MLCUDDIDL: OCaml interface for CUDD library, version 2.2.0, 01/02/11

Bertrand Jeannet

February 2, 2011

Contents

Part I

Module `Cudd` : Interface to CUDD library

Chapter 1

Introduction

User modules:

- `Cudd.Man`[5]: CUDD managers;
- `Cudd.Bdd`[6]: CUDD BDDs;
- `Cudd.Add`[15]: CUDD ADDs;
- `Cudd.Mtbdd`[11], `Cudd.Mtbddc`[12]: MTBDDs on OCaml values;
- `Cudd.Mapleaf`[14], `Cudd.User`[13]: maps user operations from leaves to MTBDDs on such leaves.
- `Cudd.Memo`[4], relying on `Cudd.Hash`[2] and `Cudd.Cache`[3]: allows the control of memoization techniques, for permutation and vector composition functions on BDDs and MTBDDs, and user operations.

Internal modules:

- `Cudd.Vdd`[7]: MTBDDs on unhashed OCaml values;
- `Cudd.Custom`[8]: for user operations on ADDs and MTBDDs;
- `Cudd.Weakke`[9] and `Cudd.PWeakke`[10]: for polymorphic weak hashtables.

This library provides an OCAML interface to the CUDD BDD library[`http://vlsi.colorado.edu/software.html`], as well as additional C functions to CUDD (in cuddauxXXX files). The reader is supposed to have looked at the user's manual[`http://vlsi.colorado.edu/~fabio/CUDD/`] of this library.

Most functions of the CUDD library are interfaced; with the exception of ZDDs functions. If you need it, please tell me, I can do it quickly.

1.1 Memory management

The diagrams are implemented as abstract types, and more precisely as OCAML *custom objects*. These objects contain both the manager which owns the diagram and the diagram itself. They are garbage collected by the OCAML garbage collection. The effect of the OCAML garbage collection is to decrease the reference count of the diagram if it has become unreachable from the OCAML heap, and to remove the OCAML custom object from the OCAML heap. Later, the CUDD may possibly garbage the diagram in the C heap, if its reference count is zero.

For technical reasons, CUDD managers come in two different flavors in the OCaml interface: one dedicated to BDDs and standard CUDD ADDs (Algebraic Decision Diagrams, with C *d*ouble values at leaves) , which has the type `Man.d Man.t`, and one dedicated to BDDs and so-called VDDs, with OCaml *v*alues at leaves., which has the type `Man.v Man.t`, see `Cudd.Man.d`[5], `Cudd.Man.v`[5] and `Cudd.Man.t`[5].

For efficiency reasons, it is better to link in some way the two garbage collectors. So, when the CUDD garbage collector is triggered, in a normal situation (during the creation of a new node) or because of a reordering operation, it first calls the OCAML garbage collector, in order to be able to garbage collect as many nodes as possible.

The function `Cudd.Man.set_gc`[5.1] allows to tune the OCAML garbage collection of diagrams and the link with the CUDD garbage collection.

It is possible to apply to the diagrams the polymorphic comparison test (`Pervasives.compare`, from which are derived `=,<=,>=,<,>`) and polymorphic hash function (polymorphic `Hashtbl.hash`). The comparison function compares lexicographically the pair `address of the manager, address of the node`). The hash function returns the address of the node.

1.2 This document

Each module is described separately. For each Ocaml function, we indicate below in typewriter font the CUDD function to which it corresponds, whenever possible. If the order of the arguments has been changed, we usually specify "variation of" before.

We do not describe in detail the functions which have a direct CUDD equivalent. Instead, we refer the user to the original CUDD documentation.

1.3 Organization of the code

The interface has been written with the help of the CamlIDL tool, the input files of which are suffixed with `.idl`. CamlIDL automatizes most of the cumbersome task of writing stub codes and converting datatypes back and forth between C and OCAML. However, as we implemented more than a direct interface, we also used the M4 preprocessor on `.idl` files to simplify the task (instead of the default cpp C preprocessor).

`.idl` files are thus filtered through M4 and transformed according to the macro file `macros.m4`, then CamlIDL generates from them four files, suffixed with `.c`, `.h`, `.ml` and `.mli`.

`cudd_caml.c`, `cudd_caml.h` `custom_caml.c` and `custom_caml.h` are not generated from a `.idl` file and contain code common to all the other files.

The normal user doesn't need to understand this process, as the library is distributed with all the C and OCAML files already generated.

1.4 Installation and Use

See the `README` file.

You need:

- GNU Make[`http://www.gnu.org/software/make`]
- OBJECTIVE CAML 3.0[`http://caml.inria.fr`]
- CAMLIDL 1.05[`http://caml.inria.fr/camlidl`]
- FINDLIB[`http://projects.camlcity.org/projects/findlib.html`]
- M4 preprocessor, SED, GREP GNU versions[`http://www.gnu.org/software`]

CUDD BDD library[`http://vlsi.colorado.edu/software.html`] is now included in the distribution.

Flags should be properly set in `Makefile.config` (starting from `Makefile.config.model`).

Also, the Make rules for some `example.ml` file shows how to compile a program with the interface.

Chapter 2

Module Hash: User hashtables

```
module Hash :
  sig
```

```
    type t
```

Abstract type for user hashtables

```
    val _create : int -> int -> t
    val table : t Weak.t Pervasives.ref
```

Internal table

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

`create ~size:n arity` create a hashtable of arity `arity`, of the optional size `n`

```
    val arity : t -> int
```

Returns the arity of the hashtable

```
    val clear : t -> unit
```

Clears the content of the hashtable

```
    val clear_all : unit -> unit
```

Clears the content of all created hashtables

```
  end
```

Chapter 3

Module `Cache`: Local caches

```
module Cache :
  sig
```

```
    type t
```

Abstract type for local caches

```
    val _create : int -> int -> int -> t
    val create : ?size:int -> ?maxsize:int -> arity:int -> t
    val create1 : ?size:int -> ?maxsize:int -> unit -> t
    val create2 : ?size:int -> ?maxsize:int -> unit -> t
    val create3 : ?size:int -> ?maxsize:int -> unit -> t
```

Creates local caches of the given arity, with initial size `size` and maximal size `maxsize`.

```
    val arity : t -> int
```

Returns the arity of the local cache.

```
    val clear : t -> unit
```

Clears the content of the local cache.

```
  end
```

Chapter 4

Module Memo: Memoization policy

```
module Memo :
  sig
```

```
    type memo_discr =
      | Global
```

CUDD global cache (arity no more than 2, currently)

```
      | Cache
      | Hash
    type t =
      | Global
```

CUDD global cache (arity no more than 2, currently)

```
      | Cache of Cudd.Cache.t
```

CUDD local cache

```
      | Hash of Cudd.Hash.t
```

CUDD local hash table

It is up to the user to clear regularly such a table. Forgetting to do so will prevent garbage collection of nodes stored in the table, which can only grows, unlike local caches.

Type of memoization table

```
    val clear : t -> unit
  end
```

Chapter 5

Module Man: CUDD Manager

```
module Man :
  sig
```

```
type d
```

Indicates that a CUDD manager manipulates standard ADDs with leaves of type C double

```
type v
```

Indicates that a CUDD manager manipulates "custom" ADDs with leaves of type an `OCaml` value, see modules `Cudd.Mtbdd`[11] and `Cudd.Mtbddc`[12]. A manager cannot manipulate the two types of ADDs (for garbage collection reasons)

```
type 'a t
```

Type of CUDD managers, where `'a` is either `d` or `v`

```
type reorder =
  | REORDER_SAME
  | REORDER_NONE
  | REORDER_RANDOM
  | REORDER_RANDOM_PIVOT
  | REORDER_SIFT
  | REORDER_SIFT_CONVERGE
  | REORDER_SYMM_SIFT
  | REORDER_SYMM_SIFT_CONV
  | REORDER_WINDOW2
  | REORDER_WINDOW3
  | REORDER_WINDOW4
  | REORDER_WINDOW2_CONV
  | REORDER_WINDOW3_CONV
  | REORDER_WINDOW4_CONV
  | REORDER_GROUP_SIFT
  | REORDER_GROUP_SIFT_CONV
  | REORDER_ANNEALING
  | REORDER_GENETIC
  | REORDER_LINEAR
  | REORDER_LINEAR_CONVERGE
  | REORDER_LAZY_SIFT
  | REORDER_EXACT
```

Reordering method.

```
type aggregation =
  | NO_CHECK
  | GROUP_CHECK
  | GROUP_CHECK2
  | GROUP_CHECK3
  | GROUP_CHECK4
  | GROUP_CHECK5
  | GROUP_CHECK6
  | GROUP_CHECK7
  | GROUP_CHECK8
  | GROUP_CHECK9
```

Type of aggregation methods.

```
type lazygroup =
  | LAZY_NONE
  | LAZY_SOFT_GROUP
  | LAZY_HARD_GROUP
  | LAZY_UNGROUP
```

Group type for lazy sifting.

```
type vartype =
  | VAR_PRIMARY_INPUT
  | VAR_PRESENT_STATE
  | VAR_NEXT_STATE
```

Variable type. Currently used only in lazy sifting.

```
type mtr =
  | MTR_DEFAULT
  | MTR_FIXED
```

Is variable order inside group fixed or not ?

```
type error =
  | NO_ERROR
  | MEMORY_OUT
  | TOO_MANY_NODES
  | MAX_MEM_EXCEEDED
  | INVALID_ARG
  | INTERNAL_ERROR
```

Type of error when CUDD raises an exception.

```
type dt = d t
type vt = v t
```

Shortcuts

```
type tbool =
  | False
  | True
  | Top
```

Ternary Boolean type, used to defines minterms where `Top` means `True` or `False`

```
val string_of_reorder : reorder -> string
val string_of_error : error -> string
```

Printing functions

5.1 Global settings

```
val print_limit : int Pervasives.ref
```

Parameter for printing functions: specify the maximum number of minterms to be printed. Above this numbers, only statistics on the BDD is printed.

```
val set_gc : int -> (unit -> unit) -> (unit -> unit) -> unit
```

`set_gc max gc reordering` performs several things:

- It sets the ratio used/max for BDDs abstract values to `1/max` (see the OCaml manual for details). 1 000 000 is a good value.
- It also sets for all the future managers that will be created the hook function to be called before a CUDD garbage collection, and the hook function to be called before a CUDD reordering. You may typically specify a OCaml garbage collection function for both hooks, in order to make OCaml dereference unused nodes, thus allowing CUDD to remove them. Default values are `Gc.full_major()` for both hooks.

```
val srandom : int -> unit
```

`Cudd_Srandom`[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Srandom]. Initializes the seed for the CUDD rnadom number generator (used in a number of functions, like `Cudd.Bdd.pick_cubes_on_support`[6.11].

5.2 Managers

```
val _make : bool -> int -> int -> int -> int -> int -> 'a t
```

Internal, do not use !

```
val make_d :
  ?numVars:int ->
  ?numVarsZ:int ->
  ?numSlots:int ->
  ?cacheSize:int -> ?maxMemory:int -> unit -> d t
val make_v :
  ?numVars:int ->
  ?numVarsZ:int ->
  ?numSlots:int ->
  ?cacheSize:int -> ?maxMemory:int -> unit -> v t
```

Variation of `Cudd_Init`[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Init]. `make_d ~numVars ~numVarsZ ~numSlots ~cacheSize ~maxMemory ()` creates a manager with the given parameters. `make_d ()` is OK. In addition, the function sets a hook function to be called whenever a CUDD garbage collection occurs, and a (dummy) hook function to be called whenever a CUDD reordering occurs. The defaults can be modified with `Cudd.Man.set_gc`[5.1].

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

`Cudd_DebugCheck`[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_DebugCheck]. Returns `false` if it is OK, `true` if there is a problem, and throw a `Failure` exception in case of `OUT_OF_MEM`.

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

Cudd_CheckKeys[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CheckKeys].

5.3 Variables, Reordering and Mapping

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

Cudd_ReadPerm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadPerm].
Returns the level of the variable (its order in the BDD)

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

Cudd_ReadInvPerm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadInvPerm].
Returns the variable associated to the given level.

```
val reduce_heap : 'a t -> reorder -> int -> unit
```

Cudd_ReduceHeap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReduceHeap].
Main reordering function, that applies the given heuristic. The provided integer is a bound below which no reordering takes place.

```
val shuffle_heap : 'a t -> int array -> unit
```

Cudd_ShuffleHeap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ShuffleHeap].
Reorder variables according to the given permutation.

