool.t ->
      'a t ->
      'a t -> 'a t
    val eq :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.Bool.t
    val eq_label :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> 'a -> 'a Bddapron.Expr0.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

21.1.4 Arithmetic expressions

```
type apron_coeff = Apron.Coeff.t
type apron_typ = Apron.Texpr1.typ
type apron_round = Apron.Texpr1.round
module Apron :
  sig
    type 'a t = 'a Bddapron.ApronexprDD.t
```

```
val of_expr : 'a Bddapron.Expr0.expr -> 'a t
val to_expr : 'a t -> 'a Bddapron.Expr0.expr
val cst : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> Apron.Coeff.t -> 'a t
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
val add :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val sub :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val mul :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val div :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val gmod :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val negate : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a t -> 'a t
val cast :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
val sqrt :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
```

```
    val supeq :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr0.Bool.t
    val sup :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr0.Bool.t
    val eq :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr0.Bool.t
    val ite :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a Bddapron.Expr0.Bool.t ->
      'a t ->
      'a t -> 'a t
    val cofactor : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr0.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

21.1.5 General expressions

The following operations raise a `Failure` exception in case of a typing error.

```
val typ_of_expr : 'a Bddapron.Env.t -> 'a t -> 'a Bddapron.Env.typ
```

Type of an expression

```
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
```

Expression representing the litteral var

```
val ite :
  'a Bddapron.Env.t ->
```

```
  'a Bddapron.Cond.t ->
  'a Bool.t ->
  'a t -> 'a t -> 'a t
```

If-then-else operation

```
val cofactor : 'a t -> 'a Bool.t -> 'a t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t -> ('a * 'a) list -> 'a t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t list -> ('a * 'a) list -> 'a t list
```

Parallel substitution of variables by variables

```
val substitute :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t -> ('a * 'a t) list -> 'a t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t list ->
  ('a * 'a t) list -> 'a t list
```

Parallel substitution of variables by expressions

```
val restrict : 'a t -> 'a Bool.t -> 'a t
val tdrestrict : 'a t -> 'a Bool.t -> 'a t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val permute : ?memo:Cudd.Memo.t -> 'a t -> int array -> 'a t
val varmap : 'a t -> 'a t
```

Permutation (rather internal)

```
val support : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t -> 'a PSette.t
```

Return the full support of the expression

```
val eq :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a t -> 'a t -> 'a Bool.t
```

Under which condition are the expressions equal ? In case of arithmetic expressions, do not take into account the careset.

```
val support_cond : Cudd.Man.vt -> 'a t -> Cudd.Bdd.vt
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

```
val print :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> Format.formatter -> [< 'a t ] -> unit
```

Printing functions

```
val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Cond.t * 'a t list ->
  'a Bddapron.Cond.t * 'a t list
```

21.2 Opened signature and Internal functions

```
module O :
  sig
    val check_typ2 :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t ->
      [< 'a Bddapron.Expr0.t ] -> [< 'a Bddapron.Expr0.t ] -> 'a Bddapron.Env.typ
    module Bool :
      sig
        type 'a t = Cudd.Bdd.vt
        val of_expr : 'a Bddapron.Expr0.expr -> 'a t
        val to_expr : 'a t -> 'a Bddapron.Expr0.expr
        val dtrue :
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'a t
        val dfalse :
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'a t
        val of_bool :
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          bool -> 'a t
        val var :
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'a -> 'a t
        val ite :
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
```

```
  'a t ->
  'a t ->
  'a t -> 'a t

val dnot :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t

val dand :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val dor :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val xor :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val nand :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val nor :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val nxor :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val leq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t

val eq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
```

```
  'a t ->
  'a t -> 'a t
val is_true :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> bool
val is_false :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> bool
val is_cst :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> bool
val is_leq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t -> bool
val is_eq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t -> bool
val is_and_false :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t -> bool
val exist :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a list -> 'a t -> 'a t
val forall :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a list -> 'a t -> 'a t
val cofactor : 'a t ->
  'a t -> 'a t
val restrict : 'a t ->
  'a t -> 'a t
val tdrestrict : 'a t ->
  'a t -> 'a t
val permute : ?memo:Cudd.Memo.t ->
  'a t -> int array -> 'a t
val varmap : 'a t -> 'a t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
```

```
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a t -> ('a * 'a) list -> 'a t
      val substitute :
        ?memo:Cudd.Memo.t ->
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a t ->
        ('a * 'a Bddapron.Expr0.expr) list -> 'a t
      val print :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        Format.formatter -> 'a t -> unit
    end

  module Bint :
    sig
      type 'a t = Cudd.Man.v Bdd.Int.t
      val of_expr : 'a Bddapron.Expr0.expr -> 'a t
      val to_expr : 'a t -> 'a Bddapron.Expr0.expr
      val of_int :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        [> `Bint of bool * int ] -> int -> 'a t
      val var :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a -> 'a t
      val ite :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a Bddapron.Expr0.O.Bool.t ->
        'a t ->
        'a t -> 'a t
      val neg :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a t -> 'a t
      val succ :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a t -> 'a t
      val pred :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
```

```
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t
val add :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t
val sub :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t
val mul :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a t
val shift_left :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  int -> 'a t -> 'a t
val shift_right :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  int -> 'a t -> 'a t
val scale :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  int -> 'a t -> 'a t
val zero :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a Bddapron.Expr0.O.Bool.t
val eq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t ->
  'a t -> 'a Bddapron.Expr0.O.Bool.t
val eq_int :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> int -> 'a Bddapron.Expr0.O.Bool.t
val supeq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
```

```
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.O.Bool.t
    val supeq_int :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> int -> 'a Bddapron.Expr0.O.Bool.t
    val sup :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.O.Bool.t
    val sup_int :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> int -> 'a Bddapron.Expr0.O.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Format.formatter -> 'a t -> unit
  end

module Benum :
  sig
    type 'a t = Cudd.Man.v Bdd.Enum.t
    val of_expr : 'a Bddapron.Expr0.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr0.expr
```

```
    val var :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a -> 'a t
    val ite :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a Bddapron.Expr0.O.Bool.t ->
      'a t ->
      'a t -> 'a t
    val eq :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr0.O.Bool.t
    val eq_label :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> 'a -> 'a Bddapron.Expr0.O.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr0.O.Bool.t -> 'a t
    val permute : ?memo:Cudd.Memo.t ->
      'a t -> int array -> 'a t
    val varmap : 'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Format.formatter -> 'a t -> unit
  end

module Apron :
  sig
```

```
type 'a t = 'a Bddapron.ApronexprDD.t
val of_expr : [> `Apron of 'a t ] -> 'a t
val to_expr : 'a t -> [> `Apron of 'a t ]
val cst :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  Bddapron.Expr0.apron_coeff -> 'a t
val var :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a -> 'a t
val add :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t ->
  'a t -> 'a t
val sub :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t ->
  'a t -> 'a t
val mul :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t ->
  'a t -> 'a t
val div :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t ->
  'a t -> 'a t
val gmod :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t ->
  'a t -> 'a t
val negate :
```

```
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a t
val cast :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t -> 'a t
val sqrt :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr0.apron_typ ->
  ?round:Bddapron.Expr0.apron_round ->
  'a t -> 'a t
val supeq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a Bddapron.Expr0.O.Bool.t
val sup :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a Bddapron.Expr0.O.Bool.t
val eq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a t -> 'a Bddapron.Expr0.O.Bool.t
val ite :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.O.Bool.t ->
  'a t ->
  'a t -> 'a t
val cofactor : 'a t ->
  'a Bddapron.Expr0.O.Bool.t -> 'a t
val restrict : 'a t ->
  'a Bddapron.Expr0.O.Bool.t -> 'a t
val tdrestrict : 'a t ->
  'a Bddapron.Expr0.O.Bool.t -> 'a t
val permute : ?memo:Cudd.Memo.t ->
  'a t -> int array -> 'a t
val varmap : 'a t -> 'a t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
```

```
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a t ->
      ('a * 'a Bddapron.Expr0.expr) list -> 'a t
    val print :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      Format.formatter -> 'a t -> unit
  end
```

The following operations raise a `Failure` exception in case of a typing error.

```
val typ_of_expr :
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
  Bddapron.Env.O.t -> [< 'a Bddapron.Expr0.t ] -> 'a Bddapron.Env.typ
```

Type of an expression

```
val var :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a -> 'a Bddapron.Expr0.t
```

Expression representing the litteral var

```
val ite :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bool.t ->
  'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t
```

If-then-else operation

```
val cofactor : 'a Bddapron.Expr0.t -> 'a Bool.t -> 'a Bddapron.Expr0.t
val restrict : 'a Bddapron.Expr0.t -> 'a Bool.t -> 'a Bddapron.Expr0.t
val tdrestrict : 'a Bddapron.Expr0.t -> 'a Bool.t -> 'a Bddapron.Expr0.t
val permute :
  ?memo:Cudd.Memo.t -> 'a Bddapron.Expr0.t -> int array -> 'a Bddapron.Expr0.t
val permute_list :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Expr0.t list -> int array -> 'a Bddapron.Expr0.t list
val varmap : 'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t -> ('a * 'a) list -> 'a Bddapron.Expr0.t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
```

```
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t list -> ('a * 'a) list -> 'a Bddapron.Expr0.t list
```

Parallel substitution of variables by variables

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t -> ('a * 'a Bddapron.Expr0.t) list -> 'a Bddapron.Expr0.t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t list ->
  ('a * 'a Bddapron.Expr0.t) list -> 'a Bddapron.Expr0.t list
```

Parallel substitution of variables by expressions

```
val support :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t -> 'a PSette.t
```

Return the full support of the expression

```
val eq :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t -> 'a Bool.t
```

Under which condition are the expressions equal ? In case of arithmetic expressions, do not take into account the careset.

```
val support_cond : Cudd.Man.vt -> 'a Bddapron.Expr0.t -> Cudd.Bdd.vt
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

Printing functions

```
val print :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  Format.formatter -> [< 'a Bddapron.Expr0.t ] -> unit
val print_bdd :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  Format.formatter -> Cudd.Bdd.vt -> unit
val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  ('a,
```

```
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t)
        Bddapron.Cond.O.t * 'a Bddapron.Expr0.t list ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t *
        'a Bddapron.Expr0.t list
    val compose_of_lvarexpr :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      ('a * 'a Bddapron.Expr0.t) list ->
      Cudd.Bdd.vt array option * ('a, 'a Bddapron.Expr0.t) PMappe.t
  end

end
```

Chapter 22

Module Expr1: Finite-type and arithmetical expressions with normalized environments

```
module Expr1 :
  sig
```

Important remark:

The following functions may require the creation of new external conditions in the conditional environment.

- the various `substitute` and `substitute_by_var` functions
- `Apron.condition`, `Apron.sup`, `Apron.supeq`, `Apron.eq` functions

22.1 Expressions

```
type 'a t = ('a Bddapron.Env.t, 'a Bddapron.Expr0.t) Bddapron.Env.value
type 'a expr = 'a t
```

Type of general expressions

22.1.1 Boolean expressions

```
module Bool :
  sig
    type 'a t = ('a Bddapron.Env.t, Cudd.Man.v Bddapron.Expr0.Bool.t) Bddapron.Env.value
    val of_expr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.Bool.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val to_expr0 : 'a t -> 'a Bddapron.Expr0.Bool.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : 'a Bddapron.Expr1.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr1.expr
```

Conversion from/to general expression

```
val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val dtrue : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t
val dfalse : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a t
val of_bool : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> bool -> 'a t
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
```

22.1.1.1 Logical connectors

```
val dnot : 'a Bddapron.Cond.t -> 'a t -> 'a t
val dand : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val dor : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
```

`not`, `and` and `or` (use of 'd' prefix because of conflict with OCaml keywords)

```
val xor : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nand : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nor : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val nxor : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
```

Exclusive or, not and, nor or and not xor

```
val eq : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
```

Same as `nxor`

```
val leq : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
```

Implication

```
val ite : 'a Bddapron.Cond.t ->
  'a t ->
  'a t ->
  'a t -> 'a t
```

If-then-else

```
val is_true : 'a Bddapron.Cond.t -> 'a t -> bool
val is_false : 'a Bddapron.Cond.t -> 'a t -> bool
```

```
    val is_cst : 'a Bddapron.Cond.t -> 'a t -> bool
    val is_eq : 'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val is_leq : 'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val is_inter_false : 'a Bddapron.Cond.t ->
      'a t -> 'a t -> bool
    val exist : 'a Bddapron.Cond.t ->
      'a list -> 'a t -> 'a t
    val forall : 'a Bddapron.Cond.t ->
      'a list -> 'a t -> 'a t
    val cofactor : 'a t ->
      'a t -> 'a t
    val restrict : 'a t ->
      'a t -> 'a t
    val tdrestrict : 'a t ->
      'a t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr1.expr) list -> 'a t
    val print : 'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

22.1.2 Bounded integer expressions

