Formula

October 3, 2007

Contents

Chapter 1

Introduction

This group of modules are "frontend" modules for manipulating BDDs and MTBDDs. It offers a higher level interface to the MLCUDDIDL interface to the C library CUDD, namely

- Functions for manipulating arrays of BDDs as CPU register, including most ALU (Arithmetic and Logical Unit) operations: [Bddredg].
- An interface for signed/unsigned integers encoded with BDDs, with features like autoresizing: [Bddint].
- An interface for enumerated types, with management of labels and types: [Bddenum];
- A module for BDD/MTBDD variables management: [Bddvar].
- A module for BDD/MTBDD output to file: [Bddoutput].

Next modules allows to manipulate formula and expressions mixing discrete and numerical types.

- Arithmetic expressions: [Arith].
- Generic decision diagrams: [CondDD].
- Binary decision diagrams with at leaves arithmetic expressions: [ArithDD].

The toplevel module of the library is [Formula]. I tis the recommended access point.

Chapter 2

Module `Bddreg` : Arithmetic with BDDs

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

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

This module requires the mlcuddidl library.

`type t = Bdd.t array`

type of arrays of bits

2.1 Logical operations

`val lnot : t -> t`

Logical negation (for all bits).

`val shift_left : Manager.t -> int -> t -> t * Bdd.t`

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

`val shift_right : Manager.t -> int -> t -> t * Bdd.t`

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

`val shift_right_logical : Manager.t -> int -> t -> t * Bdd.t`

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

`val extend : Manager.t -> signed:bool -> int -> t -> t`

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

2.2 Arithmetic operations

`val succ : Manager.t -> t -> t * Bdd.t`

Successor operation; returns the new register and the carry.

`val pred : Manager.t -> t -> t * Bdd.t`

Predecessor operation; returns the new register and the carry.

`val add : Manager.t -> t -> t -> t * Bdd.t * Bdd.t`

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

`val sub : Manager.t -> t -> t -> t * Bdd.t * Bdd.t`

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

`val neg : t -> t`

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

`val scale : int -> t -> t`

Multiplication by a positive constant.

`val ite : Bdd.t -> t -> t -> t`

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

2.3 Predicates

`val is_cst : t -> bool`

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

`val zero : Manager.t -> t -> Bdd.t`

Tests the register to zero.

`val equal : Manager.t -> t -> t -> Bdd.t`

Equality test.

`val greatereq : Manager.t -> t -> t -> Bdd.t`

Signed greater-or-equal test.

`val greater : Manager.t -> t -> t -> Bdd.t`

Signed strictly-greater-than test.

`val highereq : Manager.t -> t -> t -> Bdd.t`

Unsigned greater-or-equal test.

`val higher : Manager.t -> t -> t -> Bdd.t`

Unsigned strictly-greater-than test.

2.4 Constants: conversion and predicates

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

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

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

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

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

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

```
val equal_int : Manager.t -> t -> int -> Bdd.t
val greatereq_int : Manager.t -> t -> int -> Bdd.t
val greater_int : Manager.t -> t -> int -> Bdd.t
val highereq_int : Manager.t -> t -> int -> Bdd.t
val higher_int : Manager.t -> t -> int -> Bdd.t
```

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

2.5 Decomposition in guarded form

```
module Minterm :
  sig
```

```
type t = Manager.tbool array
```

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

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

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

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

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

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

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

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

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

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

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

```
  end
```

```
val guard_of_minterm : Manager.t -> t -> Minterm.t -> Bdd.t
```

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

```
val guard_of_int : Manager.t -> t -> int -> Bdd.t
```

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

```
val guardints : Manager.t -> signed:bool -> t -> (Bdd.t * int) list
```

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

2.6 Printing

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

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

```
val print_minterm :
  signed:bool ->
  (Format.formatter -> Bdd.t -> unit) -> Format.formatter -> t -> unit
```