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

cuddGarbageCollect[http://vlsi.colorado.edu/~fabio/CUDD/cuddAllDet.html#cuddGarbageCollect].
Force a garbage collection (with cache clearing)

```
val flush : 'a t -> unit
```

cuddCacheFlush[http://vlsi.colorado.edu/~fabio/CUDD/cuddAllDet.html#cuddCacheFlush].
Clear the global cache

```
val enable_autodyn : 'a t -> reorder -> unit
```

Cudd_AutodynEnable[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_AutodynEnable].
Enables dynamic reordering with the given heuristics.

```
val disable_autodyn : 'a t -> unit
```

Cudd_AutodynDisable[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_AutodynDisable].
Disables dynamic reordering.

```
val autodyn_status : 'a t -> reorder option
```

Cudd_ReorderingStatus[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReorderingStatu
Returns `None` if dynamic reordering is disables, `Some(heuristic)` otherwise.

```
val group : 'a t -> int -> int -> mtr -> unit
```

Cudd_MakeTreeNode[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_MakeTreeNode]. `group man low size typ` creates a new variable group, ranging from index `low` to index `low+size-1`, in which `typ` specifies if reordering is allowed inside the group.

```
val ungroupall : 'a t -> unit
```

Cudd_FreeTree[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_FreeTree]. Removes all the groups in the manager.

```
val set_varmap : 'a t -> int array -> unit
```

Cuddaux_SetVarMap/Cudd_SetVarMap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetV
Initializes the global mapping table, used by functions `Cudd.Bdd.varmap`[6.8], `Cudd.Vdd.varmap`[7.6], `Cudd.Mtbdd.varmap`[11.6], `Cudd.Mtbddc.varmap`[12.6],... Convenient when the same mapping is applied several times, because the the different calls reuse the same cache.

5.4 Parameters

5.4.1 RDDs

```
val get_background : dt -> float
```

Cudd_ReadBackground[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadBackground].

```
val set_background : dt -> float -> unit
```

Variation of Cudd_SetBackground[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetBackground].

```
val get_epsilon : dt -> float
```

Cudd_ReadEpsilon[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadEpsilon].

```
val set_epsilon : dt -> float -> unit
```

Cudd_SetEpsilon[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetEpsilon].

5.4.2 Cache related

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

Cudd_ReadMinHit[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMinHit].

```
val set_min_hit : 'a t -> int -> unit
```

Cudd_SetMinHit[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMinHit].

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

Cudd_ReadMaxCacheHard[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxCacheHar

```
val set_max_cache_hard : 'a t -> int -> unit
```

Cudd_SetMaxCacheHard[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMaxCacheHard].

5.4.3 Manager

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

Cudd_ReadLooseUpTo[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadLooseUpTo].

```
val set_looseupto : 'a t -> int -> unit
```

Cudd_SetLooseUpTo[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetLooseUpTo].

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

Cudd_ReadMaxLive[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxLive].

```
val set_max_live : 'a t -> int -> unit
```

Cudd_SetMaxLive[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMaxLive].

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

Cudd_ReadMaxMemory[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxMemory].

```
val set_max_mem : 'a t -> int -> unit
```

Cudd_SetMaxMemory[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMaxMemory].

5.4.4 Reordering methods

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

Cudd_ReadSiftMaxSwap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSiftMaxSwap].

```
val set_sift_max_swap : 'a t -> int -> unit
```

Cudd_SetSiftMaxSwap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetSiftMaxSwap].

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

Cudd_ReadSiftMaxVar[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSiftMaxVar].

```
val set_sift_max_var : 'a t -> int -> unit
```

Cudd_SetSiftMaxVar[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetSiftMaxVar].

```
val get_groupcheck : 'a t -> aggregation
```

Cudd_ReadGroupcheck[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadGroupcheck].

```
val set_groupcheck : 'a t -> aggregation -> unit
```

Cudd_SetGroupcheck[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetGroupcheck].

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

Cudd_ReadArcviolation[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadArcviolatio

```
val set_arcviolation : 'a t -> int -> unit
```

Cudd_SetArcviolation[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetArcviolation].

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

Cudd_ReadNumberXovers[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadNumberXover

```
val set_crossovers : 'a t -> int -> unit
```

Cudd_SetNumberXovers[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetNumberXovers].

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

Cudd_ReadPopulationSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadPopulatio

```
val set_population : 'a t -> int -> unit
```

Cudd_SetPopulationSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetPopulationS

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

Cudd_ReadRecomb[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadRecomb].

```
val set_recomb : 'a t -> int -> unit
```

(Cudd_SetRecomb.

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

Cudd_ReadSymmviolation[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSymmviolat

```
val set_symmviolation : 'a t -> int -> unit
```

Cudd_SetSymmviolation[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetSymmviolatio

5.4.5 Dynamic reordering

```
val get_max_growth : 'a t -> float
```

Cudd_ReadMaxGrowth[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxGrowth].

```
val set_max_growth : 'a t -> int -> unit
```

Cudd_SetMaxGrowth[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMaxGrowth].

```
val get_max_growth_alt : 'a t -> float
```

Cudd_ReadMaxGrowthAlternate[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxGr

```
val set_max_growth_alt : 'a t -> float -> unit
```

Cudd_SetMaxGrowthAlternate[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetMaxGrow

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

Cudd_ReadReorderingCycle[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadReorderi

```
val set_reordering_cycle : 'a t -> int -> unit
```

Cudd_SetReorderingCycle[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetReordering

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

Cudd_ReadNextReordering[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadNextReord

```
val set_next_autodyn : 'a t -> int -> unit
```

Cudd_SetNextReordering[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SetNextReorder

5.5 Statistics

```
val get_cache_hits : 'a t -> float
```

Cudd_ReadCacheHits[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadCacheHits].

```
val get_cache_lookups : 'a t -> float
```

Cudd_ReadCacheLookUps[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadCacheLookUp

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

Cudd_ReadCacheSlots[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadCacheSlots].

```
val get_cache_used_slots : 'a t -> float
```

Cudd_ReadCacheUsedSlots[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadCacheUsed

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

Cudd_ReadDead[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadDead].

```
val get_error : 'a t -> error
```

Cudd_ReadErrorCode[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadErrorCode].

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

Cudd_ReadGarbageCollectionTime[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadGa

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

Cudd_ReadGarbageCollections[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadGarba

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

Cudd_ReadKeys[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadKeys].

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

Cudd_ReadLinear[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadLinear].

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

Cudd_ReadMaxCache[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMaxCache].

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

Cudd_ReadMinDead[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadMinDead].

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

Cudd_ReadNodeCount[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadNodeCount].

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

Cudd_ReadPeakNodeCount[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadPeakNodeCo

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

Cudd_ReadReorderingTime[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadReorderin

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

Cudd_ReadReorderings[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadReorderings].

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

Cudd_ReadSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSize].

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

Cudd_ReadZddSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadZddSize].

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

Cudd_ReadSlots[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSlots].

```
val get_used_slots : 'a t -> float
```

Cudd_ReadUsedSlots[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadUsedSlots].

```
val get_swap_nb : 'a t -> float
```

Cudd_ReadSwapSteps[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ReadSwapSteps].

```
val print_info : 'a t -> unit
```

Cudd_PrintInfo[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_PrintInfo].

```
end
```

Chapter 6

Module Bdd: Binary Decision Diagrams

```
module Bdd :
  sig
```

```
    type 'a t
```

Abstract type for BDDs.

Objects of type `'a t` contain both the top node of the BDD and the manager to which this node belongs. `'a`, which is either `Cudd.Man.d`[5] or `Cudd.Man.v`[5] indicates the kind of manager to which the node belongs, see module `Cudd.Man`[5]. The manager can be retrieved with `Cudd.Bdd.manager`[6.1]. These objects are automatically garbage collected.

```
    type 'a bdd =
      | Bool of bool
```

Terminal value

```
      | Ite of int * 'a t * 'a t
```

Decision on CUDD variable

Public type for exploring the abstract type `t`

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

Shortcuts

6.1 Extractors

```
    val manager : 'a t -> 'a Cudd.Man.t
```

Returns the manager associated to the BDD

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

`Cudd_IsConstant`[`http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_IsConstant`]. Is the BDD constant ?

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

Cudd_IsComplement[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_IsComplement]. Is the BDD a complemented one ?

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

Cudd_NodeReadIndex[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_NodeReadIndex]. Returns the index of the (top node of the) BDD (65535 for a constant BDD)

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

Cudd_T[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_T]. Returns the positive subnode of the BDD

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

Cudd_E[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_E]. Returns the negative subnode of the BDD

```
val cofactors : int -> 'a t -> 'a t * 'a t
```

Returns the positive and negative cofactor of the BDD wrt the variable

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

Cudd_Cofactor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Cofactor]. `cofactor bdd cube` evaluates `bdd` on the cube `cube`

```
val inspect : 'a t -> 'a bdd
```

Decomposes the top node of the BDD

6.2 Supports

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

Cudd_Support[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Support]. Returns the support of the BDD

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

Cudd_SupportSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SupportSize]. Returns the size of the support of the BDD

```
val is_var_in : int -> 'a t -> bool
```

Cuddaux_IsVarIn. Does the given variable belong the support of the BDD ?

```
val vectorsupport : 'a t array -> 'a t
```

Cudd_Cudd_VectorSupport[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Cudd_VectorSu
Returns the support of the array of BDDs.

Raises a `Failure` exception in case where the array is of size 0 (in such case, the manager is unknown, and we cannot return an empty support). This operation does not use the global cache, unlike `Cudd.Bdd.support`[6.2].

6.3 Manipulation of supports

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

Cudd_bddLiteralSetIntersection[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLit
Intersection of supports

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

Cudd_bddAnd[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddAnd].
Union of supports

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

Cudd_Cofactor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Cofactor].
Difference of supports

```
val list_of_support : 'a t -> int list
```

Converts a support into a list of variables

6.4 Constants and Variables

```
val dtrue : 'a Cudd.Man.t -> 'a t
```

Returns the true BDD

```
val dfalse : 'a Cudd.Man.t -> 'a t
```

Returns the false BDD

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

Cudd_bddIthVar[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddIthVar].
Returns the BDD equivalent to the variable of the given index.

```
val newvar : 'a Cudd.Man.t -> 'a t
```

Cudd_bddNewVar[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddNewVar].
Returns the BDD equivalent to the variable of the next unused index.

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

Cudd_bddNewVarAtLevel[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddNewVarAtLeve
Returns the BDD equivalent to the variable of the next unused index and sets its level.

6.5 Logical tests

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

Is it a true BDD ?

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

Is it a false BDD ?

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

Are the two BDDs equal ?

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

Cudd_bddLeq[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLeq].
Does the first BDD implies the second one ?

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

Variation of
Cudd_bddLeq[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLeq].
Is the intersection (conjunction) of the two BDDs non empty (false) ?

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

Variation of
Cudd_EquivDC[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_EquivDC].
Are the two first BDDs equal when the third one (careset) is true ?

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

Variation of
Cudd_bddLeqUnless[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLeqUnless].
Does the first BDD implies the second one when the third one (careset) is true ?

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

Cudd_bddLeq[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLeq].
Same as Cudd.Bdd.is_leq[6.5]
Is the result of ite constant, and if it is the case, what is the constant ?

```
val is_ite_cst : 'a t -> 'a t -> 'a t -> bool option
val is_var_dependent : int -> 'a t -> bool
```

Cudd_bddVarIsDependent[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddVarIsDepend
Is the given variable dependent on others in the BDD ?

```
val is_var_essential : int -> bool -> 'a t -> bool
```

Cudd_bddIsVarEssential[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddIsVarEssent
Is the given variable with the specified phase implied by the BDD ?

6.6 Structural information

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

Cudd_DagSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_DagSize].
Size if the BDD as a graph (the number of nodes).

```
val nbpaths : 'a t -> float
```

Cudd_CountPath[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountPath].
Number of paths in the BDD from the root to the leaves.

```
val nbtruepaths : 'a t -> float
```

Cudd_CountPathsToNonZero[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountPathsTo
Number of paths in the BDD from the root to the true leaf.

```
val nbminterms : int -> 'a t -> float
```

Cudd_CountMinterm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountMinterm].
Number of minterms of the BDD assuming that it depends on the given number of variables.

```
val density : int -> 'a t -> float
```

Cudd_Density[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Density].
Density of the BDD, which is the ratio of the number of minterms to the number of nodes.
The BDD is assumed to depend on `nvars` variables.