```
module Bint :
  sig
    type 'a t = ('a Bddapron.Env.t, Cudd.Man.v Bdd.Int.t) Bddapron.Env.value
    val of_expr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.Bint.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val to_expr0 : 'a t -> 'a Bddapron.Expr0.Bint.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : 'a Bddapron.Expr1.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr1.expr
```

Conversion from/to general expression

```
    val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val of_int :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  [ `Bint of bool * int ] -> int -> 'a t
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
val neg : 'a Bddapron.Cond.t -> 'a t -> 'a t
val succ : 'a Bddapron.Cond.t -> 'a t -> 'a t
val pred : 'a Bddapron.Cond.t -> 'a t -> 'a t
val add : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val sub : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val mul : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a t
val shift_left : 'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val shift_right : 'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val scale : 'a Bddapron.Cond.t ->
  int -> 'a t -> 'a t
val ite :
  'a Bddapron.Cond.t ->
  'a Bddapron.Expr1.Bool.t ->
  'a t ->
  'a t -> 'a t
val zero : 'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr1.Bool.t
val eq : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a Bddapron.Expr1.Bool.t
val supeq :
  'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a Bddapron.Expr1.Bool.t
val sup : 'a Bddapron.Cond.t ->
  'a t ->
  'a t -> 'a Bddapron.Expr1.Bool.t
val eq_int : 'a Bddapron.Cond.t ->
  'a t -> int -> 'a Bddapron.Expr1.Bool.t
val supeq_int :
  'a Bddapron.Cond.t ->
  'a t -> int -> 'a Bddapron.Expr1.Bool.t
val sup_int : 'a Bddapron.Cond.t ->
  'a t -> int -> 'a Bddapron.Expr1.Bool.t
val cofactor : 'a t ->
  'a Bddapron.Expr1.Bool.t -> 'a t
val restrict : 'a t ->
  'a Bddapron.Expr1.Bool.t -> 'a t
val tdrestrict : 'a t ->
```

```
    'a Bddapron.Expr1.Bool.t -> 'a t
  val substitute_by_var :
    ?memo:Cudd.Memo.t ->
    'a Bddapron.Cond.t ->
    'a t -> ('a * 'a) list -> 'a t
  val substitute :
    ?memo:Cudd.Memo.t ->
    'a Bddapron.Cond.t ->
    'a t ->
    ('a * 'a Bddapron.Expr1.expr) list -> 'a t
  val guard_of_int :
    'a Bddapron.Cond.t ->
    'a t -> int -> 'a Bddapron.Expr1.Bool.t
```

Return the guard of the integer value.

```
  val guardints :
    'a Bddapron.Cond.t ->
    'a t -> ('a Bddapron.Expr1.Bool.t * int) list
```

Return the list `g -> n` of guarded values.

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

22.1.3 Enumerated expressions

```
module Benum :
  sig
    type 'a t = ('a Bddapron.Env.t, Cudd.Man.v Bdd.Enum.t) Bddapron.Env.value
    val of_expr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.Benum.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val to_expr0 : 'a t -> 'a Bddapron.Expr0.Benum.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : 'a Bddapron.Expr1.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr1.expr
```

Conversion from/to general expression

```
    val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
    val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
    val ite :
      'a Bddapron.Cond.t ->
      'a Bddapron.Expr1.Bool.t ->
      'a t ->
      'a t -> 'a t
    val eq : 'a Bddapron.Cond.t ->
      'a t ->
      'a t -> 'a Bddapron.Expr1.Bool.t
```

```
    val eq_label : 'a Bddapron.Cond.t ->
      'a t -> 'a -> 'a Bddapron.Expr1.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr1.expr) list -> 'a t
    val guard_of_label :
      'a Bddapron.Cond.t ->
      'a t -> 'a -> 'a Bddapron.Expr1.Bool.t
```

Return the guard of the label.

```
    val guardlabels :
      'a Bddapron.Cond.t ->
      'a t -> ('a Bddapron.Expr1.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

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

22.1.4 Arithmetic expressions

```
type apron_coeff = Apron.Coeff.t
type apron_typ = Apron.Texpr1.typ
type apron_round = Apron.Texpr1.round
type apron_cons_typ = Apron.Tcons1.typ
module Apron :
  sig
    type 'a t = ('a Bddapron.Env.t, 'a Bddapron.Expr0.Apron.t) Bddapron.Env.value
    val of_expr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.Apron.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val to_expr0 : 'a t -> 'a Bddapron.Expr0.Apron.t
```

Extract resp. the environment and the underlying expression of level 0

```
    val of_expr : 'a Bddapron.Expr1.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr1.expr
```

Conversion from/to general expression

```
    val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
val cst : 'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> Apron.Coeff.t -> 'a t
val add :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val mul :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val sub :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val div :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val gmod :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t ->
  'a t -> 'a t
val negate : 'a Bddapron.Cond.t -> 'a t -> 'a t
val sqrt :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
val cast :
  'a Bddapron.Cond.t ->
  ?typ:Apron.Texpr1.typ ->
  ?round:Apron.Texpr1.round ->
  'a t -> 'a t
val ite :
  'a Bddapron.Cond.t ->
  'a Bddapron.Expr1.Bool.t ->
  'a t ->
  'a t -> 'a t
val condition :
  'a Bddapron.Cond.t ->
```

```
      Apron.Tcons1.typ -> 'a t -> 'a Bddapron.Expr1.Bool.t
    val supeq : 'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr1.Bool.t
    val sup : 'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr1.Bool.t
    val eq : 'a Bddapron.Cond.t -> 'a t -> 'a Bddapron.Expr1.Bool.t
    val cofactor : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val restrict : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val tdrestrict : 'a t ->
      'a Bddapron.Expr1.Bool.t -> 'a t
    val substitute_by_var :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t -> ('a * 'a) list -> 'a t
    val substitute :
      ?memo:Cudd.Memo.t ->
      'a Bddapron.Cond.t ->
      'a t ->
      ('a * 'a Bddapron.Expr1.expr) list -> 'a t
    val print : 'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
  end
```

22.1.5 General expressions

```
val typ_of_expr : 'a t -> 'a Bddapron.Env.typ
```

Type of an expression

```
val make : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.t -> 'a t
val of_expr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
val get_env : 'a t -> 'a Bddapron.Env.t
val to_expr0 : 'a t -> 'a Bddapron.Expr0.t
```

Extract resp. the environment and the underlying expression of level 0

```
val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val var : 'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> 'a -> 'a t
```

Expression representing the litteral var

```
val ite : 'a Bddapron.Cond.t ->
  'a Bool.t ->
  'a t -> 'a t -> 'a t
```

If-then-else operation

```
val eq : 'a Bddapron.Cond.t ->
  'a t -> 'a t -> 'a Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Cond.t ->
  'a t -> ('a * 'a) list -> 'a t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Cond.t ->
  'a t list -> ('a * 'a) list -> 'a t list
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Cond.t ->
  'a t -> ('a * 'a t) list -> 'a t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  'a Bddapron.Cond.t ->
  'a t list ->
  ('a * 'a t) list -> 'a t list
```

Parallel substitution of variables by expressions

```
val support : 'a Bddapron.Cond.t -> 'a t -> 'a PSette.t
```

Support of the expression

```
val support_cond : Cudd.Man.vt -> 'a t -> Cudd.Bdd.vt
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

```
val cofactor : 'a t -> 'a Bool.t -> 'a t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val restrict : 'a t -> 'a Bool.t -> 'a t
val tdrestrict : 'a t -> 'a Bool.t -> 'a t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val print : 'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Cond.t * 'a t list ->
  'a Bddapron.Cond.t * 'a t list
```

22.1.6 List of expressions

```
module List :
  sig
    type 'a t = ('a Bddapron.Env.t, 'a Bddapron.Expr0.t list) Bddapron.Env.value
    val of_lexpr0 : 'a Bddapron.Env.t -> 'a Bddapron.Expr0.t list -> 'a t
```

Creation from a list of expressions of level 0 (without environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val to_lexpr0 : 'a t -> 'a Bddapron.Expr0.t list
```

Extract resp. the environment and the underlying list of expressions of level 0

```
val of_lexpr : 'a Bddapron.Env.t -> 'a Bddapron.Expr1.expr list -> 'a t
val to_lexpr : 'a t -> 'a Bddapron.Expr1.expr list
```

Conversion from/to lists of general expression

```
val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Cond.t * 'a t ->
  'a Bddapron.Cond.t * 'a t
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  'a Bddapron.Cond.t -> Format.formatter -> 'a t -> unit
end
```

22.2 Opened signature and Internal functions

```
module O :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
    dapron.Env.typdef ] as 'c, 'e)
       Bddapron.Env.O.t, 'a Bddapron.Expr0.t)
      Bddapron.Env.value
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
      Bddapron.Env.O.t) expr = ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'b)
       Bddapron.Env.O.t)
      t
```

Type of general expressions

```
    module Bool :
      sig
        type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
          Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
        dapron.Env.typdef ] as 'c, 'e)
           Bddapron.Env.O.t, 'a Bddapron.Expr0.Bool.t)
          Bddapron.Env.value
        val of_expr :
          (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
           Bddapron.Env.O.t, [> `Bool of 'a Bddapron.Expr0.Bool.t ])
          Bddapron.Env.value ->
          ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
        val to_expr :
          ('a,
           ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
           Bddapron.Env.O.t)
```

```
  t ->
  (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Bool of 'a Bd-
dapron.Expr0.Bool.t ])
  Bddapron.Env.value
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val dtrue :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val dfalse :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val of_bool :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  bool -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val var :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

22.2.0.1 Logical connectors

```
val dnot :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val dand :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val dor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
```

```
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

not, and and or (use of 'd' prefix because of conflict with OCaml keywords)

```
val xor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val nand :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val nor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val nxor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Exclusive or, not and, nor or and not xor

```
val eq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Same as nxor

```
val leq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Implication

```
val ite :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

If-then-else

```
val is_true :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val is_false :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val is_cst :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val is_eq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val is_leq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val is_inter_false :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> bool
val exist :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
```

```
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  'a list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val forall :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  'a list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val cofactor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val restrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr) list -
>
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val print :
  ('a,
```

```
          ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
          Bddapron.Env.O.t)
         Bddapron.Cond.O.t ->
         Format.formatter ->
         ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
    end

  module Bint :
    sig
      type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
        Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
      dapron.Env.typdef ] as 'c, 'e)
         Bddapron.Env.O.t, Cudd.Man.v Bdd.Int.t)
        Bddapron.Env.value
      val of_expr :
        (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
         Bddapron.Env.O.t, [> `Bint of 'a Bddapron.Expr0.Bint.t ])
        Bddapron.Env.value ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
      val to_expr :
        ('a,
         ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
         Bddapron.Env.O.t)
        t ->
        (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Bint of 'a Bd-
      dapron.Expr0.Bint.t ])
        Bddapron.Env.value
      val extend_environment :
        ('a,
         ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
         Bddapron.Env.O.t)
        t ->
        ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
      val of_int :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        [ `Bint of bool * int ] ->
        int -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
      val var :
        ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
        Bddapron.Env.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
        'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
      val neg :
        ('a,
         ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
         Bddapron.Env.O.t)
        Bddapron.Cond.O.t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
        ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
      val succ :
        ('a,
```

```
    ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
    Bddapron.Env.O.t)
   Bddapron.Cond.O.t ->
   ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
   ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val pred :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val add :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val sub :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val mul :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val shift_left :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  int ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val shift_right :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  int ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val scale :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
```

```
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  int ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val ite :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val zero :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val eq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val supeq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val sup :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val eq_int :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  int -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val supeq_int :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
```

```
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  int -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val sup_int :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  int -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val cofactor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val restrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr) list -
>
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val guard_of_int :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
```

```
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      int -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
```
Return the guard of the integer value.

```
    val guardints :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      (('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t * int) list
```
Return the list `g -> n` of guarded values.

```
    val print :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      Format.formatter ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
  end

module Benum :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
    dapron.Env.typdef ] as 'c, 'e)
       Bddapron.Env.O.t, Cudd.Man.v Bdd.Enum.t)
      Bddapron.Env.value
    val of_expr :
      (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t, [> `Benum of 'a Bddapron.Expr0.Benum.t ])
      Bddapron.Env.value ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
    val to_expr :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      t ->
      (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Benum of 'a Bd-
    dapron.Expr0.Benum.t ])
      Bddapron.Env.value
    val extend_environment :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      t ->
      ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
    val var :
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

```
val ite :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val eq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val eq_label :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val cofactor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val restrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute :
```

```
      ?memo:Cudd.Memo.t ->
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr) list -
    >
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

val guard_of_label :

```
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
```

Return the guard of the label.

val guardlabels :

```
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      (('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t * 'a) list
```

Return the list `g -> label` of guarded values.