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

Chapter 3

Module `Bddint` : Integer type for interpreted automaton

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

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

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

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

```
type t = {
  signed : bool ;
  reg : Bdd.t array ;
}
```

type of an enumerated variable

```
exception Typing of string
```

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

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

3.1 Conversion of integers

```
val extend : Manager.t -> int -> t -> t
```

3.2 Operations on integers

```
val neg : t -> t
val succ : t -> t
val pred : t -> t
```

```
val add : t -> t -> t
val sub : t -> t -> t
val shift_left : int -> t -> t
val shift_right : int -> t -> t
val scale : int -> t -> t
val ite : Bdd.t -> t -> t -> t
```

3.3 Predicates on integers

```
val is_cst : t -> bool
val zero : Manager.t -> t -> Bdd.t
val equal : Manager.t -> t -> t -> Bdd.t
val greatereq : Manager.t -> t -> t -> Bdd.t
val greater : Manager.t -> t -> t -> Bdd.t
```

3.4 Predicates involving constant integers

```
val of_int : Manager.t -> bool -> int -> int -> t
val to_int : t -> int
val equal_int : Manager.t -> t -> int -> Bdd.t
val greatereq_int : Manager.t -> t -> int -> Bdd.t
```

3.5 Decomposition in guarded form

```
module Minterm :
  sig
```

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

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

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

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

```
  end
```

```
val guard_of_int : Manager.t -> t -> int -> Bdd.t
```

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

```
val guardints : Manager.t -> t -> (Bdd.t * int) list
```

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

3.6 Printing

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

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

```
val print_minterm :
  (Format.formatter -> Bdd.t -> unit) -> Format.formatter -> t -> unit
```

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

Chapter 4

Module Var

```
type t = Symbol.t
val dummy : t
val print : Format.formatter -> t -> unit
val to_string : t -> string
module Set :
Sette.S  with type elt = t  and type t = Symbol.Set.t
module Map :
Mappe.S  with type key = t  and type 'a t = 'a Symbol.Map.t and module Setkey=Set
```

Chapter 5

Module Bddenum : Enumerated types with BDDs

5.1 Types

```
type label = Var.t
```

A label is just a name

```
type typ = [ ‘Benum of Var.t ]
```

A type is just a name

```
type typdef = [ ‘Benum of Var.t array ]
```

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

5.1.1 Datatype representing a BDD register of enumerated type

```
type t = {
  typ : Var.t ;
```

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

```
  reg : Bddreg.t ;
```

Value itself

```
}
```

5.1.2 Database

We need a global store where we register type names with their type definitions, and also an auxiliary table to efficiently associate types to labels.

```
class type [[> typ ], [> typdef ]] db =
  object

    val v_manager : Manager.t
    val mutable v_typdef : ([> Bddenum.typdef ] as ’a) Var.Map.t
```

Named types definitions

```
    val mutable v_vartyp : ([> Bddenum.typ ] as ’b) Var.Map.t
```

Associate to a label its type, assuming that a label is not shared among enumerated types

```
method manager : Manager.t
method typdef : 'a Var.Map.t
method vartyp : 'b Var.Map.t
method set_typdef : 'a Var.Map.t -> unit
method set_vartyp : 'b Var.Map.t -> unit
method mem_typ : Var.t -> bool
```

Is the type defined in the database ?

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

Is the label/var defined in the database ?

```
method typdef_of_typ : Var.t -> 'a
```

Return the definition of the type

```
method typ_of_var : Var.t -> 'b
```

Return the type of the label (or later variable)

```
method add_typ : Var.t -> 'a -> unit
```

Declaration of a new type

```
method private add_label : Var.t -> 'b -> unit
```

Declaration of a new label/var

```
end
```

5.2 Database

```
class [[> typ ], [> typdef ]] make_db : Manager.t -> [[[> typ ] as 'a, [> typdef ] as 'b]] db
```

Creation of a database

5.3 Constants and Operation(s)

```
val of_label : ([> typ ], [> typdef ]) #db ->
  label -> t
```

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

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

Does the register contain a constant value ?

```
val to_code : t -> int
```

Convert a constant register to its value as a code.

```
val to_label : ([> typ ], [> typdef ]) #db ->
  t -> label
```