6.7 Logical operations

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

Cudd_Not[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Not].
Negation

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

Cudd_bddAnd[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddAnd].
Conjunction/Intersection

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

Cudd_bddOr[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddOr].
Disjunction/Union

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

Cudd_bddXor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddXor].
Exclusive union

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

Cudd_bddNand[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddNand].

```
val nor : 'a t -> 'a t -> 'a t
```

Cudd_bddNor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddNor].

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

Cudd_bddXnor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddXnor].
Equality

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

Same as `Cudd.Bdd.nxor`[6.7]

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

Cudd_bddIte[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddIte].
If-then-else operation.

```
val ite_cst : 'a t -> 'a t -> 'a t -> 'a t option
```

Cudd_bddIteConstant[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddIteConstant].
If-then-else operation that succeeds when the result is a node of the arguments.

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

Cudd_bddCompose[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddCompose].
`compose var f bdd` substitutes the variable `var` with the function `f` in `bdd`.

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

Cudd_bddVectorCompose[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddVectorCompos
`vectorcompose table bdd` performs a parallel substitution of every variable `var` present in the manager by `table.(var)` in `bdd`. The size of `table` should be at least `Cudd.Man.get_bddvar_nb`[5.5]. You can optionnally control the memoization policy, see `Cudd.Memo`[4].

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

Cudd_bddIntersect[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddIntersect].
Returns a BDD included in the intersection of the arguments.

6.8 Variable mapping

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

Cudd_bddVarMap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddVarMap].
Permutes the variables as it has been specified with `Cudd.Man.set_varmap`[5.3].

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

Cudd_bddPermute[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddPermute].
Permutes the variables as it is specified by `permut` (same format as in `Cudd.Man.set_varmap`[5.3]). You can optionnally control the memoization policy, see `Cudd.Memo`[4].

6.9 Iterators

```
val iter_node : ('a t -> unit) -> 'a t -> unit
```

Cudd_ForeachNode[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ForeachNode].
Apply the function `f` to each (regularized) node of the BDD.

```
val iter_cube : (Cudd.Man.tbool array -> unit) -> 'a t -> unit
```

Cudd_ForeachCube[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ForeachCube]. Apply the function f to each cube of the BDD. The cubes are specified as arrays of elements of type Cudd.Man.tbool[5]. The size of the arrays is equal to Cudd.Man.get_bddvar_nb[5.5], the number of variables present in the manager.

```
val iter_prime : (Cudd.Man.tbool array -> unit) -> 'a t -> 'a t -> unit
```

Cudd_ForeachPrime[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ForeachPrime]. Apply the function f to each prime covering the BDD interval. The first BDD argument is the lower bound, the second the upper bound (which may be equal to the lower bound). The primes are specified as arrays of elements of type Cudd.Man.tbool[5]. The size of the arrays is equal to Cudd.Man.get_bddvar_nb[5.5], the number of variables present in the manager.

6.10 Quantifications

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

Cudd_bddExistAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddExistAbstrac
exist supp bdd quantifies existentially the set of variables defined by supp in the BDD.

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

Cudd_bddUnivAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddUnivAbstract].
forall supp bdd quantifies universally the set of variables defined by supp in the BDD.

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

Cudd_bddAndAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddAndAbstract]. Simultaneous existential quantification and intersection of BDDs. Logically, existand supp x y = exist supp (dand x y).

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

Cudd_bddXorExistAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddXorExistA
Simultaneous existential quantification and exclusive or of BDDs. Logically, existxor supp x y = exist supp (xor x y).

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

Cudd_bddBooleanDiff[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddBooleanDiff]. Boolean difference of the BDD with respect to the variable.

6.11 Cubes

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

Cudd_FindEssential[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_FindEssential]. Returns the smallest cube (in the sens of inclusion) included in the BDD.

```
val cube_of_minterm : 'a Cudd.Man.t -> Cudd.Man.tbool array -> 'a t
```

Cudd_CubeArrayToBdd[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CubeArrayToBdd]. Converts a minterm to a BDD (which is a cube).

```
val list_of_cube : 'a t -> (int * bool) list
```

Converts a cube into a list of pairs of a variable and a phase.

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

Cuddaux_bddCubeUnion. Computes the union of cubes, which is the smallest cube containing both the argument cubes.

```
val pick_minterm : 'a t -> Cudd.Man.tbool array
```

Cudd_bddPickOneCube[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddPickOneCube]. Picks randomly a minterm in the BDD.

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

Cudd_bddPickOneMinterm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddPickOneMint
`pick_cube_on_support bdd supp` picks randomly a minterm/cube in the BDD, in which all the variables in the support `supp` have a definite value.
The support argument should contain the support of the BDD (otherwise the result may be incorrect).

```
val pick_cubes_on_support : 'a t -> 'a t -> int -> 'a t array
```

Cudd_bddPickArbitraryMinterms[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddPick
`pick_cubes_on_support bdd supp nb` picks randomly `nb` minterms/cubes in the BDD, in which all the variables in the support have a definite value. The support argument should contain the support of the BDD (otherwise the result may be incorrect).
Fails if the effective number of such minterms in the BDD is less than `nb`.

6.12 Minimizations

The 6 following functions are generalized cofactor operations. `gencof f c` returns a BDD that coincides with `f` whenever `c` is true (and which is hopefully smaller). `constrain` enjoys in addition strong properties (see papers from Madre and Coudert)

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

Cudd_bddConstrain[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddConstrain].

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

Cuddaux_bddTDConstrain.

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

Cuddaux_bddRestrict.

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

Cuddaux_bddTDRestrict.

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

Cudd_bddMinimize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddMinimize].

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

Cudd_bddLICompaction[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddLICompaction].

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

Cudd_bddSqueeze[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddSqueeze].
`sqeeze lower upper` returns a (smaller) BDD which is in the functional interval
`[lower,upper]`.

6.13 Approximations

```
val clippingand : 'a t -> 'a t -> int -> bool -> 'a t
```

Cudd_bddClippingAnd[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddClippingAnd].
`clippingand f g maxdepth direction`

```
val clippingexistand : 'a t ->
  'a t -> 'a t -> int -> bool -> 'a t
```

Cudd_bddClippingAndAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddClippi
`clippingexistand supp f g maxdepth direction` (order of argulents changed).

```
val underapprox : int -> int -> bool -> float -> 'a t -> 'a t
```

Cudd_UnderApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_UnderApprox].
`underapprox nvars threshold safe quality f`

```
val overapprox : int -> int -> bool -> float -> 'a t -> 'a t
```

Cudd_OverApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_OverApprox].
`overapprox nvars threshold safe quality f`

```
val remapunderapprox : int -> int -> float -> 'a t -> 'a t
```

Cudd_RemapUnderApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_RemapUnderAppro
`remapunderapprox nvars threshold quality f`

```
val remapoverapprox : int -> int -> float -> 'a t -> 'a t
```

Cudd_RemapOverApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_RemapOverApprox].
`remapovererapprox nvars threshold quality f`

```
val biasedunderapprox :
  int ->
  int -> float -> float -> 'a t -> 'a t -> 'a t
```

Cudd_BiasedUnderApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_BiasedUnderApp
biasedunderapprox nvars threshold quality1 quality0 f g

```
val biasedoverapprox : int ->
  int -> float -> float -> 'a t -> 'a t -> 'a t
```

Cudd_BiasedOverApprox[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_BiasedOverAppro
biasedovererapprox nvars threshold quality1 quality0 f g

For the 4 next functions, the profile is `XXcompress nvars threshold f`.

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

Cudd_SubsetCompress[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SubsetCompress].

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

Cudd_SupersetCompress[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SupersetCompres

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

Cudd_SubsetHeavyBranch[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SubsetHeavyBra

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

Cudd_SupersetHeavyBranch[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SupersetHeav

For the 2 next functions, the profile is `XXXsetSP nvars threshold hardlimit f`.

```
val subsetSP : int -> int -> bool -> 'a t -> 'a t
```

Cudd_SubsetShortPaths[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SubsetShortPath

```
val supersetSP : int -> int -> bool -> 'a t -> 'a t
```

Cudd_SupersetShortPaths[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SupersetShort

The following functions perform two-way conjunctive (disjunctive) decomposition of a BDD. Returns a pair if successful, `None` if no decomposition has been found.

```
val approxconjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddApproxConjDecomp.
Cudd_bddApproxDisjDecomp.

```
val approxdisjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddIterConjDecomp.

```
val iterconjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddIterDisjDecomp.

```
val iterdisjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddGenConjDecomp.

```
val genconjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddGenDisjDecomp.

```
val gendisjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddVarConjDecomp.

```
val varconjdecomp : 'a t -> ('a t * 'a t) option
```

Cudd_bddVarDisjDecomp.

```
val vardisjdecomp : 'a t -> ('a t * 'a t) option
```

6.14 Miscellaneous

```
val transfer : 'a t -> 'b Cudd.Man.t -> 'b t
```

Cudd_bddTransfer[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddTransfer]. Transfers a BDD to a different manager.

```
val correlation : 'a t -> 'a t -> float
```

Cudd_bddCorrelation[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddCorrelation]. Computes the correlation of f and g (if `f=g`, their correlation is 1, if `f=not g`, it is 0)

```
val correlationweights : 'a t -> 'a t -> float array -> float
```

Cudd_bddCorrelationWeights[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddCorrela

6.15 Printing

```
val _print : 'a t -> unit
```

Raw (C) printing function. The output may mix badly with the OCAML output.

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

Prints the minterms of the BDD in the same way as Cudd_Printminterm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Printminterm].

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

`print_minterm bassoc fmt bdd` prints the minterms of the BDD using `bassoc` to convert indices of variables to names.

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

Prints a BDD by recursively decomposing it as monomial followed by a tree.

```
val print_list :
  (Format.formatter -> int -> unit) ->
  Format.formatter -> (int * bool) list -> unit
end
```

Chapter 7

Module Vdd: MTBDDs with OCaml values (INTERNAL)

```
module Vdd :
  sig
```

```
    type +'a t
```

Type of VDDs (that are necessarily attached to a manager of type `Man.v Man.t`).

Objects of this type contains both the top node of the ADD and the manager to which the node belongs. The manager can be retrieved with `Cudd.Vdd.manager`[7.1]. Objects of this type are automatically garbage collected.

```
    type 'a vdd =
      | Leaf of 'a
```

Terminal value

```
      | Ite of int * 'a t * 'a t
```

Decision on CUDD variable

Public type for exploring the abstract type `t`

We refer to the module `Cudd.Add`[15] for the description of the interface, as it is nearly identical to `Cudd.Add`[15], except that real leaves are replaced by OCaml leaves.

IMPORTANT NOTE: this is an internal module, which assumes that leaves are either immediate values (booleans, integers, constant sums), or values allocated with caml_alloc_shr (that can be moved only during a memory compaction).

The only case where you may use directly `Cudd.Vdd`[7] without worrying is when the leaves are represented as immediate values (booleans, integers, constant sums) in the heap.

Otherwise, use module `Cudd.Mtbdd`[11] or `Cudd.Mtbddc`[12] to be safe, and also to ensure that you do not have two constant MTBDDs pointing to different but semantically equivalent values.