val print :

```
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      Format.formatter ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
  end

module Apron :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
    dapron.Env.typdef ] as 'c, 'e)
       Bddapron.Env.O.t, 'a Bddapron.Expr0.Apron.t)
      Bddapron.Env.value
    val of_expr :
      (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t, [> `Apron of 'a Bddapron.Expr0.Apron.t ])
      Bddapron.Env.value ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
    val to_expr :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      t ->
      (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Apron of 'a Bd-
    dapron.Expr0.Apron.t ])
```

```
  Bddapron.Env.value
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val var :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val cst :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  Bddapron.Expr1.apron_coeff ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val add :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr1.apron_typ ->
  ?round:Bddapron.Expr1.apron_round ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val mul :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr1.apron_typ ->
  ?round:Bddapron.Expr1.apron_round ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val sub :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr1.apron_typ ->
  ?round:Bddapron.Expr1.apron_round ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val div :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ?typ:Bddapron.Expr1.apron_typ ->
```

```
    ?round:Bddapron.Expr1.apron_round ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val gmod :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ?typ:Bddapron.Expr1.apron_typ ->
    ?round:Bddapron.Expr1.apron_round ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val negate :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val sqrt :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ?typ:Bddapron.Expr1.apron_typ ->
    ?round:Bddapron.Expr1.apron_round ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val cast :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ?typ:Bddapron.Expr1.apron_typ ->
    ?round:Bddapron.Expr1.apron_round ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val ite :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val condition :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    Bddapron.Expr1.apron_cons_typ ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
```

```
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
  val supeq :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
  val sup :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
  val eq :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
  val cofactor :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val restrict :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val tdrestrict :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val substitute_by_var :
    ?memo:Cudd.Memo.t ->
    ('a,
     ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
     Bddapron.Env.O.t)
    Bddapron.Cond.O.t ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
    ('a * 'a) list ->
    ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
  val substitute :
    ?memo:Cudd.Memo.t ->
```

```
      ('a,
       ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
      ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr) list -
    >
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
   val print :
     ('a,
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t)
     Bddapron.Cond.O.t ->
     Format.formatter ->
     ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
 end

val typ_of_expr :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> 'a Bddapron.Env.typ
```

Type of an expression

```
val make :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  'a Bddapron.Expr0.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Creation from an expression without environment

```
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val var :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Expression representing the litteral var

```
val ite :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

If-then-else operation

```
val eq :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bool.t
```

Equality operation

```
val substitute_by_var :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * 'a) list -> ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute_by_var_list :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list ->
  ('a * 'a) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val substitute_list :
  ?memo:Cudd.Memo.t ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list ->
  ('a * ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t) list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list
```

Parallel substitution of variables by expressions

```
val support :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> 'a PSette.t
```

Support of the expression

```
val support_cond :
  Cudd.Man.vt ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> Cudd.Bdd.vt
```

Return the support of an expression as a conjunction of the BDD identifiers involved in the expression

```
val cofactor :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Evaluate the expression. The BDD is assumed to be a cube

```
val restrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val tdrestrict :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bool.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
```

Simplify the expression knowing that the BDD is true. Generalizes `cofactor`.

```
val print :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  Format.formatter ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t *
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t *
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t list
```

22.2.1 List of expressions

```
module List :
  sig
```

```
type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t) t = (('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
dapron.Env.typdef ] as 'c, 'e)
   Bddapron.Env.O.t, 'a Bddapron.Expr0.t list)
  Bddapron.Env.value

val of_lexpr0 :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  'a Bddapron.Expr0.t list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t

val get_env :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t -> ('a, 'b, 'c, 'd) Bddapron.Env.O.t

val to_lexpr0 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> 'a Bddapron.Expr0.t list

val of_lexpr :
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t

val to_lexpr :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.expr list

val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t

val normalize :
  ?reduce:bool ->
  ?careset:bool ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t *
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t *
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t

val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
```

```
                Bddapron.Env.O.t)
               Bddapron.Cond.O.t ->
               Format.formatter ->
               ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t -> unit
          end

      end

  end
```

Chapter 23

Module Expr2: Finite-type and arithmetical expressions with variable and condition environments

```
module Expr2 :
  sig
```

23.1 Opened signature

```
module O :
  sig
```

23.1.1 Boolean expressions

```
module Bool :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
      Bddapron.Env.O.t) t = (('a,
        ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'b)
        Bddapron.Env.O.t)
       Bddapron.Cond.O.t,
       ('a, ('a, 'c, 'd, 'b) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t)
      Bdd.Cond.value
    val of_expr0 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a Bddapron.Expr0.Bool.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
    val of_expr1 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      ('a,
```

```
    ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
    Bddapron.Env.O.t)
   Bddapron.Cond.O.t ->
   ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
   ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val get_env :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t -> ('a, 'b, 'c, 'd) Bddapron.Env.O.t
val get_cond :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t
val to_expr0 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> 'a Bddapron.Expr0.Bool.t
val to_expr1 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
val of_expr :
  (('a,
    ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
    Bddapron.Env.O.t)
   Bddapron.Cond.O.t,
   (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Bool of 'a Bd-
dapron.Expr0.Bool.t ])
   Bddapron.Env.value)
  Bdd.Cond.value ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val to_expr :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  (('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t,
   (('a, 'b, 'c, 'd) Bddapron.Env.O.t, [> `Bool of 'a Bd-
dapron.Expr0.Bool.t ])
   Bddapron.Env.value)
  Bdd.Cond.value
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val is_false :
```

```
      ('a,
       ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
       Bddapron.Env.O.t)
      t -> bool
  val is_true :
    ('a,
     ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
     Bddapron.Env.O.t)
    t -> bool
  val print :
    Format.formatter ->
    ('a,
     ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
     Bddapron.Env.O.t)
    t -> unit
end
```

23.1.2 General expressions

```
type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
  Bddapron.Env.O.t) t = (('a,
    ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'b)
    Bddapron.Env.O.t)
   Bddapron.Cond.O.t,
   ('a, ('a, 'c, 'd, 'b) Bddapron.Env.O.t) Bddapron.Expr1.O.t)
  Bdd.Cond.value
type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
  Bddapron.Env.O.t) expr = ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'b)
   Bddapron.Env.O.t)
  t
val of_expr0 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'a Bddapron.Expr0.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val of_expr1 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val get_env :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
```

```
  t -> ('a, 'b, 'c, 'd) Bddapron.Env.O.t
val get_cond :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t
val to_expr0 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> 'a Bddapron.Expr0.t
val to_expr1 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.t
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val print :
  Format.formatter ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> unit
```

23.1.3 List of expressions

```
module List :
  sig
    type ('a, ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
      Bddapron.Env.O.t) t = (('a,
        ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'b)
        Bddapron.Env.O.t)
       Bddapron.Cond.O.t,
       ('a, ('a, 'c, 'd, 'b) Bddapron.Env.O.t) Bddapron.Expr1.O.List.t)
      Bdd.Cond.value
    val of_lexpr0 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
      Bddapron.Env.O.t ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a Bddapron.Expr0.t list ->
      ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
    val of_lexpr1 :
      ?normalize:bool ->
```

```
  ?reduce:bool ->
  ?careset:bool ->
  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
  Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.t list ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val of_listexpr1 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  Bddapron.Cond.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.List.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val get_env :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t -> ('a, 'b, 'c, 'd) Bddapron.Env.O.t
val get_cond :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t
val to_lexpr0 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
   Bddapron.Env.O.t)
  t -> 'a Bddapron.Expr0.t list
val to_lexpr1 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.t list
val to_listexpr1 :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.List.t
val extend_environment :
  ('a,
   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bddapron.Env.typdef ] as 'c, 'd)
   Bddapron.Env.O.t)
  t ->
  ('a, 'b, 'c, 'd) Bddapron.Env.O.t ->
  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) t
val print :
  Format.formatter ->
```

```
          ('a,
           ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
           Bddapron.Env.O.t)
          t -> unit
      end

  end
```

23.2 Closed signature

```
type 'a t = ('a Bddapron.Cond.t, 'a Bddapron.Expr1.t) Bdd.Cond.value
type 'a expr = 'a t
val of_expr0 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a Bddapron.Expr0.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
val of_expr1 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Cond.t -> 'a Bddapron.Expr1.t -> 'a t
```

Creation from an expression of level 1 (without condition environment)

```
val get_env : 'a t -> 'a Bddapron.Env.t
val get_cond : 'a t -> 'a Bddapron.Cond.t
```

Extract resp. the environment and the condition environment

```
val to_expr0 : 'a t -> 'a Bddapron.Expr0.t
val to_expr1 : 'a t -> 'a Bddapron.Expr1.t
```

Extract the underlying expression of level 0 and 1

```
val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
val print : Format.formatter -> 'a t -> unit
module Bool :
  sig
    type 'a t = ('a Bddapron.Cond.t, 'a Bddapron.Expr1.Bool.t) Bdd.Cond.value
    val of_expr0 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a Bddapron.Expr0.Bool.t -> 'a t
```

Creation from an expression of level 0 (without environment)

```
    val of_expr1 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      'a Bddapron.Cond.t -> 'a Bddapron.Expr1.Bool.t -> 'a t
```

Creation from an expression of level 1 (without condition environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val get_cond : 'a t -> 'a Bddapron.Cond.t
```

Extract resp. the environment and the condition environment

```
    val to_expr0 : 'a t -> 'a Bddapron.Expr0.Bool.t
    val to_expr1 : 'a t -> 'a Bddapron.Expr1.Bool.t
```

Extract the underlying expression of level 0 and 1

```
    val of_expr : 'a Bddapron.Expr2.expr -> 'a t
    val to_expr : 'a t -> 'a Bddapron.Expr2.expr
```

Conversion from/to general expression

```
    val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
```

Extend the underlying environment to a superenvironment, and adapt accordingly the underlying representation

```
    val is_false : 'a t -> bool
    val is_true : 'a t -> bool
    val print : Format.formatter -> 'a t -> unit
  end

module List :
  sig
    type 'a t = ('a Bddapron.Cond.t, 'a Bddapron.Expr1.List.t) Bdd.Cond.value
    val of_lexpr0 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a Bddapron.Expr0.t list -> 'a t
```

Creation from a list of expressions of level 0 (without environment)

```
    val of_lexpr1 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'a Bddapron.Expr1.t list -> 'a t
```

Creation from a list of expressions of level 1 (without condition environment)

```
    val of_listexpr1 :
      ?normalize:bool ->
      ?reduce:bool ->
      ?careset:bool ->
      'a Bddapron.Cond.t -> 'a Bddapron.Expr1.List.t -> 'a t
```

Creation from an expression list of level 1 (without condition environment)

```
    val get_env : 'a t -> 'a Bddapron.Env.t
    val get_cond : 'a t -> 'a Bddapron.Cond.t
```

Extract resp. the environment and the condition environment

```
    val to_lexpr0 : 'a t -> 'a Bddapron.Expr0.t list
    val to_listexpr1 : 'a t -> 'a Bddapron.Expr1.List.t
    val to_lexpr1 : 'a t -> 'a Bddapron.Expr1.t list
```

Extract the underlying list of expressions of level 0 and 1

```
    val extend_environment : 'a t -> 'a Bddapron.Env.t -> 'a t
    val print : Format.formatter -> 'a t -> unit
  end

end
```

Chapter 24

Module Descend: Recursive descend on sets of diagrams (internal)

```
module Descend :
  sig
    val texpr_cofactor :
      ('a Bddapron.Expr0.t -> 'b -> 'a Bddapron.Expr0.t) ->
      'a Bddapron.Expr0.t array -> 'b -> 'a Bddapron.Expr0.t array
    val texpr_support :
      ('a,
       ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'b)
       Bddapron.Env.O.t)
      Bddapron.Cond.O.t -> 'a Bddapron.Expr0.t array -> Cudd.Man.v Cudd.Bdd.t
    val texpr_cofactors :
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'd)
      Bddapron.Env.O.t ->
      'a Bddapron.Expr0.t array ->
      int -> 'a Bddapron.Expr0.t array * 'a Bddapron.Expr0.t array
    val split_lvar :
      'a Bdd.Env.symbol ->
      'a list -> 'a Bddapron.Expr0.t list -> 'a list * Apron.Var.t array
    val split_texpr :
      'a Bddapron.Expr0.t array ->
      Cudd.Man.v Bdd.Expr0.t list * 'a Bddapron.ApronexprDD.t array
    val split_lvarlexpr :
      'a Bdd.Env.symbol ->
      'a list ->
      'a Bddapron.Expr0.t list ->
      'a list * Cudd.Man.v Bdd.Expr0.t list * Apron.Var.t array *
      'a Bddapron.ApronexprDD.t array
    val cofactors :
      'a Bddapron.ApronDD.man ->
      ('b, [> 'b Bddapron.Env.typ ] as 'c, [> 'b Bddapron.Env.typdef ] as 'd, 'e)
      Bddapron.Env.O.t ->
      ('b, ('b, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'a Bddapron.ApronDD.t -> int -> 'a Bddapron.ApronDD.t * 'a Bddapron.ApronDD.t
    val descend_mtbdd :
      'a Bddapron.ApronDD.man ->
      ('b, [> 'b Bddapron.Env.typ ] as 'c, [> 'b Bddapron.Env.typdef ] as 'd, 'e)
```

```
    Bddapron.Env.O.t ->
    ('b, ('b, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
    ('a Bddapron.ApronDD.t -> 'b Bddapron.Expr0.t array -> 'a Bddapron.ApronDD.t) ->
    'a Bddapron.ApronDD.t -> 'b Bddapron.Expr0.t array -> 'a Bddapron.ApronDD.t
```

```
  val descend :
    cudd:'c Cudd.Man.t ->
    maxdepth:int ->
    nocare:('a -> bool) ->
    cube_of_down:('a -> 'c Cudd.Bdd.t) ->
    cofactor:('a -> 'c Cudd.Bdd.t -> 'a) ->
    select:('a -> int) ->
    terminal:(depth:int ->
              newcube:'c Cudd.Bdd.t -> cube:'c Cudd.Bdd.t -> down:'a -> 'b option) ->
    ite:(depth:int ->
         newcube:'c Cudd.Bdd.t ->
         cond:int -> dthen:'b option -> delse:'b option -> 'b option) ->
    down:'a -> 'b option
```

Obsolete, moved in Bdd.Decompose[9]

```
end
```

Chapter 25

Module Mtbdddomain0: Boolean/Numerical domain, with MTBDDs over APRON values

```
module Mtbdddomain0 :
  sig
```

```
type ('a, 'b) man = 'b Bddapron.ApronDD.man
```

BDDAPRON Manager. The type parameter 'b indicates the underlying APRON abstract domain, as in type 'b Apron.Abstract1.t

```
type 'b t = 'b Bddapron.ApronDD.t
```

BDDAPRON Abstract value.