Convert a constant register to its value as a label.

```
val equal_label : ([> typ ], [> typdef ]) #db ->
  t -> label -> Bdd.t
```

Under which condition the register is equal to the label ?

```
val equal : ([> typ ], [> typdef ]) #db ->
  t -> t -> Bdd.t
```

Under which condition the 2 registers are equal ?

```
val ite : Bdd.t -> t -> t -> t
```

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

5.4 Decomposition in guarded form

```
val guard_of_label : ([> typ ], [> typdef ]) #db ->
  t -> label -> Bdd.t
```

Return the guard of the label in the BDD register.

```
val guardlabels : ([> typ ], [> typdef ]) #db ->
  t -> (Bdd.t * label) list
```

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

5.5 Printing

```
val print_db : Format.formatter ->
  ([> typ ], [> typdef ]) #db -> unit
```

Print the database

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

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

```
val print_minterm :
  (Format.formatter -> Bdd.t -> unit) ->
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> t -> unit
```

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

5.6 Internal functions

```
val size_of_typ : ([> typ ], [> typdef ]) #db -> Var.t -> int
```

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

```
val maxcode_of_typ : ([> typ ], [> typdef ]) #db -> Var.t -> int
```

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

```
val mem_typcode : ([> typ ], [> typdef ]) #db -> Var.t -> int -> bool
```

Does the integer code some label of the given type ?

```
val labels_of_typ : ([> typ ], [> typdef ]) #db ->
  Var.t -> label array
```

Return the array of labels defining the type

```
val code_of_label : ([> typ ], [> typdef ]) #db -> label -> int
```

Return the code associated to the label

```
val label_of_typcode : ([> typ ], [> typdef ]) #db ->
  Var.t -> int -> label
```

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

```
module Minterm :
  sig
```

```
    val iter :
      ([> Bddenum.typ ], [> Bddenum.typdef ]) #Bddenum.db ->
      Var.t -> (Bddenum.label -> unit) -> Bddreg.Minterm.t -> unit
```

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

```
    val map :
      ([> Bddenum.typ ] as 'a, [> Bddenum.typdef ]) #Bddenum.db ->
      Var.t -> (Bddenum.label -> 'a) -> Bddreg.Minterm.t -> 'a list
```

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

```
  end
```

Chapter 6

Module Bddvar : BDDs and discrete variables

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

6.1 Datatypes

```
type expr = [ `Benum of Bddenum.t | `Bint of Bddint.t | `Bool of Bdd.t ]
```

Expression

```
type typdef = [ `Benum of Var.t array | `Bint of bool * int ]
```

Type defintion

```
type typ = [ `Benum of Var.t | `Bint of bool * int | `Bool ]
```

Types

```
type info = {
  tid : int array ;
```

(Sorted) array of involved BDD indices.

```
  var : expr ;
```

Expression representing the literal

```
}
```

Information associated to each variable

```
class type [[> typ ], [> typdef ]] db =
  object
```

```
    inherit Bddenum.db [5.1.2]
    val mutable v_bddindex : int
    val mutable v_bddincr : int
    val mutable v_idcondvar : Var.t MappeI.t
```

Associates to a BDD index the variable involved by it

```
    val mutable v_varinfo : Bddvar.info Var.Map.t
```

info associated to variables

```
val mutable v_varset : Bdd.t Var.Map.t
```

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

```
val mutable v_print_external_idcondb : Format.formatter -> int * bool -> unit
```

Printing conditions not managed by Bddvar. By default, pp_print_int.

```
method bddindex : int
method bddincr : int
method idcondvar : Var.t MappeI.t
method varinfo : Bddvar.info Var.Map.t
method varset : Bdd.t Var.Map.t
method print_external_idcondb : Format.formatter -> int * bool -> unit
method set_bddindex : int -> unit
method set_bddincr : int -> unit
method set_idcondvar : Var.t MappeI.t -> unit
method set_varinfo : Bddvar.info Var.Map.t -> unit
method set_varset : Bdd.t Var.Map.t -> unit
method set_print_external_idcondb :
  (Format.formatter -> int * bool -> unit) -> unit
method var_of_idcond : int -> Var.t
```