7.1 Extractors

```
    val manager : 'a t -> Cudd.Man.v Cudd.Man.t
    val is_cst : 'a t -> bool
    val topvar : 'a t -> int
    val dthen : 'a t -> 'a t
    val delse : 'a t -> 'a t
```

```
val cofactors : int -> 'a t -> 'a t * 'a t
val cofactor : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val dval : 'a t -> 'a
val inspect : 'a t -> 'a vdd
```

7.2 Supports

```
val support : 'a t -> Cudd.Man.v Cudd.Bdd.t
val supportsize : 'a t -> int
val is_var_in : int -> 'a t -> bool
val vectorsupport : 'a t array -> Cudd.Man.v Cudd.Bdd.t
val vectorsupport2 :
  Cudd.Man.v Cudd.Bdd.t array -> 'a t array -> Cudd.Man.v Cudd.Bdd.t
```

7.3 Classical operations

```
val cst : Cudd.Man.v Cudd.Man.t -> 'a -> 'a t
val _background : Cudd.Man.v Cudd.Man.t -> 'a t
```

Be cautious, it is not type safe (if you use `Cudd.Vdd.nodes_below_level`[7.8], etc...: you can try to retrieve a constant value of some type and `()` value of the background value will be treated as another type.

```
val ite : Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t -> 'a t
val ite_cst : Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a t option
val eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t option
val compose : int -> Cudd.Bdd.vt -> 'a t -> 'a t
val vectorcompose : ?memo:Cudd.Memo.t -> Cudd.Bdd.vt array -> 'a t -> 'a t
```

7.4 Logical tests

```
val is_equal : 'a t -> 'a t -> bool
val is_equal_when : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t -> bool
val is_eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a option
val is_ite_cst : Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t -> 'a option
```

7.5 Structural information

```
val size : 'a t -> int
val nbpaths : 'a t -> float
val nbnonzeropaths : 'a t -> float
val nbminterms : int -> 'a t -> float
val density : int -> 'a t -> float
val nbleaves : 'a t -> int
```

7.6 Variable mapping

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

Variant with controllable memoization policy.

7.7 Iterators

```
val iter_cube : (Cudd.Man.tbool array -> 'a -> unit) -> 'a t -> unit
val iter_node : ('a t -> unit) -> 'a t -> unit
```

7.8 Leaves and guards

```
val guard_of_node : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t
val guard_of_nonbackground : 'a t -> Cudd.Man.v Cudd.Bdd.t
val nodes_below_level : ?max:int -> 'a t -> int option -> 'a t array
```

`Cuddaux_NodesBelowLevel`. `nodes_below_level ?max f olevel` returns all (if `max=None`), otherwise at most `Some max` nodes pointed by the ADD, indexed by a variable of level greater or equal than `level`, and encountered first in the top-down exploration (i.e., whenever a node is collected, its sons are not collected). If `olevel=None`, then only constant nodes are collected.

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

Guard of the given leaf

```
val leaves : 'a t -> 'a array
```

Returns the set of leaf values (excluding the background value)

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

Picks (but not randomly) a non background leaf. Return `None` if the only leaf is the background leaf.

```
val guardleafs : 'a t -> (Cudd.Man.v Cudd.Bdd.t * 'a) array
```

Returns the set of leaf values together with their guard in the ADD

7.9 Minimizations

```
val constrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdconstrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val restrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdrestrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
```

7.10 Conversions

7.11 User operations

Two options:

- By decomposition into guards and leafs: see module `Cudd.Mapleaf`[14]
- By using CUDD cache: see module `Cudd.User`[13]

7.12 Miscellaneous

```
val transfer : 'a t -> Cudd.Man.v Cudd.Man.t -> 'a t
```

7.13 Printing

```
val print__minterm :
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
val print_minterm :
  (Format.formatter -> int -> unit) ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
val print :
  (Format.formatter -> Cudd.Man.v Cudd.Bdd.t -> unit) ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
end
```

Chapter 8

Module Custom: Type of identifiers

```
module Custom :
  sig
```

Custom Operations on VDDs

```
    type pid
    type mlvalue
    type common = {
      pid : pid ;
      arity : int ;
      memo : Cudd.Memo.t ;
    }
```

Common information

```
    type ('a, 'b) op1 = {
      common1 : common ;
      closure1 : 'a -> 'b ;
    }
```

Unary operation

```
    type ('a, 'b, 'c) op2 = {
      common2 : common ;
      closure2 : 'a -> 'b -> 'c ;
      ospecial2 : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t option) option ;
      commutative : bool ;
      idempotent : bool ;
    }
```

Binary operation

```
    type ('a, 'b) test2 = {
      common2t : common ;
      closure2t : 'a -> 'b -> bool ;
      ospecial2t : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool option) option ;
      symetric : bool ;
      reflexive : bool ;
    }
```

Binary test

```
    type ('a, 'b, 'c, 'd) op3 = {
      common3 : common ;
```

```
  closure3 : 'a -> 'b -> 'c -> 'd ;
  ospecial3 : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -
> 'd Cudd.Vdd.t option)
  option ;
}
```

Ternary operation

```
type ('a, 'b) opN = {
  commonN : common ;
  arityNbdd : int ;
  closureN : Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option ;
}
```

N-ary operation

```
type ('a, 'b) opG = {
  commonG : common ;
  arityGbdd : int ;
  closureG : Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option ;
  oclosureBeforeRec : (int * bool ->
   Cudd.Bdd.vt array ->
   'a Cudd.Vdd.t array -> Cudd.Bdd.vt array * 'a Cudd.Vdd.t array)
  option ;
  oclosureIte : (int -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t) option ;
}
```

N-ary general operation

```
type 'a exist = {
  commonexist : common ;
  combineexist : ('a, 'a, 'a) op2 ;
}
```

Existential quantification

```
type 'a existand = {
  commonexistand : common ;
  combineexistand : ('a, 'a, 'a) op2 ;
  bottomexistand : 'a ;
}
```

Existential quantification combined with intersection

```
type ('a, 'b) existop1 = {
  commonexistop1 : common ;
  combineexistop1 : ('b, 'b, 'b) op2 ;
  existop1 : ('a, 'b) op1 ;
}
```

Existop1ential quantification

```
type ('a, 'b) existandop1 = {
  commonexistandop1 : common ;
  combineexistandop1 : ('b, 'b, 'b) op2 ;
  existandop1 : ('a, 'b) op1 ;
  bottomexistandop1 : 'b ;
}
```

Existential quantification combined with intersection

```
    val newpid : unit -> pid
    val apply_op1 : ('a, 'b) op1 -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
    val apply_op2 :
      ('a, 'b, 'c) op2 ->
      'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t
    val apply_test2 : ('a, 'b) test2 -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool
    val apply_op3 :
      ('a, 'b, 'c, 'd) op3 ->
      'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t
    val apply_opN :
      ('a, 'b) opN ->
      Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t
    val apply_opG :
      ('a, 'b) opG ->
      Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t
    val _apply_exist : 'a exist -> Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t
    val _apply_existand :
      'a existand ->
      Cudd.Bdd.vt -> Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t
    val _apply_existop1 :
      ('a, 'b) existop1 ->
      Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
    val _apply_existandop1 :
      ('a, 'b) existandop1 ->
      Cudd.Bdd.vt -> Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
  end
```

Chapter 9

Module Weakke: Hash tables of weak pointers.

```
module Weakke :
  sig
```

Original `Weak` module of OCaml distribution modified by Bertrand Jeannet with a `Custom` (polymorphic) module.

A weak hash table is a hashed set of values. Each value may magically disappear from the set when it is not used by the rest of the program any more. This is normally used to share data structures without inducing memory leaks. Weak hash tables are defined on values from a `Hashtbl.HashedType` module; the `equal` relation and `hash` function are taken from that module. We will say that `v` is an instance of `x` if `equal x v` is `true`.

The `equal` relation must be able to work on a shallow copy of the values and give the same result as with the values themselves.

```
type 'a t
type 'a hashtbl = 'a t
type 'a compare = {
  hash : 'a -> int ;
  equal : 'a -> 'a -> bool ;
}
val create : int -> 'a t
val clear : 'a t -> unit
val merge : 'a t -> 'a -> 'a
val merge_map : 'a t -> 'a -> ('a -> 'a) -> 'a
val add : 'a t -> 'a -> unit
val remove : 'a t -> 'a -> unit
val find : 'a t -> 'a -> 'a
val find_all : 'a t -> 'a -> 'a list
val mem : 'a t -> 'a -> bool
val iter : ('a -> 'b) -> 'a t -> unit
val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
val count : 'a t -> int
val stats : 'a t -> int * int * int * int * int * int
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 -> unit) ->
  Format.formatter -> 'a t -> unit
module type S =
  sig
```

`type data`

The type of the elements stored in the table.

`type t`

The type of tables that contain elements of type `data`. Note that weak hash tables cannot be marshaled using `Pervasives.output_value` or the functions of the `Marshal` module.

`val create : int -> t`

`create n` creates a new empty weak hash table, of initial size `n`. The table will grow as needed.

`val clear : t -> unit`

Remove all elements from the table.

`val merge : t -> data -> data`

`merge t x` returns an instance of `x` found in `t` if any, or else adds `x` to `t` and return `x`.

```
val merge_map : t ->
  data ->
  (data -> data) -> data
```

Variant of `merge`: `merge_map t x f` is equivalent to `try find t x with Not_found -> let y = f x in add t y; Some y`. bE CAUTIOUS: `f x` is assumed to be equal to `x`.

`val add : t -> data -> unit`

`add t x` adds `x` to `t`. If there is already an instance of `x` in `t`, it is unspecified which one will be returned by subsequent calls to `find` and `merge`.

`val remove : t -> data -> unit`

`remove t x` removes from `t` one instance of `x`. Does nothing if there is no instance of `x` in `t`.

`val find : t -> data -> data`

`find t x` returns an instance of `x` found in `t`. Raise `Not_found` if there is no such element.

`val find_all : t -> data -> data list`

`find_all t x` returns a list of all the instances of `x` found in `t`.

`val mem : t -> data -> bool`

`mem t x` returns `true` if there is at least one instance of `x` in `t`, false otherwise.

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

`iter f t` calls `f` on each element of `t`, in some unspecified order. It is not specified what happens if `f` tries to change `t` itself.

`val fold : (data -> 'a -> 'a) -> t -> 'a -> 'a`

`fold f t init` computes `(f d1 (... (f dN init)))` where `d1 ... dN` are the elements of `t` in some unspecified order. It is not specified what happens if `f` tries to change `t` itself.

```
val count : t -> int
```

Count the number of elements in the table. `count t` gives the same result as `fold (fun _ n -> n+1) t 0` but does not delay the deallocation of the dead elements.

```
val stats : t -> int * int * int * int * int * int
```

Return statistics on the table. The numbers are, in order: table length, number of entries, sum of bucket lengths, smallest bucket length, median bucket length, biggest bucket length.

```
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 -> data -> unit) ->
  Format.formatter -> t -> unit
```

Printing function

```
end
```

The output signature of the functor `Weak.Make`.

```
module Make :
functor (H : Hashtbl.HashedType) -> S  with type data = H.t
```

Functor building an implementation of the weak hash table structure.

```
module Compare :
  sig
    val add : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> unit
    val find_or :
      'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> (int -> int -> 'a) -
    > 'a
    val merge : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> 'a
    val merge_map :
      'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> ('a -> 'a) -> 'a
    val find : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> 'a
    val find_shadow :
      'a Cudd.Weakke.compare ->
      'a Cudd.Weakke.t -> 'a -> ('a Weak.t -> int -> 'b) -> 'b -> 'b
    val remove : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> unit
    val mem : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> bool
    val find_all : 'a Cudd.Weakke.compare -> 'a Cudd.Weakke.t -> 'a -> 'a list
  end

end
```

Chapter 10

Module PWeakke: Hash tables of weak pointers, parametrized polymorphic version.

```
module PWeakke :
  sig
```

Same interface as Cudd.Weakke[9].

```
    type 'a compare = 'a Cudd.Weakke.compare = {
      hash : 'a -> int ;
      equal : 'a -> 'a -> bool ;
    }
    type 'a t = {
      compare : 'a compare ;
      hashtbl : 'a Cudd.Weakke.t ;
    }
    val create : ('a -> int) -> ('a -> 'a -> bool) -> int -> 'a t
    val clear : 'a t -> unit
    val merge : 'a t -> 'a -> 'a
    val merge_map : 'a t -> 'a -> ('a -> 'a) -> 'a
    val add : 'a t -> 'a -> unit
    val remove : 'a t -> 'a -> unit
    val find : 'a t -> 'a -> 'a
    val find_all : 'a t -> 'a -> 'a list
    val mem : 'a t -> 'a -> bool
    val iter : ('a -> 'b) -> 'a t -> unit
    val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val count : 'a t -> int
    val stats : 'a t -> int * int * int * int * int * int
    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 -> unit) ->
      Format.formatter -> 'a t -> unit
  end
```

Chapter 11

Module Mtbdd: MTBDDs with OCaml values

```
module Mtbdd :
  sig
```

```
    type 'a unique
```

Type of unique representants of MTBDD leaves of type 'a.

For technical reason, type 'a should not be implemented as a custom block with finalization function. (This is checked and the program aborts with an error message).