```
val make_man : ?global:bool -> 'b Apron.Manager.t -> ('a, 'b) man
```

Makes a BDDAPRON manager from an APRON manager. If `global=true` (default: `false`), uses a global (persistent) BDD cache for the operations `is_leq`, `join`, `meet` and `exist` (internal).

```
val size : ('a, 'b) man -> 'b t -> int
```

Size of an abstract value in terms of number of nodes of the MTBDD.

```
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  'a Bddapron.Env.t -> Format.formatter -> 'b t -> unit
```

Printing function

25.1 Constructors, accessors, tests and property extraction

25.1.1 Basic constructor

```
val bottom : ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b t
val top : ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b t
```

```
val of_apron :
  ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b Apron.Abstract0.t -> 'b t
val of_bddapron :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
  'b t
```

25.1.2 Tests

```
val is_bottom : ('a, 'b) man -> 'b t -> bool
val is_top : ('a, 'b) man -> 'b t -> bool
```

Emtpiness and Universality tests

```
val is_leq : ('a, 'b) man ->
  'b t -> 'b t -> bool
val is_eq : ('a, 'b) man ->
  'b t -> 'b t -> bool
```

Inclusion and equality tests

25.1.3 Extraction of properties

```
val to_bddapron :
  ('a, 'b) man ->
  'b t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list
```

Conversion to a disjunction of a conjunction of pair of a purely Boolean formula (without numerical constraints) and an APRON abstract value

25.2 Operations

```
val meet : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
val join : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
```

Meet and join

```
val meet_condition :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a Bddapron.Expr0.Bool.t -> 'b t
```

Intersection with a Boolean expression (that may involve numerical constraints)

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'b t option -> 'b t
val substitute_lexpr :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'b t option -> 'b t
```

Parallel assignement/substitution of a list of variables by a list of expressions

```
val forget_list :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'b t -> 'a list -> 'b t
```

Forget (existential quantification) a list of variables

```
val widening : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
val widening_threshold :
  ('a, 'b) man ->
  'b t ->
  'b t ->
  Apron.Lincons0.t array -> 'b t
```

Widening

```
val apply_change :
  bottom:'b t ->
  ('a, 'b) man ->
  'b t ->
  Bddapron.Env.change -> 'b t
val apply_permutation :
  ('a, 'b) man ->
  'b t ->
  int array option * Apron.Dim.perm option -> 'b t
```

25.3 Opened signature and Internal functions

We provide here the same functions and modules as before, but with opened types (this allows extensions). The functions above are actually derived from the functions below by just constraining their types. We provide here also more internal functions

```
module O :
  sig
```

```
val meet_idcondb :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
  Bddapron.Env.O.t ->
  ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'b Bddapron.Mtbdddomain0.t -> int * bool -> 'b Bddapron.Mtbdddomain0.t
val size :
  ('a, 'b) Bddapron.Mtbdddomain0.man -> 'b Bddapron.Mtbdddomain0.t -> int
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> Format.formatter -> 'b Bddapron.Mtbdddomain0.t -> unit
val bottom :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Bddapron.Mtbdddomain0.t
val top :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Bddapron.Mtbdddomain0.t
val of_apron :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Apron.Abstract0.t -> 'b Bddapron.Mtbdddomain0.t
val of_bddapron :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
  'b Bddapron.Mtbdddomain0.t
val is_bottom :
  ('a, 'b) Bddapron.Mtbdddomain0.man -> 'b Bddapron.Mtbdddomain0.t -> bool
val is_top :
  ('a, 'b) Bddapron.Mtbdddomain0.man -> 'b Bddapron.Mtbdddomain0.t -> bool
val is_leq :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  'b Bddapron.Mtbdddomain0.t -> 'b Bddapron.Mtbdddomain0.t -> bool
val is_eq :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  'b Bddapron.Mtbdddomain0.t -> 'b Bddapron.Mtbdddomain0.t -> bool
val to_bddapron :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  'b Bddapron.Mtbdddomain0.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list
val meet :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  'b Bddapron.Mtbdddomain0.t ->
  'b Bddapron.Mtbdddomain0.t -> 'b Bddapron.Mtbdddomain0.t
val join :
  ('a, 'b) Bddapron.Mtbdddomain0.man ->
  'b Bddapron.Mtbdddomain0.t ->
  'b Bddapron.Mtbdddomain0.t -> 'b Bddapron.Mtbdddomain0.t
```

```
    val meet_condition :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
      Bddapron.Env.O.t ->
      ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'b Bddapron.Mtbdddomain0.t ->
      'a Bddapron.Expr0.Bool.t -> 'b Bddapron.Mtbdddomain0.t
    val assign_lexpr :
      ?relational:bool ->
      ?nodependency:bool ->
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
      Bddapron.Env.O.t ->
      ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'b Bddapron.Mtbdddomain0.t ->
      'a list ->
      'a Bddapron.Expr0.t list ->
      'b Bddapron.Mtbdddomain0.t option -> 'b Bddapron.Mtbdddomain0.t
    val substitute_lexpr :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
      Bddapron.Env.O.t ->
      ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
      'b Bddapron.Mtbdddomain0.t ->
      'a list ->
      'a Bddapron.Expr0.t list ->
      'b Bddapron.Mtbdddomain0.t option -> 'b Bddapron.Mtbdddomain0.t
    val forget_list :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t ->
      'b Bddapron.Mtbdddomain0.t -> 'a list -> 'b Bddapron.Mtbdddomain0.t
    val widening :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      'b Bddapron.Mtbdddomain0.t ->
      'b Bddapron.Mtbdddomain0.t -> 'b Bddapron.Mtbdddomain0.t
    val widening_threshold :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      'b Bddapron.Mtbdddomain0.t ->
      'b Bddapron.Mtbdddomain0.t ->
      Apron.Lincons0.t array -> 'b Bddapron.Mtbdddomain0.t
    val apply_change :
      bottom:'b Bddapron.Mtbdddomain0.t ->
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      'b Bddapron.Mtbdddomain0.t ->
      Bddapron.Env.change -> 'b Bddapron.Mtbdddomain0.t
    val apply_permutation :
      ('a, 'b) Bddapron.Mtbdddomain0.man ->
      'b Bddapron.Mtbdddomain0.t ->
      int array option * Apron.Dim.perm option -> 'b Bddapron.Mtbdddomain0.t
  end

end
```

Chapter 26

Module Bddleaf: Manipulation of lists of guards and leafs (internal)

```
module Bddleaf :
  sig
```

```
    type ('a, 'b) elt = {
      guard : 'a Cudd.Bdd.t ;
      leaf : 'b ;
    }
```

```
    type ('a, 'b) t = ('a, 'b) elt list
```

26.1 Utilities

```
    val fold2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
```

```
    val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
```

Applies f to all pairs `(elt1,elt2)` with `elt1` in `list1` and `elt2` in `list2`. Iterates first of the first list, then on the second.

26.2 Normalisation

```
    val check_unicity : is_equal:('b -> 'b -> bool) -> ('a, 'b) elt list -> bool
```

Checking function: raises `Failure` if problem, returns `true` otherwise.

Checks that

- no guard is false
- no abstract value is bottom
- no duplicatas of abstract values

```
    val check_disjointness : ('a, 'b) elt list -> bool
```

Checking function: raises `Failure` if problem, returns `true` otherwise. Checks that the guards are exclusive.

```
    val cons_unique :
      is_equal:('b -> 'b -> bool) ->
      ('a, 'b) elt ->
      ('a, 'b) elt list -> ('a, 'b) elt list
```

Performs the join of a list with an element.

Assuming that the list argument satisfies the unicity property, ensures it in the result

```
val append_unique :
  is_equal:('b -> 'b -> bool) ->
  ('a, 'b) elt list ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Append the two lists.

Assuming that the first list argument satisfies the unicity property, ensures it in the result

```
val cons_disjoint :
  is_equal:('b -> 'b -> bool) ->
  merge:('b -> 'b -> 'b) ->
  ('a, 'b) elt ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Performs the join of a list with an element.

Assuming that the list argument satisfies the disjointness (and unicity) property, ensures it in the result

```
val append_disjoint :
  is_equal:('b -> 'b -> bool) ->
  merge:('b -> 'b -> 'b) ->
  ('a, 'b) elt list ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Appends the two lists.

Assuming that the first list argument satisfies the disjointness (and unicity) property, ensures it in the result

```
val cons :
  is_equal:('b -> 'b -> bool) ->
  merge:('b -> 'b -> 'b) ->
  unique:bool ->
  disjoint:bool ->
  ('a, 'b) elt ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Calls the right cons function depending on the options.

```
val append :
  is_equal:('b -> 'b -> bool) ->
  merge:('b -> 'b -> 'b) ->
  unique:bool ->
  disjoint:bool ->
  ('a, 'b) elt list ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Calls the right append function depending on the options.

```
val make_unique :
  is_equal:('b -> 'b -> bool) ->
  merge:('b -> 'b -> 'b) ->
  disjoint:bool ->
  ('a, 'b) elt list -> ('a, 'b) elt list
```

Remove duplicatas (by reconstructing the list)

```
val guard : cudd:'a Cudd.Man.t -> ('a, 'b) t -> 'a Cudd.Bdd.t
```

26.3 Others

Return the union of guards in the list

```
end
```

Chapter 27

Module Bdddomain0: Combined Boolean/Numerical domain, with lists of BDDs and APRON values

```
module Bdddomain0 :
  sig
```

```
    type ('a, 'b) man = {
      apron : 'b Apron.Manager.t ;
      mutable bdd_restrict : Cudd.Bdd.vt -> Cudd.Bdd.vt -> Cudd.Bdd.vt ;
      mutable expr_restrict : 'a Bddapron.Expr0.t -> Cudd.Bdd.vt -> 'a Bd-
    dapron.Expr0.t ;
      mutable meet_disjoint : bool ;
      mutable join_disjoint : bool ;
      mutable meet_cond_unique : bool ;
      mutable meet_cond_disjoint : bool ;
      mutable meet_cond_depth : int ;
      mutable assign_unique : bool ;
      mutable assign_disjoint : bool ;
      mutable substitute_unique : bool ;
      mutable substitute_disjoint : bool ;
      mutable forget_unique : bool ;
      mutable forget_disjoint : bool ;
      mutable change_environment_unique : bool ;
      mutable change_environment_disjoint : bool ;
    }
```

BDDAPRON Manager. The type parameter 'b indicates the underlying APRON abstract domain, as in type 'b Apron.Abstract1.t, and 'a is the type of symbols.

```
    type 'b elt = (Cudd.Man.v, 'b Apron.Abstract0.t) Bddapron.Bddleaf.elt
    type 'b t = {
      mutable list : 'b elt list ;
      bottom : 'b elt ;
      mutable unique : bool ;
      mutable disjoint : bool ;
    }
```

Abstract value.

```
    val make_man : 'b Apron.Manager.t -> ('a, 'b) man
```

Makes a BDDAPRON manager from an APRON manager, and fills options with default values

```
val canonicalize :
  ?apron:bool ->
  ?unique:bool ->
  ?disjoint:bool ->
  ('a, 'b) man -> 'b t -> unit
```

Canonicalize an abstract value by ensuring uniqueness and disjointness properties. If `apron` is true, then also normalize APRON abstract values. By default: `apron=false, unique=disjoint=true`.

```
val size : ('a, 'b) man -> 'b t -> int
```

Size of an abstract value in terms of number of nodes of the MTBDD.