Use Cudd.Mtbddc[12] module if your type does not fulfill this requirement. Mtbddc modules automatically encapsulate the value into a ML type.

```
    type 'a t = 'a unique Cudd.Vdd.t
```

Type of MTBDDs.

Objects of this type contains both the top node of the MTBDD and the manager to which the node belongs. The manager can be retrieved with Cudd.Mtbdd.manager[11.1]. Objects of this type are automatically garbage collected.

```
    type 'a table = 'a unique Cudd.PWeakke.t
```

Hashtable to manage unique constants

```
    val print_table :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> 'a -> unit) ->
      Format.formatter -> 'a table -> unit
    val make_table : hash:('a -> int) -> equal:('a -> 'a -> bool) -> 'a table
```

Building a table

```
    val unique : 'a table -> 'a -> 'a unique
```

Building a unique constant

```
    val get : 'a unique -> 'a
```

Type conversion (no computation)

```
    type 'a mtbdd =
      | Leaf of 'a unique
```

Terminal value

| Ite of int * 'a t * 'a t

Decision on CUDD variable

Public type for exploring the abstract type t

We refer to the modules Cudd.Add[15] and Cudd.Vdd[7] for the description of the interface.

11.1 Extractors

```
val manager : 'a t -> Cudd.Man.v Cudd.Man.t
```

Returns the manager associated to the MTBDD

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

Is the MTBDD constant ?

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

Returns the index of the top node of the MTBDD (65535 for a constant MTBDD)

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

Returns the positive subnode of the MTBDD

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

Returns the negative subnode of the MTBDD

```
val cofactors : int -> 'a t -> 'a t * 'a t
```

Returns the positive and negative cofactor of the MTBDD wrt the variable

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

cofactor mtbdd cube evaluates mtbbdd on the cube cube

```
val dval_u : 'a t -> 'a unique
val dval : 'a t -> 'a
```

Returns the value of the assumed constant MTBDD

```
val inspect : 'a t -> 'a mtbdd
```

Decompose the MTBDD

11.2 Supports

```
val support : 'a t -> Cudd.Man.v Cudd.Bdd.t
val supportsize : 'a t -> int
val is_var_in : int -> 'a t -> bool
val vectorsupport : 'a t array -> Cudd.Man.v Cudd.Bdd.t
val vectorsupport2 :
  Cudd.Man.v Cudd.Bdd.t array -> 'a t array -> Cudd.Man.v Cudd.Bdd.t
```

11.3 Classical operations

```
val cst_u : Cudd.Man.v Cudd.Man.t -> 'a unique -> 'a t
val cst : Cudd.Man.v Cudd.Man.t -> 'a table -> 'a -> 'a t
val ite : Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a t
val ite_cst : Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a t option
val eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t option
val compose : int -> Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t
val vectorcompose :
  ?memo:Cudd.Memo.t ->
  Cudd.Man.v Cudd.Bdd.t array -> 'a t -> 'a t
```

11.4 Logical tests

```
val is_equal : 'a t -> 'a t -> bool
val is_equal_when : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t -> bool
val is_eval_cst_u : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a unique option
val is_ite_cst_u :
  Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a unique option
val is_eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a option
val is_ite_cst : Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t -> 'a option
```

11.5 Structural information

```
val size : 'a t -> int
val nbpaths : 'a t -> float
val nbnonzeropaths : 'a t -> float
val nbminterms : int -> 'a t -> float
val density : int -> 'a t -> float
val nbleaves : 'a t -> int
```

11.6 Variable mapping

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

11.7 Iterators

```
val iter_cube_u :
  (Cudd.Man.tbool array -> 'a unique -> unit) ->
  'a t -> unit
val iter_cube : (Cudd.Man.tbool array -> 'a -> unit) -> 'a t -> unit
val iter_node : ('a t -> unit) -> 'a t -> unit
```

11.8 Leaves and guards

```
val guard_of_node : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t
val guard_of_nonbackground : 'a t -> Cudd.Man.v Cudd.Bdd.t
val nodes_below_level : ?max:int -> 'a t -> int option -> 'a t array
val guard_of_leaf_u : 'a t -> 'a unique -> Cudd.Man.v Cudd.Bdd.t
```

Guard of the given leaf

```
val guard_of_leaf : 'a table -> 'a t -> 'a -> Cudd.Man.v Cudd.Bdd.t
val leaves_u : 'a t -> 'a unique array
```

Returns the set of leaf values (excluding the background value)

```
val leaves : 'a t -> 'a array
val pick_leaf_u : 'a t -> 'a unique
```

Picks (but not randomly) a non background leaf. Return `None` if the only leaf is the background leaf.

```
val pick_leaf : 'a t -> 'a
val guardleafs_u : 'a t -> (Cudd.Man.v Cudd.Bdd.t * 'a unique) array
```

Returns the set of leaf values together with their guard in the ADD

```
val guardleafs : 'a t -> (Cudd.Man.v Cudd.Bdd.t * 'a) array
```

11.9 Minimizations

```
val constrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdconstrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val restrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdrestrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
```

11.10 Conversions

11.11 User operations

Two options:

- By decomposition into guards and leafs: see module `Cudd.Mapleaf`[14]
- By using CUDD cache: see module `Cudd.User`[13]

11.12 Miscellaneous

```
val transfer : 'a t -> Cudd.Man.v Cudd.Man.t -> 'a t
```

11.13 Printing

```
  val print__minterm :
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
  val print_minterm :
    (Format.formatter -> int -> unit) ->
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
  val print :
    (Format.formatter -> Cudd.Man.v Cudd.Bdd.t -> unit) ->
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
end
```

Chapter 12

Module Mtbddc: MTBDDs with finalized OCaml values.

```
module Mtbddc :
  sig
```

```
    type 'a capsule = private {
      content : 'a ;
    }
```

```
    type 'a unique
```

Type of unique representants of MTBDD leaves of type `'a`.

Use this module rather than `Cudd.Mtbdd`[11] when `'a` is implemented as a a custom block with finalization function.

```
    type 'a t = 'a unique Cudd.Vdd.t
```

Type of MTBDDs.

Objects of this type contains both the top node of the MTBDD and the manager to which the node belongs. The manager can be retrieved with `Cudd.Mtbddc.manager`[12.1]. Objects of this type are automatically garbage collected.

```
    type 'a table = 'a unique Cudd.PWeakke.t
```

Hashtable to manage unique constants

```
    val print_table :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> 'a -> unit) ->
      Format.formatter -> 'a table -> unit
    val make_table : hash:('a -> int) -> equal:('a -> 'a -> bool) -> 'a table
```

Building a table

```
    val unique : 'a table -> 'a -> 'a unique
```

Building a unique constant

```
    val get : 'a unique -> 'a
```

Type conversion (no computation)

```
type 'a mtbdd =
  | Leaf of 'a unique
```

Terminal value

```
  | Ite of int * 'a t * 'a t
```

Decision on CUDD variable

Public type for exploring the abstract type `t`

We refer to the modules `Cudd.Add`[15] and `Cudd.Vdd`[7] for the description of the interface.

12.1 Extractors

```
val manager : 'a t -> Cudd.Man.v Cudd.Man.t
```

Returns the manager associated to the MTBDD

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

Is the MTBDD constant ?

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

Returns the index of the top node of the MTBDD (65535 for a constant MTBDD)

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

Returns the positive subnode of the MTBDD

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

Returns the negative subnode of the MTBDD

```
val cofactors : int -> 'a t -> 'a t * 'a t
```

Returns the positive and negative cofactor of the MTBDD wrt the variable

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

`cofactor mtbdd cube` evaluates `mtbbdd` on the cube `cube`

```
val dval_u : 'a t -> 'a unique
val dval : 'a t -> 'a
```

Returns the value of the assumed constant MTBDD

```
val inspect : 'a t -> 'a mtbdd
```

Decompose the MTBDD

12.2 Supports

```
val support : 'a t -> Cudd.Man.v Cudd.Bdd.t
val supportsize : 'a t -> int
val is_var_in : int -> 'a t -> bool
val vectorsupport : 'a t array -> Cudd.Man.v Cudd.Bdd.t
val vectorsupport2 :
  Cudd.Man.v Cudd.Bdd.t array ->
  'a t array -> Cudd.Man.v Cudd.Bdd.t
```

12.3 Classical operations

```
val cst_u : Cudd.Man.v Cudd.Man.t -> 'a unique -> 'a t
val cst : Cudd.Man.v Cudd.Man.t -> 'a table -> 'a -> 'a t
val ite : Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a t
val ite_cst : Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a t option
val eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t option
val compose : int -> Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t
val vectorcompose : Cudd.Man.v Cudd.Bdd.t array -> 'a t -> 'a t
```

12.4 Logical tests

```
val is_equal : 'a t -> 'a t -> bool
val is_equal_when : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t -> bool
val is_eval_cst_u : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a unique option
val is_ite_cst_u :
  Cudd.Man.v Cudd.Bdd.t ->
  'a t -> 'a t -> 'a unique option
val is_eval_cst : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a option
val is_ite_cst : Cudd.Man.v Cudd.Bdd.t -> 'a t -> 'a t -> 'a option
```

12.5 Structural information

```
val size : 'a t -> int
val nbpaths : 'a t -> float
val nbnonzeropaths : 'a t -> float
val nbminterms : int -> 'a t -> float
val density : int -> 'a t -> float
val nbleaves : 'a t -> int
```

12.6 Variable mapping

```
val varmap : 'a t -> 'a t
val permute : 'a t -> int array -> 'a t
```

12.7 Iterators

```
val iter_cube_u :
  (Cudd.Man.tbool array -> 'a unique -> unit) ->
  'a t -> unit
val iter_cube : (Cudd.Man.tbool array -> 'a -> unit) -> 'a t -> unit
val iter_node : ('a t -> unit) -> 'a t -> unit
```

12.8 Leaves and guards

```
val guard_of_node : 'a t -> 'a t -> Cudd.Man.v Cudd.Bdd.t
val guard_of_nonbackground : 'a t -> Cudd.Man.v Cudd.Bdd.t
val nodes_below_level : ?max:int -> 'a t -> int option -> 'a t array
val guard_of_leaf_u : 'a t -> 'a unique -> Cudd.Man.v Cudd.Bdd.t
```

Guard of the given leaf

```
val guard_of_leaf : 'a table -> 'a t -> 'a -> Cudd.Man.v Cudd.Bdd.t
val leaves_u : 'a t -> 'a unique array
```

Returns the set of leaf values (excluding the background value)

```
val leaves : 'a t -> 'a array
val pick_leaf_u : 'a t -> 'a unique
```

Picks (but not randomly) a non background leaf. Return `None` if the only leaf is the background leaf.

```
val pick_leaf : 'a t -> 'a
val guardleafs_u : 'a t -> (Cudd.Man.v Cudd.Bdd.t * 'a unique) array
```

Returns the set of leaf values together with their guard in the ADD

```
val guardleafs : 'a t -> (Cudd.Man.v Cudd.Bdd.t * 'a) array
```

12.9 Minimizations

```
val constrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdconstrain : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val restrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
val tdrestrict : 'a t -> Cudd.Man.v Cudd.Bdd.t -> 'a t
```

12.10 Conversions

12.11 User operations

Two options:

- By decomposition into guards and leafs: see module `Cudd.Mapleaf`[14];
- By using CUDD cache: see module `Cudd.User`[13].

12.12 Miscellaneous

```
val transfer : 'a t -> Cudd.Man.v Cudd.Man.t -> 'a t
```

12.13 Printing

```
  val print__minterm :
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
  val print_minterm :
    (Format.formatter -> int -> unit) ->
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
  val print :
    (Format.formatter -> Cudd.Man.v Cudd.Bdd.t -> unit) ->
    (Format.formatter -> 'a -> unit) ->
    Format.formatter -> 'a t -> unit
end
```

Chapter 13

Module `User`: Custom operations for MTBDDs

```
module User :
  sig
```

Important note: The OCaml closure defining the custom operation should not use free variables that may be modified and so impact its result: they would act as hidden parameters that are not taken into account in the memoization table.

If such hidden parameters are modified, the cache should be cleared with `Memo.clear`, if it is local, otherwise the global cache should be cleared with `Cudd.Man.flush`[5.3].

13.1 Types and values

13.1.1 Type of registered operations

```
type pid = Cudd.Custom.pid
```

Identifiers of closures used in shared memoization tables

```
type common = Cudd.Custom.common = {
  pid : pid ;
```

Identifiers for shared memoization tables

```
  arity : int ;
```

Arity of the operations

```
  memo : Cudd.Memo.t ;
```

Memoization table

```
}
```

Common information to all operations

```
val newpid : unit -> Cudd.Custom.pid
val make_common : ?memo:Cudd.Memo.t -> int -> common
```

13.2 Unary operations

```
type ('a, 'b) op1 = private ('a, 'b) Cudd.Custom.op1 = private {
  common1 : common ;
  closure1 : 'a -> 'b ;
```

Operation on leaves

```
}
val make_op1 : ?memo:Cudd.Memo.t -> ('a -> 'b) -> ('a, 'b) op1
```

Makes a binary operation, with the given memoization policy.

```
val apply_op1 : ('a, 'b) op1 -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
```

Example:

Assuming `type t = bool Vdd.t`, and corresponding diagrams `bdd:t` and `bdd2:t` (with type `bool`, it is safe to use directly VDDs, and it simplifies the examples).

We want to negate every leaf of `bdd1` and `bdd2`.

- We register the operation:
  `let op = make_op1 ~memo:(Cache (Cache.create 1)) (fun b -> not b);;`
- Later we apply it on `bdd1` and `bdd2` with
  `let res1 = apply_op1 op bdd1 and res2 = apply_op1 op bdd2;;`
- The local cache is reused between the two calls to `apply_op1`, which is nice if `bdd1` and `bdd2` share common nodes. The cache is automatically garbage collected when needed. But even if diagrams in the caches entries may be garbage collected, the cache itself takes memory. You can clear it with `Cache.clear` or `Memo.clear`.
- If `~memo::(Cache (Cache.create 1))` is replaced by `~memo::(Hash (Hash.create 1))`, then diagrams in the table are referenced and cannot be gabage collected. You should clear them explicitly with `Hash.clear` or `Memo.clear`.
- The third option is to use the CUDD global regular cache, which is automatically garbage collected when needed:
- If the operation is applied only once to one diagram, it is simpler to write `let res1 = map_op1 (fun b -> not b) bdd1;;`

13.3 Binary operations

```
type ('a, 'b, 'c) op2 = private ('a, 'b, 'c) Cudd.Custom.op2 = private {
  common2 : common ;
  closure2 : 'a -> 'b -> 'c ;
```

Operation on leaves

```
  ospecial2 : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t option) option ;
```

Special case operation

```
  commutative : bool ;
```

Is the operation commutative ?

```
  idempotent : bool ;
```

Is the operation idempotent (`op x x = x`) ?

```
}
val make_op2 :
  ?memo:Cudd.Memo.t ->
  ?commutative:bool ->
  ?idempotent:bool ->
  ?special:('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t option) ->
  ('a -> 'b -> 'c) -> ('a, 'b, 'c) op2
```

Makes a binary operation, with the given memoization policy.

`commutative` (default: `false`), when `true`, allows to optimize the cache usage (hence the speed) when the operation is commutative.

`idempotent` (default: `false`) allows similarly some optimization when op is idempotent: `op x x = x`. This makes sense only if `'a='b='c` (the case will never happens otherwise).

`?special` (default: `None`), if equal to `Some specialcase`, modifies `op` as follows: it is applied to every pair of node during the recursive descend, and if `specialcase vdda vddb = (Some vddc)`, then `vddc` is assumed to be the result of `map_op2 op vdda vddb`. This allows not to perform a full recursive descend when a special case is encountered. See the example below.

```
val apply_op2 :
  ('a, 'b, 'c) op2 -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t
```

Example:

Assuming as for unary operation example `type t = bool Vdd.t` and corresponding diagrams `bdd1:t` and `bdd2:t`.

We can compute their conjunction with

```
let res = map_op2
    ~commutative:true ~idempotent:true
    ~special:(fun bdd1 bdd2 ->
      if Vdd.is_cst bdd1 && Vdd.dval bdd1 = false then Some(bdd1)
      else if Vdd.is_cst bdd2 && Vdd.dval bdd2 = false then Some(bdd2)
      else None
    (fun b1 b2 -> b1 && b2) bdd1 bdd2;;
```

13.4 Ternary operations

```
type ('a, 'b, 'c, 'd) op3 = private ('a, 'b, 'c, 'd) Cudd.Custom.op3 = private {
  common3 : common ;
  closure3 : 'a -> 'b -> 'c -> 'd ;
```

Operation on leaves

```
  ospecial3 : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -
> 'd Cudd.Vdd.t option)
  option ;
```

Special cases

```
}
val make_op3 :
  ?memo:Cudd.Memo.t ->
  ?special:('a Cudd.Vdd.t ->
            'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t option) ->
  ('a -> 'b -> 'c -> 'd) -> ('a, 'b, 'c, 'd) op3
val apply_op3 :
  ('a, 'b, 'c, 'd) op3 ->
  'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t
```

Combine the two previous operations. if `?memo=None`, then a hash table is used, and cleared at the end.

Example:

Still assuming `type t = bool Vdd.t` and corresponding diagrams `bdd1:t`, `bdd2:t`, `bdd3:t`.

We can define `if-then-else` with

```
let res = map_op3
    ~special:(fun bdd1 bdd2 bdd3 ->
      if Vdd.is_cst bdd1
      then Some(if Vdd.dval bdd1 (* = true *) then bdd2 else bdd3)
      else None
    )
    (fun b1 b2 b3 -> if b1 then b2 else b3) bdd1 bdd2 bdd3
```

13.5 Nary operations

```
type ('a, 'b) opN = private ('a, 'b) Cudd.Custom.opN = private {
  commonN : common ;
  arityNbdd : int ;
  closureN : Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option ;
```

Operation on leaves

```
}
```

N-ary operation

```
val make_opN :
  ?memo:Cudd.Memo.t ->
  int ->
  int ->
  (Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option) ->
  ('a, 'b) opN
val apply_opN :
  ('a, 'b) opN ->
  Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t
type ('a, 'b) opG = private ('a, 'b) Cudd.Custom.opG = private {
  commonG : common ;
  arityGbdd : int ;
  closureG : Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option ;
  oclosureBeforeRec : (int * bool ->
   Cudd.Bdd.vt array ->
   'a Cudd.Vdd.t array -> Cudd.Bdd.vt array * 'a Cudd.Vdd.t array)
  option ;
  oclosureIte : (int -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t) option ;
}
```

N-ary general operation

```
val make_opG :
  ?memo:Cudd.Memo.t ->
  ?beforeRec:(int * bool ->
              Cudd.Bdd.vt array ->
              'a Cudd.Vdd.t array -> Cudd.Bdd.vt array * 'a Cudd.Vdd.t array) ->
```

```
  ?ite:(int -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'b Cudd.Vdd.t) ->
  int ->
  int ->
  (Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option) ->
  ('a, 'b) opG
val apply_opG :
  ('a, 'b) opG ->
  Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t
```

13.6 Binary tests

```
type ('a, 'b) test2 = private ('a, 'b) Cudd.Custom.test2 = private {
  common2t : common ;
  closure2t : 'a -> 'b -> bool ;
```

Test on leaves

```
  ospecial2t : ('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool option) option ;
```

Special cases

```
  symetric : bool ;
```

Is the relation symetric ?

```
  reflexive : bool ;
```

Is the relation reflexive ? (`test x x = true`) ?

```
}
val make_test2 :
  ?memo:Cudd.Memo.t ->
  ?symetric:bool ->
  ?reflexive:bool ->
  ?special:('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool option) ->
  ('a -> 'b -> bool) -> ('a, 'b) test2
```

Register a binary test, with the given memoization policy,
`symetric` (default: `false`), when `true`, allows to optimize the cache usage (hence the speed) when the relation is symetric.
`reflexive` (default: `false`) allows similarly some optimization when the relation is reflexive: `test x x = true`. This makes sense only if `'a='b` (the case will never happen otherwise).
`?special` (default: `None`) has the same semantics as for binary operation above.

```
val apply_test2 : ('a, 'b) test2 -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool
```

13.7 Quantification

```
type 'a exist = private 'a Cudd.Custom.exist = private {
  commonexist : common ;
  combineexist : ('a, 'a, 'a) op2 ;
```

Combining operation when a decision is eliminated

```
}
val make_exist : ?memo:Cudd.Memo.t -> ('a, 'a, 'a) op2 -> 'a exist
```

Make an existential quantification operation, with the given memoization policy, and the given underlying binary operation, assumed to be commutative and idempotent, that combines the two branch of the diagram when a decision is quantified out.

```
val apply_exist :
  'a exist -> supp:Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

Example:

Still assuming `type t = bool Vdd.t` and corresponding diagrams `bdd:t`

We define ordinary existential quantification with

```
let dor = make_op2 ~commutative:true ~idempotent:true ( || );;
  let exist = make_exist dor;;
  let res = apply_exist exist ~supp bdd;;
```

We can define ordinary universal quantification by replacing `||` with `&&`.

13.8 Quantification combined with intersection

```
type 'a existand = private 'a Cudd.Custom.existand = private {
  commonexistand : common ;
  combineexistand : ('a, 'a, 'a) op2 ;
```

Combining operation when a decision is eliminated

```
  bottomexistand : 'a ;
```

Value returned when intersecting with `Bdd.dfalse`

```
}
val make_existand :
  ?memo:Cudd.Memo.t ->
  bottom:'a -> ('a, 'a, 'a) op2 -> 'a existand
val apply_existand :
  'a existand ->
  supp:Cudd.Bdd.vt -> Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

`existand ~bottom op2 supp bdd f` is equivalent to `exist op2 supp (ite bdd f bottom)`.

The leaf operation `op2:'a -> 'a -> 'a` is assumed to be commutative, idempotent, and also `op2 f bottom = f`.

13.9 Quantification combined with unary operation

```
type ('a, 'b) existop1 = private ('a, 'b) Cudd.Custom.existop1 = private {
  commonexistop1 : common ;
  combineexistop1 : ('b, 'b, 'b) op2 ;
```

Combining operation when a decision is eliminated

```
  existop1 : ('a, 'b) op1 ;
```

Unary operations applied before elimination

```
}
val make_existop1 :
  ?memo:Cudd.Memo.t ->
  op1:('a, 'b) op1 ->
  ('b, 'b, 'b) op2 -> ('a, 'b) existop1
val apply_existop1 :
  ('a, 'b) existop1 ->
  supp:Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
```

Type of unary operation, conjunction and quantification

`existop1 op1 op2 supp f` is equivalent to `exist op2 supp (op1 f)`.

The leaf operation `op2:'b -> 'b -> 'b` is assumed to be commutative and idempotent.

13.10 Quantification combined with intersection and unary operation

```
type ('a, 'b) existandop1 = private ('a, 'b) Cudd.Custom.existandop1 = private {
  commonexistandop1 : common ;
  combineexistandop1 : ('b, 'b, 'b) op2 ;
```

Combining operation when a decision is eliminated

```
  existandop1 : ('a, 'b) op1 ;
```

Unary operations applied before elimination

```
  bottomexistandop1 : 'b ;
```

Value returned when intersecting with `Bdd.dfalse`

```
}
val make_existandop1 :
  ?memo:Cudd.Memo.t ->
  op1:('a, 'b) op1 ->
  bottom:'b -> ('b, 'b, 'b) op2 -> ('a, 'b) existandop1
val apply_existandop1 :
  ('a, 'b) existandop1 ->
  supp:Cudd.Bdd.vt -> Cudd.Bdd.vt -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
```

`existandop1 ~bottom op op1 supp bdd f` is equivalent to `exist op2 supp (ite bdd (op1 f) bottom))`.

The leaf operation `op2:'b -> 'b -> 'b` is assumed to be commutative, idempotent, and also `op2 f bottom = f`.