```
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  'a Bddapron.Env.t -> Format.formatter -> 'b t -> unit
```

Printing function

27.1 Constructors, accessors, tests and property extraction

27.1.1 Basic constructor

```
val bottom : ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b t
val top : ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b t
val of_apron :
  ('a, 'b) man ->
  'a Bddapron.Env.t -> 'b Apron.Abstract0.t -> 'b t
val of_bddapron :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
  'b t
```

27.1.2 Tests

```
val is_bottom : ('a, 'b) man -> 'b t -> bool
val is_top : ('a, 'b) man -> 'b t -> bool
```

Emtpiness and Universality tests

```
val is_leq : ('a, 'b) man ->
  'b t -> 'b t -> bool
val is_eq : ('a, 'b) man ->
  'b t -> 'b t -> bool
```

Inclusion and equality tests

27.1.3 Extraction of properties

```
val to_bddapron :
  ('a, 'b) man ->
  'b t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list
```

Conversion to a disjunction of a conjunction of pair of a purely Boolean formula (without numerical constraints) and an APRON abstract value

27.2 Operations

```
val meet : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
val join : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
```

Meet and join

```
val meet_condition :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a Bddapron.Expr0.Bool.t -> 'b t
```

Intersection with a Boolean expression (that may involve numerical constraints)

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'b t option -> 'b t
val substitute_lexpr :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'b t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'b t option -> 'b t
```

Parallel assignement/substitution of a list of variables by a list of expressions

```
val forget_list :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'b t -> 'a list -> 'b t
```

Forget (existential quantification) a list of variables

```
val widening : ('a, 'b) man ->
  'b t ->
  'b t -> 'b t
val widening_threshold :
  ('a, 'b) man ->
  'b t ->
  'b t ->
  Apron.Lincons0.t array -> 'b t
```

Widening

```
val apply_change :
  bottom:'b t ->
  ('a, 'b) man ->
  'b t -> Bddapron.Env.change -> 'b t
val apply_permutation :
  ('a, 'b) man ->
  'b t ->
  int array option * Apron.Dim.perm option -> 'b t
```

27.3 Opened signature and Internal functions

```
module O :
  sig
    module L :
      sig
        val is_bottom : 'a Apron.Manager.t -> 'a Bddapron.Bdddomain0.elt -> bool
        val meet_cube :
          'a Apron.Manager.t ->
          ('b, [> 'b Bddapron.Env.typ ] as 'c, [> 'b Bddapron.Env.typdef ] as 'd, 'e)
          Bddapron.Env.O.t ->
          ('b, ('b, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'a Bddapron.Bdddomain0.elt -> Cudd.Bdd.vt -> 'a Bddapron.Bdddomain0.elt
        val forget :
          'a Apron.Manager.t ->
          'a Bddapron.Bdddomain0.elt ->
          Cudd.Bdd.vt -> Apron.Dim.t array -> 'a Bddapron.Bdddomain0.elt
      end
    val check_wellformed :
      ('a, 'b) Bddapron.Bdddomain0.man -> 'b Bddapron.Bdddomain0.t -> bool
    val canonicalize :
      ?apron:bool ->
      ?unique:bool ->
      ?disjoint:bool ->
      ('a, 'b) Bddapron.Bdddomain0.man -> 'b Bddapron.Bdddomain0.t -> unit
    val size :
      ('a, 'b) Bddapron.Bdddomain0.man -> 'b Bddapron.Bdddomain0.t -> int
    val print :
      ?print_apron:((int -> string) ->
                    Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
      ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
      Bddapron.Env.O.t -> Format.formatter -> 'b Bddapron.Bdddomain0.t -> unit
```

```
val bottom :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Bddapron.Bdddomain0.t

val top :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Bddapron.Bdddomain0.t

val of_apron :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t -> 'b Apron.Abstract0.t -> 'b Bddapron.Bdddomain0.t

val of_bddapron :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
  Bddapron.Env.O.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
  'b Bddapron.Bdddomain0.t

val is_bottom :
  ('a, 'b) Bddapron.Bdddomain0.man -> 'b Bddapron.Bdddomain0.t -> bool

val is_top :
  ('a, 'b) Bddapron.Bdddomain0.man -> 'b Bddapron.Bdddomain0.t -> bool

val is_leq :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  'b Bddapron.Bdddomain0.t -> 'b Bddapron.Bdddomain0.t -> bool

val is_eq :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  'b Bddapron.Bdddomain0.t -> 'b Bddapron.Bdddomain0.t -> bool

val to_bddapron :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  'b Bddapron.Bdddomain0.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list

val meet :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  'b Bddapron.Bdddomain0.t ->
  'b Bddapron.Bdddomain0.t -> 'b Bddapron.Bdddomain0.t

val join :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  'b Bddapron.Bdddomain0.t ->
  'b Bddapron.Bdddomain0.t -> 'b Bddapron.Bdddomain0.t

val meet_condition :
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
  Bddapron.Env.O.t ->
  ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
  'b Bddapron.Bdddomain0.t ->
  'a Bddapron.Expr0.Bool.t -> 'b Bddapron.Bdddomain0.t

val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) Bddapron.Bdddomain0.man ->
  ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
  Bddapron.Env.O.t ->
```

```
          ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'b Bddapron.Bdddomain0.t ->
          'a list ->
          'a Bddapron.Expr0.t list ->
          'b Bddapron.Bdddomain0.t option -> 'b Bddapron.Bdddomain0.t
        val substitute_lexpr :
          ('a, 'b) Bddapron.Bdddomain0.man ->
          ('a, [> 'a Bddapron.Env.typ ] as 'c, [> 'a Bddapron.Env.typdef ] as 'd, 'e)
          Bddapron.Env.O.t ->
          ('a, ('a, 'c, 'd, 'e) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
          'b Bddapron.Bdddomain0.t ->
          'a list ->
          'a Bddapron.Expr0.t list ->
          'b Bddapron.Bdddomain0.t option -> 'b Bddapron.Bdddomain0.t
        val forget_list :
          ('a, 'b) Bddapron.Bdddomain0.man ->
          ('a, [> 'a Bddapron.Env.typ ], [> 'a Bddapron.Env.typdef ], 'e)
          Bddapron.Env.O.t ->
          'b Bddapron.Bdddomain0.t -> 'a list -> 'b Bddapron.Bdddomain0.t
        val widening :
          ('a, 'b) Bddapron.Bdddomain0.man ->
          'b Bddapron.Bdddomain0.t ->
          'b Bddapron.Bdddomain0.t -> 'b Bddapron.Bdddomain0.t
        val widening_threshold :
          ('a, 'b) Bddapron.Bdddomain0.man ->
          'b Bddapron.Bdddomain0.t ->
          'b Bddapron.Bdddomain0.t ->
          Apron.Lincons0.t array -> 'b Bddapron.Bdddomain0.t
        val apply_change :
          bottom:'b Bddapron.Bdddomain0.t ->
          ('a, 'b) Bddapron.Bdddomain0.man ->
          'b Bddapron.Bdddomain0.t -> Bddapron.Env.change -> 'b Bddapron.Bdddomain0.t
        val apply_permutation :
          ('a, 'b) Bddapron.Bdddomain0.man ->
          'b Bddapron.Bdddomain0.t ->
          int array option * Apron.Dim.perm option -> 'b Bddapron.Bdddomain0.t
      end

end
```

Chapter 28

Module Domain0: Boolean/Numerical domain: generic interface

```
module Domain0 :
  sig
```

28.1 Generic interface

28.1.1 Types

```
type ('a, 'b, 'c, 'd) man = {
  typ : string ;
  man : 'c ;
  canonicalize : ?apron:bool -> 'c -> 'd -> unit ;
  size : 'c -> 'd -> int ;
  print : ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  'a Bddapron.Env.t -> Format.formatter -> 'd -> unit ;
  bottom : 'c -> 'a Bddapron.Env.t -> 'd ;
  top : 'c -> 'a Bddapron.Env.t -> 'd ;
  of_apron : 'c -> 'a Bddapron.Env.t -> 'b Apron.Abstract0.t -> 'd ;
  of_bddapron : 'c ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list -> 'd ;
  is_bottom : 'c -> 'd -> bool ;
  is_top : 'c -> 'd -> bool ;
  is_leq : 'c -> 'd -> 'd -> bool ;
  is_eq : 'c -> 'd -> 'd -> bool ;
  to_bddapron : 'c -> 'd -> ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ;
  meet : 'c -> 'd -> 'd -> 'd ;
  join : 'c -> 'd -> 'd -> 'd ;
  meet_condition : 'c ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'd ;
  assign_lexpr : ?relational:bool ->
  ?nodependency:bool ->
  'c ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'd -> 'a list -> 'a Bddapron.Expr0.t list -> 'd option -> 'd ;
```

```
  substitute_lexpr : 'c ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'd -> 'a list -> 'a Bddapron.Expr0.t list -> 'd option -> 'd ;
  forget_list : 'c -> 'a Bddapron.Env.t -> 'd -> 'a list -> 'd ;
  widening : 'c -> 'd -> 'd -> 'd ;
  widening_threshold : 'c -> 'd -> 'd -> Apron.Lincons0.t array -> 'd ;
  apply_change : bottom:'d -> 'c -> 'd -> Bddapron.Env.change -> 'd ;
  apply_permutation : 'c -> 'd -> int array option * Apron.Dim.perm option -> 'd ;
}
```

Type of generic managers.

- 'a: type of symbols
- 'b: as in 'b Apron.Manager.t (Box.t, Polka.strict Polka.t, etc);
- 'c: type of the underlying manager;
- 'd: type of the underlying abstract values of level 0.

```
type 'd t = 'd
```

Type of generic abstract values

28.1.2 Functions

```
val canonicalize : ?apron:bool ->
  ('a, 'b, 'c, 'd) man -> 'd t -> unit
val size : ('a, 'b, 'c, 'd) man -> 'd t -> int
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> Format.formatter -> 'd t -> unit
val bottom : ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> 'd t
val top : ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> 'd t
val of_apron :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> 'b Apron.Abstract0.t -> 'd t
val of_bddapron :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
  'd t
val is_bottom : ('a, 'b, 'c, 'd) man -> 'd t -> bool
val is_top : ('a, 'b, 'c, 'd) man -> 'd t -> bool
val is_leq : ('a, 'b, 'c, 'd) man ->
  'd t -> 'd t -> bool
val is_eq : ('a, 'b, 'c, 'd) man ->
  'd t -> 'd t -> bool
val to_bddapron :
  ('a, 'b, 'c, 'd) man ->
  'd t ->
  ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list
```

```
val meet : ('a, 'b, 'c, 'd) man ->
  'd t -> 'd t -> 'd t
val join : ('a, 'b, 'c, 'd) man ->
  'd t -> 'd t -> 'd t
val meet_condition :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'd t -> 'a Bddapron.Expr0.Bool.t -> 'd t
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'd t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'd t option -> 'd t
val substitute_lexpr :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'd t ->
  'a list ->
  'a Bddapron.Expr0.t list ->
  'd t option -> 'd t
val forget_list :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  'd t -> 'a list -> 'd t
val widening : ('a, 'b, 'c, 'd) man ->
  'd t -> 'd t -> 'd t
val widening_threshold :
  ('a, 'b, 'c, 'd) man ->
  'd t ->
  'd t -> Apron.Lincons0.t array -> 'd t
val apply_change :
  bottom:'d t ->
  ('a, 'b, 'c, 'd) man ->
  'd t -> Bddapron.Env.change -> 'd t
val apply_permutation :
  ('a, 'b, 'c, 'd) man ->
  'd t ->
  int array option * Apron.Dim.perm option -> 'd t
val man_get_apron : ('a, 'b, 'c, 'd) man -> 'b Apron.Manager.t
```

28.2 Implementation based on Bddapron.Mtbdddomain0[25]

```
type ('a, 'b) mtbdd = ('a, 'b, ('a, 'b) Bddapron.Mtbdddomain0.man, 'b Bddapron.Mtbdddomain0.t)
  man
val mtbdd_of_mtbdddomain :
  ('a, 'b) Bddapron.Mtbdddomain0.man -> ('a, 'b) mtbdd
```

Make a mtbdd manager from an underlying BDDAPRON manager

```
val make_mtbdd : ?global:bool -> 'b Apron.Manager.t -> ('a, 'b) mtbdd
```

Make a mtbdd manager from an APRON manager

28.2.1 Type conversion functions

```
val man_is_mtbdd : ('a, 'b, 'c, 'd) man -> bool
```

Return `true` iff the argument manager is a mtbdd manager

```
val man_of_mtbdd : ('a, 'b) mtbdd -> ('a, 'b, 'c, 'd) man
```

Makes a mtbdd manager generic

```
val man_to_mtbdd : ('a, 'b, 'c, 'd) man -> ('a, 'b) mtbdd
```

Instanciate the type of a mtbdd manager. Raises `Failure` if the argument manager is not a mtbdd manager

```
val of_mtbdd :
  ('a, 'b) mtbdd *
  'b Bddapron.Mtbdddomain0.t t ->
  ('a, 'b, 'c, 'd) man * 'd t
```