13.11 Clearing memoization tables

```
val clear_common : common -> unit
val clear_op1 : ('a, 'b) op1 -> unit
val clear_op2 : ('a, 'b, 'c) op2 -> unit
val clear_op3 : ('a, 'b, 'c, 'd) op3 -> unit
val clear_opN : ('a, 'b) opN -> unit
val clear_opG : ('a, 'b) opG -> unit
val clear_test2 : ('a, 'b) test2 -> unit
val clear_exist : 'a exist -> unit
val clear_existand : 'a existand -> unit
val clear_existop1 : ('a, 'b) existop1 -> unit
val clear_existandop1 : ('a, 'b) existandop1 -> unit
```

13.12 Map operations

These operations combine `make_opXXX` and `apply_opXXX` operations.

if `?memo=None`, then a hash table is used, and cleared at the end.

```
val map_op1 :
  ?memo:Cudd.Memo.t -> ('a -> 'b) -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
val map_op2 :
  ?memo:Cudd.Memo.t ->
```

```
      ?commutative:bool ->
      ?idempotent:bool ->
      ?special:('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t option) ->
      ('a -> 'b -> 'c) -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t
    val map_op3 :
      ?memo:Cudd.Memo.t ->
      ?special:('a Cudd.Vdd.t ->
                'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t option) ->
      ('a -> 'b -> 'c -> 'd) ->
      'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t
    val map_opN :
      ?memo:Cudd.Memo.t ->
      (Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t option) ->
      Cudd.Bdd.vt array -> 'a Cudd.Vdd.t array -> 'b Cudd.Vdd.t
    val map_test2 :
      ?memo:Cudd.Memo.t ->
      ?symetric:bool ->
      ?reflexive:bool ->
      ?special:('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool option) ->
      ('a -> 'b -> bool) -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> bool
  end
```

Chapter 14

Module Mapleaf: Lifting operation on leaves to operations on MTBDDs

```
module Mapleaf :
  sig
```

This module offers functions to lift operations on leaves to operations on MTBDDs on such leaves. Algorithmically, this is done by decomposing MTBDDs in lists of pairs `(guard,leaf)`.

An alternative, which may be more efficient but a bit less flexible, is to use functions of module `Cudd.User`[13].

In order to be usable with both modules `Cudd.Mtbdd`[11] and `Cudd.Mtbddc`[12], the signature of the functions of this modules use the type `'a Vdd.t`, but Vdds should not be used directly.

14.1 Global option

```
val restrict : bool Pervasives.ref
```

If `true`, simplifies in some functions MTBDDs using `Cudd.Mtbdd.restrict`[11.9] or `Cudd.Mtbddc.restrict`[12.9].

14.2 Functions of arity 1

In the following documentation, the pair `guard,leaf` is implicitly iterated on all such pairs contained in the argument MTBDD.

```
val mapleaf1 : ('a -> 'b) -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t
```

Return the MTBDD `\/ guard -> f leaf`

```
val retractivemapleaf1 :
  default:'a Cudd.Vdd.t ->
  (Cudd.Bdd.vt -> 'b -> Cudd.Bdd.vt * 'a) -> 'b Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

Assuming that the new guards delivered by the function `f` are disjoint, return the MTBDD `default \/ (\/ nguard -> nleaf)` with `(nguard,nleaf) = f guard leaf`.

```
val expansivemapleaf1 :
  default:'a Cudd.Vdd.t ->
  merge:('a Cudd.Vdd.t -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t) ->
  (Cudd.Bdd.vt -> 'b -> Cudd.Bdd.vt * 'a) -> 'b Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

Same as above, but with \/ replaced by merge (supposed to be commutative and associative).

```
val combineleaf1 :
  default:'a ->
  combine:('b -> 'a -> 'a) -> (Cudd.Bdd.vt -> 'c -> 'b) -> 'c Cudd.Vdd.t -> 'a
```

Generic function, instanciated above. The result acc (kind of accumulator) is initialized with default, to which one progressively add combine acc (f guard leaf).

combine should not be sensitive to the order in which one iterates on guards and leaves.

14.3 Functions of arity 2

In the following documentation, the pair guard1,leaf1 (resp. guard2,leaf2) is implicitly iterated on all such pairs contained in the first (resp. second) argument MTBDD.

```
val mapleaf2 :
  ('a -> 'b -> 'c) -> 'a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t
```

Return the MTBDD \/ guard1 /\ guard2 -> f leaf1 leaf2

```
val retractivemapleaf2 :
  default:'a Cudd.Vdd.t ->
  (Cudd.Bdd.vt -> 'b -> 'c -> Cudd.Bdd.vt * 'a) ->
  'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

Assuming that the new guards delivered by the function f are disjoint, return the MTBDD default \/ (\/ nguard -> nleaf) with (nguard,nleaf) = f (guard1 /\ guard2) leaf1 leaf2.

```
val expansivemapleaf2 :
  default:'a Cudd.Vdd.t ->
  merge:('a Cudd.Vdd.t -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t) ->
  (Cudd.Bdd.vt -> 'b -> 'c -> Cudd.Bdd.vt * 'a) ->
  'b Cudd.Vdd.t -> 'c Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

Same as above, but with \/ replaced by merge (supposed to be commutative and associative).

```
val combineleaf2 :
  default:'a ->
  combine:('b -> 'a -> 'a) ->
  (Cudd.Bdd.vt -> 'c -> 'd -> 'b) -> 'c Cudd.Vdd.t -> 'd Cudd.Vdd.t -> 'a
```

Generic function, instanciated above. The result acc (kind of accumulator) is initialized with default, to which one progressively add combine acc (f (guard1 /\ guard2) leaf1 leaf2).

combine should not be sensitive to the order in which one iterates on guards and leaves.

14.4 Functions on arrays

In the following documentation, tguard,tleaves denotes resp. the conjunctions of guards (of the array of MTBDD) and the associated array of leaves.

```
val combineleaf_array :
  default:'a ->
  combine:('b -> 'a -> 'a) ->
  tabsorbant:('c -> bool) option array ->
  (Cudd.Bdd.vt -> 'c array -> 'b) -> 'c Cudd.Vdd.t array -> 'a
```

Generic function,. The result `acc` (kind of accumulator) is initialized with `default`, to which one progressively add `combine acc (f (/\ tguard) tleaves)`.

The arrays are assumed to be non-empty.

If for some `i`, `tabsorbant.(i)=Some abs` and `absorbant tleaves.(i)=true`, then `f (/\ tguard) tleaves` is assumed to return `default` (this allows optimisation).

`combine` should not be sensitive to the order in which one iterates on guards and leaves.

```
val combineleaf1_array :
  default:'a ->
  combine:('b -> 'a -> 'a) ->
  ?absorbant:('c -> bool) ->
  tabsorbant:('d -> bool) option array ->
  (Cudd.Bdd.vt -> 'c -> 'd array -> 'b) ->
  'c Cudd.Vdd.t -> 'd Cudd.Vdd.t array -> 'a
```

```
val combineleaf2_array :
  default:'a ->
  combine:('b -> 'a -> 'a) ->
  ?absorbant1:('c -> bool) ->
  ?absorbant2:('d -> bool) ->
  tabsorbant:('e -> bool) option array ->
  (Cudd.Bdd.vt -> 'c -> 'd -> 'e array -> 'b) ->
  'c Cudd.Vdd.t -> 'd Cudd.Vdd.t -> 'e Cudd.Vdd.t array -> 'a
```

Functions similar to `combineleaf_array`, but in which the first (resp. first and second) leaves (and MTBDD) type may be different.

14.5 Internal functions

```
val combineretractive : Cudd.Bdd.vt * 'a -> 'a Cudd.Vdd.t -> 'a Cudd.Vdd.t
```

`combinetractive (guard,leaf) vdd = Vdd.ite guard leaf vdd`. Used in cases where `guard` and the guard of "interesting things" in `vdd` are disjoint, hence the name.

```
val combineexpansive :
  default:'a Cudd.Vdd.t ->
  merge:('a Cudd.Vdd.t -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t) ->
  Cudd.Bdd.vt * 'a -> 'b Cudd.Vdd.t -> 'c Cudd.Vdd.t
```

`combineexpansive ~default ~merge (guard,leaf) vdd = merge (Vdd.ite guard leaf default) vdd`. Implements in some way an "union" of `(guard,leaf)` and `vdd`.

```
end
```

Chapter 15

Module Add: MTBDDs with floats (CUDD ADDs)

```
module Add :
  sig
```

```
    type t
```

Abstract type for ADDs (that are necessarily attached to a manager of type `Man.d Man.t`). Objects of this type contains both the top node of the ADD and the manager to which the node belongs. The manager can be retrieved with `Cudd.Add.manager`[15.1]. Objects of this type are automatically garbage collected.

```
    type add =
      | Leaf of float
```

Terminal value

```
      | Ite of int * t * t
```

Decision on CUDD variable

Public type for exploring the abstract type `t`

15.1 Extractors

```
    val manager : t -> Cudd.Man.dt
```

Returns the manager associated to the ADD

```
    val is_cst : t -> bool
```

`Cudd_IsConstant`[`http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_IsConstant`]. Is the ADD constant ?

```
    val topvar : t -> int
```

`Cudd_NodeReadIndex`[`http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_NodeReadIndex`]. Returns the index of the ADD (65535 for a constant ADD)

```
    val dthen : t -> t
```

Cudd_T[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_T]. Returns the positive subnode of the ADD

```
val delse : t -> t
```

Cudd_E[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_E]. Returns the negative subnode of the ADD

```
val dval : t -> float
```

Cudd_V[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_V]. Returns the value of the assumed constant ADD

```
val cofactors : int -> t -> t * t
```

Returns the positive and negative cofactor of the ADD wrt the variable

```
val cofactor : t -> Cudd.Bdd.dt -> t
```

Cudd_Cofactor[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Cofactor]. `cofactor add cube` evaluates `add` on the cube `cube`

```
val inspect : t -> add
```

Decomposes the top node of the ADD

15.2 Supports

```
val support : t -> Cudd.Bdd.dt
```

Cudd_Support[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Support]. Returns the support (positive cube) of the ADD

```
val supportsize : t -> int
```

Cudd_SupportSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_SupportSize]. Returns the size of the support of the ADD

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

Cuddaux_IsVarIn. Does the given variable belong to the support of the ADD ?

```
val vectorsupport : t array -> Cudd.Bdd.dt
```

Cudd_VectorSupport[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_VectorSupport]. Returns the support of the array of ADDs.

Raises a `Failure` exception in case where the array is of size 0 (in such case, the manager is unknown, and we cannot return an empty support).

```
val vectorsupport2 : Cudd.Bdd.dt array -> t array -> Cudd.Bdd.dt
```

Cudd_VectorSupport[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_VectorSupport]. Returns the support of the BDDs and ADDs arrays.

Raises a `Failure` exception when both arrays are of size 0 (in such case, the manager is unknown, and we cannot return an empty support).

15.3 Classical operations

```
val cst : Cudd.Man.dt -> float -> t
```

Cudd_addConst[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addConst].
Return a constant ADD with the given value.

```
val background : Cudd.Man.dt -> t
```

Cuddaux_addIte/Cudd_addIte[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addIte].
If-then-else operation, with the condition being a BDD.

```
val ite : Cudd.Bdd.dt -> t -> t -> t
val ite_cst : Cudd.Bdd.dt -> t -> t -> t option
```

Cuddaux_addIteConstant/Cudd_addIteConstant[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html
If-then-else operation, which succeeds only if the resulting node belongs to one of the two
ADD.

```
val eval_cst : t -> Cudd.Bdd.dt -> t option
```

Cuddaux_addEvalConst/Cudd_addEvalConst[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cud

```
val compose : int -> Cudd.Bdd.dt -> t -> t
```

Cuddaux_addCompose/Cudd_addCompose[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_ad
Substitutes the variable with the BDD in the ADD.

```
val vectorcompose : ?memo:Cudd.Memo.t -> Cudd.Bdd.dt array -> t -> t
```

Cuddaux_addVectorCompose/Cudd_addVectorCompose[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.
Parallel substitution of every variable `var` present in the manager by the BDD `table.(var)`
in the ADD. You can optionnally control the memoization policy, see `Cudd.Memo`[4].

15.4 Variable mapping

```
val varmap : t -> t
```

Cuddaux_addVarMap/Cudd_bddVarMap[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_bddV
Permutes the variables as it has been specified with `Cudd.Man.set_varmap`[5.3].

```
val permute : ?memo:Cudd.Memo.t -> t -> int array -> t
```

Cudd_addPermute[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addPermute].
Permutes the variables as it is specified by `permut` (same format as in
`Cudd.Man.set_varmap`[5.3]). You can optionnally control the memoization policy, see
`Cudd.Memo`[4].

15.5 Logical tests

```
val is_equal : t -> t -> bool
```

Equality test
Variation of
Cudd_EquivDC[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_EquivDC].
Are the two ADDs equal when the BDD (careset) is true ?

```
val is_equal_when : t -> t -> Cudd.Bdd.dt -> bool
```

```
val is_eval_cst : t -> Cudd.Bdd.dt -> float option
```

Variation of
Cuddaux_addEvalConst/Cudd_addEvalConst[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cud
Is the ADD constant when the BDD (careset) is true, and in this case what is its value ?

```
val is_ite_cst : Cudd.Bdd.dt -> t -> t -> float option
```

Is the result of ite constant, and if it is the case, what is its value ?

15.6 Structural information

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

Cudd_DagSize[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_DagSize].
Size if the ADD as a graph (the number of nodes).

```
val nbpaths : t -> float
```

Cudd_CountPath[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountPath].
Number of paths in the ADD from the root to the leafs.

```
val nbnonzeropaths : t -> float
```

Cudd_CountPathsToNonZero[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountPathsTo
Number of paths in the ADD from the root to non-zero leaves.

```
val nbminterms : int -> t -> float
```

Cudd_CountMinterm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountMinterm].
Number of minterms of the ADD knowing that it depends on the given number of variables.

```
val density : int -> t -> float
```

Cudd_Density[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Density].
Density of the ADD, which is the ratio of the number of minterms to the number of nodes.
The ADD is assumed to depend on nvars variables.

```
val nbleaves : t -> int
```

Cudd_CountLeaves[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_CountLeaves].
Number of leaves.

15.7 Iterators

```
val iter_cube : (Cudd.Man.tbool array -> float -> unit) -> t -> unit
```

Similar to Cudd.Bdd.iter_cube[6.9]

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

Similar to Cudd.Bdd.iter_node[6.9]

15.8 Leaves and guards

```
val guard_of_node : t -> t -> Cudd.Bdd.dt
```

Cuddaux_addGuardOfNode. guard_of_node f node returns the sum of the paths leading from the root node f to the node node of f.

```
val guard_of_nonbackground : t -> Cudd.Bdd.dt
```

Guard of non background leaves

```
val nodes_below_level : t -> int option -> int -> t array
```

Cuddaux_NodesBelowLevel. nodes_below_level f olevel max returns all (if max<=0), otherwise at most max nodes pointed by the ADD, indexed by a variable of level greater or equal than level, and encountered first in the top-down exploration (i.e., whenever a node is collected, its sons are not collected). If olevel=None, then only constant nodes are collected. The background node may be in the result.

```
val guard_of_leaf : t -> float -> Cudd.Bdd.dt
```

Guard of the given leaf

```
val leaves : t -> float array
```

Returns the set of leaf values (excluding the background value)

```
val pick_leaf : t -> float
```

Picks (but not randomly) a non background leaf. Return None if the only leaf is the background leaf.

```
val guardleafs : t -> (Cudd.Bdd.dt * float) array
```

Returns the set of leaf values together with their guard in the ADD

15.9 Minimizations

See Cudd.Bdd.constrain[6.12], Cudd.Bdd.tdconstrain[6.12], Cudd.Bdd.restrict[6.12], Cudd.Bdd.tdrestrict[6.12]

```
val constrain : t -> Cudd.Bdd.dt -> t
```

```
val tdconstrain : t -> Cudd.Bdd.dt -> t
```

```
val restrict : t -> Cudd.Bdd.dt -> t
```

```
val tdrestrict : t -> Cudd.Bdd.dt -> t
```

15.10 Conversions

```
val of_bdd : Cudd.Bdd.dt -> t
```

Cudd_BddToAdd[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_BddToAdd].
Conversion from BDD to 0-1 ADD

```
val to_bdd : t -> Cudd.Bdd.dt
```

Cudd_addBddPattern[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addBddPattern].
Conversion from ADD to BDD by replacing all leaves different from 0 by true.

```
val to_bdd_threshold : float -> t -> t
```

Cudd_addBddThreshold[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addBddThreshold].
Conversion from ADD to BDD by replacing all leaves greater than or equal to the threshold by true.

```
val to_bdd_strictthreshold : float -> t -> t
```

Cudd_addBddStrictThreshold[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addBddStri
Conversion from ADD to BDD by replacing all leaves strictly greater than the threshold by true.

```
val to_bdd_interval : float -> float -> t -> t
```

Cudd_addBddInterval[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addBddInterval].
Conversion from ADD to BDD by replacing all leaves in the interval by true.

15.11 Quantifications

```
val exist : Cudd.Bdd.dt -> t -> t
```

Variation of
Cudd_addExistAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addExistAbstrac
Abstracts all the variables in the cube from the ADD by summing over all possible values taken by those variables.

```
val forall : Cudd.Bdd.dt -> t -> t
```

Variation of
Cudd_addUnivAbstract[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addUnivAbstract].
Abstracts all the variables in the cube from the ADD by taking the product over all possible values taken by those variables.

15.12 Algebraic operations

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

Cudd_addLeq[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addLeq].

```
val add : t -> t -> t
```

Cudd_addPlus[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addPlus].

```
val sub : t -> t -> t
```

Cudd_addMinus[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addMinus].

```
val mul : t -> t -> t
```

Cudd_addTimes[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addTimes].

```
val div : t -> t -> t
```

Cudd_addDivide[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addDivide].

```
val min : t -> t -> t
```

Cudd_addMinimum[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addMinimum].

```
val max : t -> t -> t
```

Cudd_addMaximum[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addMaximum].

```
val agreement : t -> t -> t
```

Cudd_addAgreement[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addAgreement].

```
val diff : t -> t -> t
```

Cudd_addDiff[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addDiff].

```
val threshold : t -> t -> t
```

Cudd_addThreshold[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addThreshold].

```
val setNZ : t -> t -> t
```

Cudd_addSetNZ[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addSetNZ].

```
val log : t -> t
```

Cudd_addLog[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addLog].

15.13 Matrix operations

```
val matrix_multiply : int array -> t -> t -> t
```

Variation of
Cudd_addMatrixMultiply[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addMatrixMulti
`matrix_multiply z A B` performs matrix multiplication of `A` and `B`, with `z` being the summation variables, which means that they are used to refer columns of `A` and to rows of `B`.

```
val times_plus : int array -> t -> t -> t
```

Variation of Cudd_addTimesPlus[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addTimesPlus].

```
val triangle : int array -> t -> t -> t
```

Variation of Cudd_addTriangle[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_addTriangle].

15.14 User operations

15.14.1 By decomposition into guards and leaves

```
val mapleaf1 : default:t ->
  (Cudd.Bdd.dt -> float -> float) -> t -> t
val mapleaf2 :
  default:t ->
  (Cudd.Bdd.dt -> float -> float -> float) ->
  t -> t -> t
```

15.14.2 By using CUDD cache

Consult Cudd.User[13] for explanations.

Type of operations

```
type op1 = (float, float) Cudd.Custom.op1
type op2 = (float, float, float) Cudd.Custom.op2
type op3 = (float, float, float, float) Cudd.Custom.op3
type opN = {
  commonN : Cudd.Custom.common ;
  closureN : Cudd.Bdd.dt array -> t array -> t option ;
  arityNbdd : int ;
}
type opG = {
  commonG : Cudd.Custom.common ;
  arityGbdd : int ;
  closureG : Cudd.Bdd.dt array -> t array -> t option ;
  oclosureBeforeRec : (int * bool ->
   Cudd.Bdd.dt array ->
   t array -> Cudd.Bdd.dt array * t array)
  option ;
  oclosureIte : (int -> t -> t -> t) option ;
}
type test2 = (float, float) Cudd.Custom.test2
type exist = float Cudd.Custom.exist
type existand = float Cudd.Custom.existand
type existop1 = (float, float) Cudd.Custom.existop1
type existandop1 = (float, float) Cudd.Custom.existandop1
val make_op1 : ?memo:Cudd.Memo.t -> (float -> float) -> op1
```

Making operations

```
val make_op2 :
  ?memo:Cudd.Memo.t ->
  ?commutative:bool ->
  ?idempotent:bool ->
  ?special:(t -> t -> t option) ->
  (float -> float -> float) -> op2

val make_op3 :
  ?memo:Cudd.Memo.t ->
  ?special:(t -> t -> t -> t option) ->
  (float -> float -> float -> float) -> op3

val make_opN :
  ?memo:Cudd.Memo.t ->
  int ->
  int ->
  (Cudd.Bdd.dt array -> t array -> t option) -> opN

val make_opG :
  ?memo:Cudd.Memo.t ->
  ?beforeRec:(int * bool ->
              Cudd.Bdd.dt array ->
              t array -> Cudd.Bdd.dt array * t array) ->
  ?ite:(int -> t -> t -> t) ->
  int ->
  int ->
  (Cudd.Bdd.dt array -> t array -> t option) -> opG

val make_test2 :
  ?memo:Cudd.Memo.t ->
  ?symetric:bool ->
  ?reflexive:bool ->
  ?special:(t -> t -> bool option) ->
  (float -> float -> bool) -> test2

val make_exist : ?memo:Cudd.Memo.t -> op2 -> exist

val make_existand : ?memo:Cudd.Memo.t -> bottom:float -> op2 -> existand

val make_existop1 : ?memo:Cudd.Memo.t -> op1:op1 -> op2 -> existop1

val make_existandop1 :
  ?memo:Cudd.Memo.t ->
  op1:op1 -> bottom:float -> op2 -> existandop1
```

Clearing memoization tables

```
val clear_op1 : op1 -> unit

val clear_op2 : op2 -> unit

val clear_op3 : op3 -> unit

val clear_opN : opN -> unit

val clear_opG : opG -> unit

val clear_test2 : test2 -> unit

val clear_exist : exist -> unit

val clear_existand : existand -> unit

val clear_existop1 : existop1 -> unit

val clear_existandop1 : existandop1 -> unit
```

Applying operations

```
val apply_op1 : op1 -> t -> t
val apply_op2 : op2 -> t -> t -> t
val apply_op3 : op3 -> t -> t -> t
val apply_opN : opN -> Cudd.Bdd.dt array -> t array -> t
val apply_opG : opG -> Cudd.Bdd.dt array -> t array -> t
val apply_test2 : test2 -> t -> t -> bool
val apply_exist : exist -> supp:Cudd.Bdd.dt -> t -> t
val apply_existand : existand ->
  supp:Cudd.Bdd.dt -> Cudd.Bdd.dt -> t -> t
val apply_existop1 : existop1 -> supp:Cudd.Bdd.dt -> t -> t
val apply_existandop1 :
  existandop1 ->
  supp:Cudd.Bdd.dt -> Cudd.Bdd.dt -> t -> t
```

Map functions

```
val map_op1 : ?memo:Cudd.Memo.t -> (float -> float) -> t -> t
val map_op2 :
  ?memo:Cudd.Memo.t ->
  ?commutative:bool ->
  ?idempotent:bool ->
  ?special:(t -> t -> t option) ->
  (float -> float -> float) -> t -> t -> t
val map_op3 :
  ?memo:Cudd.Memo.t ->
  ?special:(t -> t -> t -> t option) ->
  (float -> float -> float -> float) ->
  t -> t -> t -> t
val map_opN :
  ?memo:Cudd.Memo.t ->
  (Cudd.Bdd.dt array -> t array -> t option) ->
  Cudd.Bdd.dt array -> t array -> t
val map_test2 :
  ?memo:Cudd.Memo.t ->
  ?symetric:bool ->
  ?reflexive:bool ->
  ?special:(t -> t -> bool option) ->
  (float -> float -> bool) -> t -> t -> bool
```

15.15 Miscellaneous

```
val transfer : t -> Cudd.Man.d Cudd.Man.t -> t
```

Cuddaux_addTransfer/Cudd_bddTransfer[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_
Transfers a ADD to a different manager.

15.16 Printing

```
val _print : t -> unit
```

C printing function. The output may mix badly with the OCaml output.

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

Prints the minterms of the BDD in the same way as Cudd_Printminterm[http://vlsi.colorado.edu/~fabio/CUDD/cuddExtDet.html#Cudd_Printminterm].

```
val print_minterm :
  (Format.formatter -> int -> unit) ->
  (Format.formatter -> float -> unit) -> Format.formatter -> t -> unit
```

`print_minterm print_id print_leaf fmt bdd` prints the minterms of the BDD using `print_id` to print indices of variables and `print_leaf` to print leaf values.

```
val print :
  (Format.formatter -> int -> unit) ->
  (Format.formatter -> float -> unit) -> Format.formatter -> t -> unit
```

Prints a BDD by recursively decomposing it as monomial followed by a tree.

```
end
```

Index