Makes a pair (mtbdd manager,mtbdd abstract value) generic

```
val to_mtbdd :
  ('a, 'b, 'c, 'd) man * 'd t ->
  ('a, 'b) mtbdd *
  'b Bddapron.Mtbdddomain0.t t
```

Instanciate the type of a pair (mtbdd manager,mtbdd abstract value). Raises `Failure` if the argument manager is not a mtbdd manager

28.3 Implementation based on `Bddapron.Bdddomain0`[27]

```
type ('a, 'b) bdd = ('a, 'b, ('a, 'b) Bddapron.Bdddomain0.man, 'b Bddapron.Bdddomain0.t)
  man
val bdd_of_bdddomain : ('a, 'b) Bddapron.Bdddomain0.man -> ('a, 'b) bdd
```

Make a bdd manager from an underlying BDDAPRON manager

```
val make_bdd : 'b Apron.Manager.t -> ('a, 'b) bdd
```

Make a bdd manager from an APRON manager

28.3.1 Type conversion functions

```
val man_is_bdd : ('a, 'b, 'c, 'd) man -> bool
```

Return `true` iff the argument manager is a bdd manager

```
val man_of_bdd : ('a, 'b) bdd -> ('a, 'b, 'c, 'd) man
```

Makes a bdd manager generic

```
val man_to_bdd : ('a, 'b, 'c, 'd) man -> ('a, 'b) bdd
```

Instanciate the type of a bdd manager. Raises `Failure` if the argument manager is not a bdd manager

```
val of_bdd :
  ('a, 'b) bdd * 'b Bddapron.Bdddomain0.t t ->
  ('a, 'b, 'c, 'd) man * 'd t
```

Makes a pair (bdd manager,bdd abstract value) generic

```
val to_bdd :
  ('a, 'b, 'c, 'd) man * 'd t ->
  ('a, 'b) bdd * 'b Bddapron.Bdddomain0.t t
```

Instanciate the type of a pair (bdd manager,bdd abstract value). Raises `Failure` if the argument manager is not a bdd manager

```
end
```

Chapter 29

Module Domainlevel1: Functor to transform an abstract domain interface from level 0 to level 1 (internal)

```
module Domainlevel1 :
  sig
```

```
    module type Level0 =
      sig
```

```
        type ('a, 'b) man
```

BDDAPRON Manager. The type parameter 'b indicates the underlying APRON abstract domain, as in type 'b Apron.Abstract0.t, and 'a the type of symbols.

```
        type 'b t
```

BDDAPRON Abstract value. The type parameter 'b indicates the underlying APRON abstract domain, as in type 'b Apron.Abstract0.t, and 'a the type of symbols.

```
        val size : ('a, 'b) man ->
          'b t -> int
        val print :
          ?print_apron:((int -> string) ->
                        Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
          'a Bddapron.Env.t ->
          Format.formatter -> 'b t -> unit
        val bottom : ('a, 'b) man ->
          'a Bddapron.Env.t -> 'b t
        val top : ('a, 'b) man ->
          'a Bddapron.Env.t -> 'b t
        val of_apron :
          ('a, 'b) man ->
          'a Bddapron.Env.t ->
          'b Apron.Abstract0.t -> 'b t
        val of_bddapron :
          ('a, 'b) man ->
          'a Bddapron.Env.t ->
```

```
    ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list ->
    'b t
  val is_bottom : ('a, 'b) man ->
    'b t -> bool
  val is_top : ('a, 'b) man ->
    'b t -> bool
  val is_leq : ('a, 'b) man ->
    'b t ->
    'b t -> bool
  val is_eq : ('a, 'b) man ->
    'b t ->
    'b t -> bool
  val to_bddapron :
    ('a, 'b) man ->
    'b t ->
    ('a Bddapron.Expr0.Bool.t * 'b Apron.Abstract0.t) list
  val meet : ('a, 'b) man ->
    'b t ->
    'b t -> 'b t
  val join : ('a, 'b) man ->
    'b t ->
    'b t -> 'b t
  val meet_condition :
    ('a, 'b) man ->
    'a Bddapron.Env.t ->
    'a Bddapron.Cond.t ->
    'b t ->
    'a Bddapron.Expr0.Bool.t -> 'b t
  val assign_lexpr :
    ?relational:bool ->
    ?nodependency:bool ->
    ('a, 'b) man ->
    'a Bddapron.Env.t ->
    'a Bddapron.Cond.t ->
    'b t ->
    'a list ->
    'a Bddapron.Expr0.t list ->
    'b t option -> 'b t
  val substitute_lexpr :
    ('a, 'b) man ->
    'a Bddapron.Env.t ->
    'a Bddapron.Cond.t ->
    'b t ->
    'a list ->
    'a Bddapron.Expr0.t list ->
    'b t option -> 'b t
  val forget_list :
    ('a, 'b) man ->
    'a Bddapron.Env.t ->
    'b t ->
    'a list -> 'b t
  val widening : ('a, 'b) man ->
    'b t ->
```

```
    'b t -> 'b t
  val widening_threshold :
    ('a, 'b) man ->
    'b t ->
    'b t ->
    Apron.Lincons0.t array -> 'b t
  val apply_change :
    bottom:'b t ->
    ('a, 'b) man ->
    'b t ->
    Bddapron.Env.change -> 'b t
  val apply_permutation :
    ('a, 'b) man ->
    'b t ->
    int array option * Apron.Dim.perm option -> 'b t
end
```

29.1 Abstract domain of level 1

```
module type Level1 =
  sig
```

```
    type ('a, 'b) man
```

BDDAPRON Manager. The type parameter 'b indicates the underlying APRON abstract domain, as in type 'b Apron.Abstract0.t, and 'a the type of symbols.

```
    type 'b t0
```

Level 0 abstract value.

```
    type ('a, 'b) t = ('a Bddapron.Env.t, 'b t0) Bddapron.Env.value
```

Level 1 abstract value

```
    val get_env : ('a, 'b) t -> 'a Bddapron.Env.t
    val to_level0 : ('a, 'b) t -> 'b t0
    val of_level0 : 'a Bddapron.Env.t ->
      'b t0 -> ('a, 'b) t
    val size : ('a, 'b) man ->
      ('a, 'b) t -> int
```

Size of an abstract value.

```
    val print :
      ?print_apron:((int -> string) ->
                    Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
      Format.formatter -> ('a, 'b) t -> unit
```

Printing function

```
    val bottom : ('a, 'b) man ->
      'a Bddapron.Env.t -> ('a, 'b) t
```

29.1.1 Basic constructor

```
val top : ('a, 'b) man ->
  'a Bddapron.Env.t -> ('a, 'b) t
val of_apron :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  'b Apron.Abstract1.t -> ('a, 'b) t
val of_bddapron :
  ('a, 'b) man ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr1.Bool.t * 'b Apron.Abstract1.t) list ->
  ('a, 'b) t
```

29.1.2 Tests

```
val is_bottom : ('a, 'b) man ->
  ('a, 'b) t -> bool
val is_top : ('a, 'b) man ->
  ('a, 'b) t -> bool
```

Emtpiness and Universality tests

```
val is_leq : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t -> bool
val is_eq : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t -> bool
```

Inclusion and equality tests

```
val to_bddapron :
  ('a, 'b) man ->
  ('a, 'b) t ->
  ('a Bddapron.Expr1.Bool.t * 'b Apron.Abstract1.t) list
```

29.1.3 Extraction of properties

Conversion to a disjunction of a conjunction of pair of a purely Boolean formula (without numerical constraints) and an APRON abstract value

29.1.4 Operations

```
val meet : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t ->
  ('a, 'b) t
val join : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t ->
  ('a, 'b) t
```

Meet and join

```
val meet_condition :
  ('a, 'b) man ->
```

```
  'a Bddapron.Cond.t ->
  ('a, 'b) t ->
  'a Bddapron.Expr1.Bool.t -> ('a, 'b) t
val meet_condition2 :
  ('a, 'b) man ->
  ('a, 'b) t ->
  'a Bddapron.Expr2.Bool.t -> ('a, 'b) t
```

Intersection with a Boolean expression (that may involve numerical constraints)

```
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) man ->
  'a Bddapron.Cond.t ->
  ('a, 'b) t ->
  'a list ->
  'a Bddapron.Expr1.t list ->
  ('a, 'b) t option ->
  ('a, 'b) t
val assign_listexpr2 :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b) man ->
  ('a, 'b) t ->
  'a list ->
  'a Bddapron.Expr2.List.t ->
  ('a, 'b) t option ->
  ('a, 'b) t
val substitute_lexpr :
  ('a, 'b) man ->
  'a Bddapron.Cond.t ->
  ('a, 'b) t ->
  'a list ->
  'a Bddapron.Expr1.t list ->
  ('a, 'b) t option ->
  ('a, 'b) t
val substitute_listexpr2 :
  ('a, 'b) man ->
  ('a, 'b) t ->
  'a list ->
  'a Bddapron.Expr2.List.t ->
  ('a, 'b) t option ->
  ('a, 'b) t
```

Parallel assignement/substitution of a list of variables by a list of expressions

```
val forget_list : ('a, 'b) man ->
  ('a, 'b) t ->
  'a list -> ('a, 'b) t
```

Forget (existential quantification) a list of variables

```
val widening : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t ->
  ('a, 'b) t
val widening_threshold :
```

```
('a, 'b) man ->
('a, 'b) t ->
('a, 'b) t ->
Apron.Lincons1.earray -> ('a, 'b) t
```

Widening

```
val change_environment :
  ('a, 'b) man ->
  ('a, 'b) t ->
  'a Bddapron.Env.t -> ('a, 'b) t
```

29.1.5 Change of environments and renaming

Change the environment (eliminate (forget) variables not belonging to the new environment, and introduce new variables)

```
val rename : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a * 'a) list -> ('a, 'b) t
```

Rename a list of variables (thus changing the environment).

```
val unify : ('a, 'b) man ->
  ('a, 'b) t ->
  ('a, 'b) t ->
  ('a, 'b) t
```

Unify two abstract values on their least common environment (lce, that should exist, which implies that no variable is defined with different types in the two initial environments).
This is equivalent to change the environment to the lce, and to perform meet.

```
  end

module Make :

functor (Level0 :  Level0) -> Level1  with type ('a,'b) man = ('a,'b) Level0.man
and type 'b t0 = 'b Level0.t

end
```

Chapter 30

Module `Mtbdddomain1`: Boolean/Numerical domain with normalized environment

```
module Mtbdddomain1 :
  sig
```

```
    val make_man :
      ?global:bool -> 'b Apron.Manager.t -> ('a, 'b) Bddapron.Mtbdddomain0.man
```

Makes a BDDAPRON manager from an APRON manager. If `global=true` (default: `false`), uses a global (persistent) BDD cache for the operations `is_leq`, `join`, `meet` and `exist` (internal).

```
    include Bddapron.Domainlevel1.Level1
  end
```

Chapter 31

Module Bdddomain1: Boolean/Numerical domain with normalized environment

```
module Bdddomain1 :
  sig
```

```
    include Bddapron.Domainlevel1.Level1
```

```
    val make_man : 'b Apron.Manager.t -> ('a, 'b) man
```

Makes a BDDAPRON manager from an APRON manager, and fills options with default values

```
    val canonicalize :
      ?apron:bool ->
      ?unique:bool -> ?disjoint:bool -> ('a, 'b) man -> ('a, 'b) t -> unit
```

Canonicalize an abstract value by ensuring uniqueness and disjointness properties. If `apron` is true, then also normalize APRON abstract values. By default: `apron=false, unique=disjoint=true`.

```
  end
```

Chapter 32

Module Domain1: Boolean/Numerical domain with normalized environment

```
module Domain1 :
  sig
```

32.1 Generic interface

32.1.1 Types

```
type ('a, 'b, 'c, 'd) man = ('a, 'b, 'c, 'd) Bddapron.Domain0.man
```

Type of generic managers.

- 'a: type of symbols
- 'b: as in 'b Apron.Manager.t (Box.t, Polka.strict Polka.t, etc);
- 'c: type of the underlying manager;
- 'd: type of the underlying abstract values of level 0.

```
type ('a, 'b) mtbdd = ('a, 'b, ('a, 'b) Bddapron.Mtbdddomain0.man, 'b Bddapron.Mtbdddomain0.t)
  man
type ('a, 'b) bdd = ('a, 'b, ('a, 'b) Bddapron.Bdddomain0.man, 'b Bddapron.Bdddomain0.t)
  man
type ('a, 'd) t = ('a Bddapron.Env.t, 'd) Bddapron.Env.value
```

Type of generic abstract values

```
val canonicalize :
  ?apron:bool ->
  ('a, 'b, 'c, 'd) man -> ('a, 'd) t -> unit
val print :
  ?print_apron:((int -> string) ->
                Format.formatter -> 'b Apron.Abstract0.t -> unit) ->
  ('a, 'b, 'c, 'd) man ->
  Format.formatter -> ('a, 'd) t -> unit
val get_env : ('a, 'd) t -> 'a Bddapron.Env.t
val to_level0 : ('a, 'd) t -> 'd
val of_level0 : 'a Bddapron.Env.t -> 'd -> ('a, 'd) t
```

```
val size : ('a, 'b, 'c, 'd) man -> ('a, 'd) t -> int
val bottom : ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> ('a, 'd) t
val top : ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> ('a, 'd) t
val of_apron :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t -> 'b Apron.Abstract1.t -> ('a, 'd) t
val of_bddapron :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Env.t ->
  ('a Bddapron.Expr1.Bool.t * 'b Apron.Abstract1.t) list ->
  ('a, 'd) t
val is_bottom : ('a, 'b, 'c, 'd) man -> ('a, 'd) t -> bool
val is_top : ('a, 'b, 'c, 'd) man -> ('a, 'd) t -> bool
val is_leq : ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t -> ('a, 'd) t -> bool
val is_eq : ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t -> ('a, 'd) t -> bool
val to_bddapron :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a Bddapron.Expr1.Bool.t * 'b Apron.Abstract1.t) list
val meet : ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a, 'd) t -> ('a, 'd) t
val join : ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a, 'd) t -> ('a, 'd) t
val meet_condition :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Cond.t ->
  ('a, 'd) t ->
  'a Bddapron.Expr1.Bool.t -> ('a, 'd) t
val meet_condition2 :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  'a Bddapron.Expr2.Bool.t -> ('a, 'd) t
val assign_lexpr :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Cond.t ->
  ('a, 'd) t ->
  'a list ->
  'a Bddapron.Expr1.t list ->
  ('a, 'd) t option -> ('a, 'd) t
val assign_listexpr2 :
  ?relational:bool ->
  ?nodependency:bool ->
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  'a list ->
```

```
  'a Bddapron.Expr2.List.t ->
  ('a, 'd) t option -> ('a, 'd) t
val substitute_lexpr :
  ('a, 'b, 'c, 'd) man ->
  'a Bddapron.Cond.t ->
  ('a, 'd) t ->
  'a list ->
  'a Bddapron.Expr1.t list ->
  ('a, 'd) t option -> ('a, 'd) t
val substitute_listexpr2 :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  'a list ->
  'a Bddapron.Expr2.List.t ->
  ('a, 'd) t option -> ('a, 'd) t
val forget_list :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t -> 'a list -> ('a, 'd) t
val widening :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a, 'd) t -> ('a, 'd) t
val widening_threshold :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a, 'd) t ->
  Apron.Lincons1.earray -> ('a, 'd) t
val change_environment :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  'a Bddapron.Env.t -> ('a, 'd) t
val unify :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t ->
  ('a, 'd) t -> ('a, 'd) t
val rename :
  ('a, 'b, 'c, 'd) man ->
  ('a, 'd) t -> ('a * 'a) list -> ('a, 'd) t
val man_get_apron : ('a, 'b, 'c, 'd) man -> 'b Apron.Manager.t
```

32.2 Implementation based on Bddapron.Mtbdddomain0[25]

```
val mtbdd_of_mtbdddomain :
  ('a, 'b) Bddapron.Mtbdddomain0.man -> ('a, 'b) mtbdd
```

Make a mtbdd manager from an underlying BDDAPRON manager

```
val make_mtbdd : ?global:bool -> 'b Apron.Manager.t -> ('a, 'b) mtbdd
```

Make a mtbdd manager from an APRON manager

32.2.1 Type conversion functions

```
val man_is_mtbdd : ('a, 'b, 'c, 'd) man -> bool
```

Return `true` iff the argument manager is a mtbdd manager

```
val man_of_mtbdd : ('a, 'b) mtbdd -> ('a, 'b, 'c, 'd) man
```

Makes a mtbdd manager generic

```
val man_to_mtbdd : ('a, 'b, 'c, 'd) man -> ('a, 'b) mtbdd
```

Instanciate the type of a mtbdd manager. Raises `Failure` if the argument manager is not a mtbdd manager

```
val of_mtbdd :
  ('a, 'b) mtbdd *
  ('a, 'b Bddapron.Mtbdddomain0.t) t ->
  ('a, 'b, 'c, 'd) man * ('a, 'd) t
```

Makes a pair (mtbdd manager,mtbdd abstract value) generic

```
val to_mtbdd :
  ('a, 'b, 'c, 'd) man * ('a, 'd) t ->
  ('a, 'b) mtbdd *
  ('a, 'b Bddapron.Mtbdddomain0.t) t
```

Instanciate the type of a pair (mtbdd manager,mtbdd abstract value). Raises `Failure` if the argument manager is not a mtbdd manager

32.3 Implementation based on `Bddapron.Bdddomain0`[27]

```
val bdd_of_bdddomain : ('a, 'b) Bddapron.Bdddomain0.man -> ('a, 'b) bdd
```

Make a bdd manager from an underlying BDDAPRON manager

```
val make_bdd : 'b Apron.Manager.t -> ('a, 'b) bdd
```

Make a bdd manager from an APRON manager

32.3.1 Type conversion functions

```
val man_is_bdd : ('a, 'b, 'c, 'd) man -> bool
```

Return `true` iff the argument manager is a bdd manager

```
val man_of_bdd : ('a, 'b) bdd -> ('a, 'b, 'c, 'd) man
```

Makes a bdd manager generic

```
val man_to_bdd : ('a, 'b, 'c, 'd) man -> ('a, 'b) bdd
```

Instanciate the type of a bdd manager. Raises `Failure` if the argument manager is not a bdd manager

```
val of_bdd :
  ('a, 'b) bdd *
  ('a, 'b Bddapron.Bdddomain0.t) t ->
  ('a, 'b, 'c, 'd) man * ('a, 'd) t
```

Makes a pair (bdd manager,bdd abstract value) generic

```
val to_bdd :
  ('a, 'b, 'c, 'd) man * ('a, 'd) t ->
  ('a, 'b) bdd *
  ('a, 'b Bddapron.Bdddomain0.t) t
```

Instanciate the type of a pair (bdd manager,bdd abstract value). Raises `Failure` if the argument manager is not a bdd manager

```
end
```

Chapter 33

Module Formula: Extra-operations on formula

```
module Formula :
  sig
    module O :
      sig
        module Expr0 :
          sig
            module Bool :
              sig
                val to_lconjunction :
                  ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
                dapron.Env.typdef ] as 'c, 'd)
                  Bddapron.Env.O.t ->
                  ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
                  'f Bddapron.Expr0.Bool.t ->
                  ('f Bddapron.Expr0.Bool.t * 'f Bddapron.Expr0.Bool.t) list
                val forget :
                  ('a, 'b, 'c, 'd) Bddapron.Domain0.man ->
                  ('e, [> 'e Bddapron.Env.typ ] as 'f, [> 'e Bd-
                dapron.Env.typdef ] as 'g, 'h)
                  Bddapron.Env.O.t ->
                  ('e, ('e, 'f, 'g, 'h) Bddapron.Env.O.t) Bddapron.Cond.O.t ->
                  'j Bddapron.Expr0.Bool.t -> 'e list -> 'j Bddapron.Expr0.Bool.t
              end
          end
        module Expr1 :
          sig
            module Bool :
              sig
                val to_lconjunction :
                  ('a,
                   ('a, [> 'a Bddapron.Env.typ ] as 'b, [> 'a Bd-
                dapron.Env.typdef ] as 'c, 'd)
                     Bddapron.Env.O.t)
```

```
                Bddapron.Cond.O.t ->
                ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
                (('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t *
                 ('a, ('a, 'b, 'c, 'd) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t)
                list
              val forget :
                ('a, 'b, 'c, 'd) Bddapron.Domain0.man ->
                ('e,
                 ('e, [> 'e Bddapron.Env.typ ] as 'f, [> 'e Bd-
              dapron.Env.typdef ] as 'g, 'h)
                 Bddapron.Env.O.t)
                Bddapron.Cond.O.t ->
                ('e, ('e, 'f, 'g, 'h) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t ->
                'e list -> ('e, ('e, 'f, 'g, 'h) Bddapron.Env.O.t) Bddapron.Expr1.O.Bool.t
            end

        end

    end

  module Expr0 :
    sig

      module Bool :
        sig

          val to_lconjunction :
            'a Bddapron.Env.t ->
            'a Bddapron.Cond.t ->
            'a Bddapron.Expr0.Bool.t ->
            ('a Bddapron.Expr0.Bool.t * 'a Bddapron.Expr0.Bool.t) list
          val forget :
            ('a, 'b, 'c, 'd) Bddapron.Domain0.man ->
            'e Bddapron.Env.t ->
            'e Bddapron.Cond.t ->
            'e Bddapron.Expr0.Bool.t -> 'e list -> 'e Bddapron.Expr0.Bool.t
        end

    end

  module Expr1 :
    sig

      module Bool :
        sig

          val to_lconjunction :
            'a Bddapron.Cond.t ->
            'a Bddapron.Expr1.Bool.t ->
            ('a Bddapron.Expr1.Bool.t * 'a Bddapron.Expr1.Bool.t) list
          val forget :
            ('a, 'b, 'c, 'd) Bddapron.Domain0.man ->
            'e Bddapron.Cond.t ->
            'e Bddapron.Expr1.Bool.t -> 'e list -> 'e Bddapron.Expr1.Bool.t
        end

    end

end
```

Chapter 34

Module `Policy`: Policies for BDDAPRON abstract values

```
module Policy :
  sig
```

This module provides a generic BddApron lift for policies on Apron abstract values. Policies are available only for MTBDD abstract values (`Domain0.t` built on `Mtbdddomain0.t`).

```
val apron_policy_print :
  'a Apron.Policy.man ->
  'b Bddapron.Env.t -> Format.formatter -> 'a Apron.Policy.t -> unit
module Dnf :
  sig
    type 'a t = 'a Bddapron.Cond.cond Bdd.Normalform.dnf
```

Disjunction of conjuction of conditions. `(c1 and c2) or c3` is represented by `[|[c1,c2],[c3]|]`. Unicity of the representation is guaranteed by a lexicographic ordering based on the order on conditions.

```
  end
module DDDnf :
  sig
    type 'a t = 'a Bddapron.Policy.Dnf.t Cudd.Mtbdd.t
    type 'a table = 'a Bddapron.Policy.Dnf.t Cudd.Mtbdd.table
  end
module DPolicy :
  sig
    type 'a t = {
      hash : int ;
      dpolicy : 'a Apron.Policy.t Bdd.Normalform.disjunction ;
    }
    type 'a table = 'a t Cudd.Mtbdd.table
  end
module PMtbdddomain0 :
  sig
```

```
    type ('a, 'b) man = {
      man : ('a, 'b) Bddapron.Mtbdddomain0.man ;
      papron : 'b Apron.Policy.man ;
      ptable : 'b Bddapron.Policy.DPolicy.table ;
      betable : 'a Bddapron.Policy.DDDnf.table ;
      symbol : 'a Bddapron.Env.symbol ;
    }
    type 'a t = 'a Bddapron.Policy.DPolicy.t Cudd.Mtbdd.t
  end

module PDomain0 :
  sig

    type ('a, 'b, 'c, 'd, 'e, 'f) man = {
      man : ('a, 'b, 'c, 'd) Bddapron.Domain0.man ;
      pman : 'e ;
      print : 'e ->
      'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> Format.formatter -> 'f -> unit ;
      meet_condition_apply : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'd ;
      meet_condition_improve : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f option -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'f ;
    }
  end
```

34.1 Policy, level 1

```
module Domain1 :
  sig

    type ('a, 'b, 'c, 'd, 'e, 'f) man = ('a, 'b, 'c, 'd, 'e, 'f) Bd-
    dapron.Policy.PDomain0.man = {
      man : ('a, 'b, 'c, 'd) Bddapron.Domain0.man ;
      pman : 'e ;
      print : 'e ->
      'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> Format.formatter -> 'f -> unit ;
      meet_condition_apply : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'd ;
      meet_condition_improve : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f option -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'f ;
    }
```

Type of generic policy managers.

- 'a: type of symbols
- 'b: as in 'b Apron.Manager.t (Box.t, Polka.strict Polka.t, etc);
- 'c: type of the underlying manager;
- 'd: type of the underlying abstract values of level 0.
- 'e: type of the underlying policy manager
- 'f: type of the underlying policy

```
    type ('a, 'b) mtbdd = ('a, 'b, ('a, 'b) Bddapron.Mtbdddomain0.man, 'b Bd-
    dapron.Mtbdddomain0.t,
       ('a, 'b) Bddapron.Policy.PMtbdddomain0.man,
       'b Bddapron.Policy.PMtbdddomain0.t)
      man
    val manager_get_manager :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      ('a, 'b, 'c, 'd) Bddapron.Domain0.man
    val print :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> Format.formatter -> 'f -> unit
    val meet_condition_apply :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      'a Bddapron.Cond.t ->
      'f ->
      ('a, 'd) Bddapron.Domain1.t ->
      'f Bddapron.Expr1.Bool.t -> ('a, 'd) Bddapron.Domain1.t
    val meet_condition_improve :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      'a Bddapron.Cond.t ->
      'f option -> ('a, 'd) Bddapron.Domain1.t -> 'f Bddapron.Expr1.Bool.t -> 'f
    val meet_condition2_apply :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      'f ->
      ('a, 'd) Bddapron.Domain1.t ->
      'a Bddapron.Expr2.Bool.t -> ('a, 'd) Bddapron.Domain1.t
    val meet_condition2_improve :
      ('a, 'b, 'c, 'd, 'e, 'f) man ->
      'f option -> ('a, 'd) Bddapron.Domain1.t -> 'a Bddapron.Expr2.Bool.t -> 'f
  end
```

34.2 Policy, level 0

```
module Domain0 :
  sig
    type ('a, 'b, 'c, 'd, 'e, 'f) man = ('a, 'b, 'c, 'd, 'e, 'f) Bd-
    dapron.Policy.PDomain0.man = {
      man : ('a, 'b, 'c, 'd) Bddapron.Domain0.man ;
      pman : 'e ;
      print : 'e ->
      'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> Format.formatter -> 'f -> unit ;
      meet_condition_apply : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'd ;
      meet_condition_improve : 'e ->
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t -> 'f option -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'f ;
    }
```

Type of generic policy managers.

- 'a: type of symbols
- 'b: as in 'b Apron.Manager.t (Box.t, Polka.strict Polka.t, etc);

- 'c: type of the underlying manager;
- 'd: type of the underlying abstract values of level 0.
- 'e: type of the underlying policy manager
- 'f: type of the underlying policy

```
type ('a, 'b) mtbdd = ('a, 'b, ('a, 'b) Bddapron.Mtbdddomain0.man, 'b Bd-
dapron.Mtbdddomain0.t,
   ('a, 'b) Bddapron.Policy.PMtbdddomain0.man,
   'b Bddapron.Policy.PMtbdddomain0.t)
  man
val manager_get_manager :
  ('a, 'b, 'c, 'd, 'e, 'f) man ->
  ('a, 'b, 'c, 'd) Bddapron.Domain0.man
val print :
  ('a, 'b, 'c, 'd, 'e, 'f) man ->
  'a Bddapron.Env.t -> 'a Bddapron.Cond.t -> Format.formatter -> 'f -> unit
val meet_condition_apply :
  ('a, 'b, 'c, 'd, 'e, 'f) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'f -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'd
val meet_condition_improve :
  ('a, 'b, 'c, 'd, 'e, 'f) man ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'f option -> 'd -> 'a Bddapron.Expr0.Bool.t -> 'f
val make_mtbdd :
  ?global:bool ->
  symbol:'a Bddapron.Env.symbol ->
  'b Apron.Policy.man -> ('a, 'b) mtbdd
end
```

34.3 Policy, level 0, MTBDD implementation

```
module Mtbdddomain0 :
  sig
    type ('a, 'b) man = ('a, 'b) Bddapron.Policy.PMtbdddomain0.man = {
      man : ('a, 'b) Bddapron.Mtbdddomain0.man ;
      papron : 'b Apron.Policy.man ;
      ptable : 'b Bddapron.Policy.DPolicy.table ;
      betable : 'a Bddapron.Policy.DDDnf.table ;
      symbol : 'a Bddapron.Env.symbol ;
    }
    type 'a t = 'a Bddapron.Policy.PMtbdddomain0.t
    val manager_get_manager :
      ('a, 'b) man ->
      ('a, 'b) Bddapron.Mtbdddomain0.man
    val make_man :
      ?global:bool ->
      symbol:'a Bddapron.Env.symbol ->
      'b Apron.Policy.man -> ('a, 'b) man
    val equal : 'a ->
      'b t ->
```

```
          'b t -> bool
        val print :
          ('a, 'b) man ->
          'c Bddapron.Env.t ->
          'c Bddapron.Cond.t ->
          Format.formatter -> 'b t -> unit
        val meet_condition_apply :
          ('a, 'b) man ->
          'a Bddapron.Env.t ->
          'a Bddapron.Cond.t ->
          'b t ->
          'b Bddapron.Mtbdddomain0.t ->
          'a Bddapron.Expr0.Bool.t -> 'b Bddapron.Mtbdddomain0.t
        val meet_condition_improve :
          ('a, 'b) man ->
          'a Bddapron.Env.t ->
          'a Bddapron.Cond.t ->
          'b t option ->
          'b Bddapron.Mtbdddomain0.t ->
          'a Bddapron.Expr0.Bool.t -> 'b t
      end

  end
```

Chapter 35

Module Syntax: Building BDDAPRON expressions from Abstract Syntax Trees

```
module Syntax :
  sig
```

35.1 Types

```
type cst = [ `Apron of Apron.Coeff.t | `Bint of (bool * int) * int | `Bool of bool ]
```

Constant

```
type unop = [ `Apron of Apron.Texpr1.unop * Apron.Texpr1.typ * Apron.Texpr1.round | `Not ]
```

Unary operators

```
type bbinop = [ `And | `EQ | `GEQ | `GT | `LEQ | `LT | `NEQ | `Or ]
```

Boolean/finite-type binary operators
Binary operators

```
type binop = [ `Apron of Apron.Texpr1.binop * Apron.Texpr1.typ * Apron.Texpr1.round
  | `Bool of bbinop ]
type 'a expr = [ `Binop of
      binop * 'a expr * 'a expr
  | `Cst of cst
  | `If of
      'a expr * 'a expr *
      'a expr
  | `In of 'a expr * 'a expr list
  | `Ref of 'a
  | `Unop of unop * 'a expr ]
```

Expressions

35.2 Error and printing functions

```
val print_cst : Format.formatter -> cst -> unit
```

```
val print_unop : Format.formatter -> unop -> unit
val print_bbinop : Format.formatter -> bbinop -> unit
val print_binop : Format.formatter -> binop -> unit
val print_expr :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a expr -> unit
exception Error of string
```

35.3 Translation functions

Exception raised in case of typing error

```
val to_expr0 :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a expr -> 'a Bddapron.Expr0.t
val to_expr1 :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a expr -> 'a Bddapron.Expr1.t
val to_listexpr1 :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a expr list -> 'a Bddapron.Expr1.List.t
val to_listexpr2 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
  'a expr list -> 'a Bddapron.Expr2.List.t
val to_boolexpr2 :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> 'a expr -> 'a Bddapron.Expr2.Bool.t
```

35.4 Internal functions

```
val error : ('a, Format.formatter, unit, 'b) Pervasives.format4 -> 'a
val is_zero : 'a expr -> bool
val precedence_of_unop : unop -> int
val precedence_of_binop : binop -> int
val precedence_of_expr : 'a expr -> int
val cst_to_expr0 :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t -> [< cst ] -> 'a Bddapron.Expr0.expr
val apply_bbinop :
  'a Bddapron.Env.t ->
  'a Bddapron.Cond.t ->
```

```
      bbinop ->
      'a Bddapron.Expr0.expr -> 'a Bddapron.Expr0.expr -> 'a Bddapron.Expr0.Bool.t
    val apply_binop :
      'a Bddapron.Env.t ->
      'a Bddapron.Cond.t ->
      binop ->
      'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t -> 'a Bddapron.Expr0.t
end
```

Chapter 36

Module Yacc

```
module Yacc :
  sig

     type token =
       | TK_LBRACKET
       | TK_RBRACKET
       | TK_SEMICOLON
       | TK_COLON
       | TK_LPAR
       | TK_RPAR
       | TK_LBRACE
       | TK_RBRACE
       | TK_BOOL
       | TK_UINT
       | TK_SINT
       | TK_INT
       | TK_REAL
       | TK_IN
       | TK_COMMA
       | TK_TYPEDEF
       | TK_ENUM
       | TK_IF
       | TK_THEN
       | TK_ELSE
       | TK_VERTEX
       | TK_RAY
       | TK_LINE
       | TK_MOD
       | TK_RAYMOD
       | TK_LINEMOD
       | TK_MUL of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_ADD of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_SUB of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_DIV of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_MODULO of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_CAST of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_SQRT of (Apron.Texpr1.typ * Apron.Texpr1.round)
       | TK_MPQF of Mpqf.t
       | TK_FLOAT of float
       | TK_LEQ
       | TK_GEQ
```

```
      | TK_LT
      | TK_GT
      | TK_EQ
      | TK_NEQ
      | TK_AND
      | TK_OR
      | TK_NOT
      | TK_ID of string
      | TK_TRUE
      | TK_FALSE
      | TK_EOF
   val expr :
     (Lexing.lexbuf -> token) ->
     Lexing.lexbuf -> string Bddapron.Syntax.expr
end
```

Chapter 37

Module Lex

```
module Lex :
  sig
     exception Error of int * int
     val lex : Lexing.lexbuf -> Bddapron.Yacc.token
  end
```

Chapter 38

Module Parser: Parsing BDDAPRON expressions from strings (or lexing buffers)

```
module Parser :
  sig
```

The grammar is indicated below

38.1 From strings

```
val expr0_of_string :
  string Bddapron.Env.t ->
  string Bddapron.Cond.t -> string -> string Bddapron.Expr0.t
val expr1_of_string :
  string Bddapron.Env.t ->
  string Bddapron.Cond.t -> string -> string Bddapron.Expr1.t
val listexpr1_of_lstring :
  string Bddapron.Env.t ->
  string Bddapron.Cond.t -> string list -> string Bddapron.Expr1.List.t
val listexpr2_of_lstring :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  string Bddapron.Env.t ->
  string Bddapron.Cond.t -> string list -> string Bddapron.Expr2.List.t
val boolexpr2_of_string :
  ?normalize:bool ->
  ?reduce:bool ->
  ?careset:bool ->
  string Bddapron.Env.t ->
  string Bddapron.Cond.t -> string -> string Bddapron.Expr2.Bool.t
```

38.2 Misc.

```
val expr0_of_lexbuf :
  string Bddapron.Env.t ->
```

```
  string Bddapron.Cond.t -> Lexing.lexbuf -> string Bddapron.Expr0.t
```

38.3 Grammar of expressions

```
  <expr> ::= <bexpr>      Boolean expression
   |  <iexpr>      bounded integer expression
   |  <eexpr>      enumerated type expression
   |  <nexpr>      numerical expression

  Boolean expressions
  <bexpr> ::= true | false
    | id                            variable
    | <constraint>
    | not <bexpr>
    | <bexpr> (or | and) <bexpr>
    | ( <bexpr> )
    | if <bexpr> then <bexpr> else <bexpr>
  <constraint> ::= id == <expr>
    <iexpr> (== | >= | > | <= | <) <iexpr>
    <nexpr> (== | >= | > | <= | <) <nexpr>

  Bounded integer expressions
  <iexpr> ::= uint[<integer>][-]<integer>  constant (snd integer) of given type
    |  sint[<integer>][-]<integer>  constant (snd integer) of given type
    |  id                           variable
    |  <iexpr> ( +|-|* ) <iexpr>
    |  ( <iexpr> )
    |  if <bexpr> then <iexpr> else <iexpr>

  Enumerated type expressions
  <eexpr> ::= id                          variable or constant (label)
    |  ( <eexpr> )
    |  if <bexpr> then <eexpr> else <eexpr>

  Numerical expressions
  <nexpr> ::= <coeff>
    |  id                           variable
    |  <unop> <nexpr>
    |  <nexpr> <binop> <nexpr>
    |  ( <nexpr> )
    |  if <bexpr> then <nexpr> else <nexpr>

  <binop>    ::= (+|-|*|/|%)[_(i|f|d|l|q)[,(n|0|+oo|-oo)]]
  <unop>     ::= -
       |  (cast|sqrt)[_(i|f|d|l|q)[,(n|0|+oo|-oo)]]
  <coeff>    ::= <float>
       |  <rational>
  <float>    ::= C/OCaml floating-point number syntax
  <rational> ::= <integer>  | <integer>/<integer>
```

Here are is an example (`bddapron/test2.ml` file).

```
open Bddapron;;

  let cudd = Cudd.Man.make_v ();;
```

```
      let env = Env.make cudd;;
      let cond = Cond.make cudd;;
      Env.add_typ_with "e0" (`Benum [|"l0";"l1";"l2"|]);;
      Env.add_vars_with [
        ("b0",`Bool);
        ("b1",`Bool);
        ("i0",`Bint(false,3));
        ("i1",`Bint(false,3));
        ("e0",`Benum("e0"));
        ("x0",`Real);
        ("x1",`Real);
        ("x2",`Real);
      ];;

      let bexpr1 = Parser.boolexpr2_of_string env cond
        "(i0==uint[3](3) + i1 or e0==(if b0 then l0 else l1)) and x0 +_f,oo 2 *_i,0 x1 >=0";;

      let apron = Polka.manager_alloc_loose ();;
      let man = Domain1.make_man ~global:false apron;;
      let top = Domain1.top man env;;
      let abs = Domain0.meet_condition man cond top bexpr1;;


  end
```