Return the variable associated to the BDD identifier. Raise Not_found if it does not exist

```
method add_var : Var.t -> ([> Bddvar.typ ] as 'a) -> unit
```

Add a variable

```
method print_order : Format.formatter -> unit
```

Print the BDD variable ordering

```
end
```

Database

6.2 Database

```
class [[> typ ], [> typdef ]] make_db : Manager.t -> [[[> typ ] as 'a, [> typdef ] as 'b]] db
```

Create a new database. bddindex and bddincr initalized to 0 and 1.

6.3 Expressions

```
val typ_of_expr : expr -> [> typ ]
```

Type of an expression

```
val var : ([> typ ], [> typdef ]) #db -> Var.t -> expr
```

Expression representing the litteral var

```
val ite : Bdd.t -> expr -> expr -> expr
```

If-then-else operation

```
val rename :
  ([> typ ], [> typdef ]) #db ->
  expr -> (Var.t * Var.t) list -> expr
```

Variable renaming. The new variables should already have been declared

```
val substitute :
  ([> typ ], [> typdef ]) #db ->
  expr -> (Var.t * expr) list -> expr
```

Parallel substitution of variables by expressions

```
val exist : ([> typ ], [> typdef ]) #db ->
  expr -> Var.t list -> expr
```

Existential quantification of a set of variables

```
val forall : ([> typ ], [> typdef ]) #db ->
  expr -> Var.t list -> expr
```

Universal quantification of a set of variables

```
val cofactor : expr -> Bdd.t -> expr
val tdrestrict : expr -> Bdd.t -> expr
val support_cond : ([> typ ], [> typdef ]) #db -> expr -> Bdd.t
```

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

```
val vectorsupport_cond : ([> typ ], [> typdef ]) #db ->
  expr array -> Bdd.t
```

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

6.3.1 Miscellaneous

```
val cube_of_bdd : ([> typ ], [> typdef ]) #db -> Bdd.t -> Bdd.t
```

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

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

```
val iter_ordered :
  ([> typ ], [> typdef ]) #db ->
  (Var.t -> info -> unit) -> unit
```

Iter on all variables declared in the database

6.4 Printing

```
val print_db : Format.formatter -> ([> typ ], [> typdef ]) #db -> unit
```

Print the database

```
val print_typ : Format.formatter -> typ -> unit
```

Print a type

```
val print_typdef : Format.formatter -> typdef -> unit
```

Print a type definition

```
val print_expr :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> expr -> unit
```

Print an expression

```
val print_minterm :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> Manager.tbool array -> unit
```

Print a minterm

```
val print_bdd :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> Bdd.t -> unit
```

Print a BDD

```
val print_idd :
  ([> typ ], [> typdef ]) #db ->
  (Format.formatter -> int -> unit) -> Format.formatter -> Idd.t -> unit
```

Print an IDD, using the function to associate to leaf indices strings.

```
val print_idcondb :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> int * bool -> unit
```

Print the condition represented by the signed BDD index.

```
val print_idcond :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> int -> unit
```

Print the condition

6.4.1 Internal

```
val print_info : Format.formatter -> info -> unit
```

Print an object of type `info`

6.5 Internal functions

6.5.1 Permutation and composition

```
val permutation_of_rename :
  ([> typ ], [> typdef ]) #db ->
  (Var.t * Var.t) list -> int array
val composition_of_substitution :
  ([> typ ], [> typdef ]) #db ->
  (Var.t * expr) list -> Bdd.t array
val bddsupport : ([> typ ], [> typdef ]) #db -> Var.t list -> Bdd.t
val permute : expr -> int array -> expr
val compose : expr -> Bdd.t array -> expr
```

6.5.2 Access functions

```
val mem_id : ([> typ ], [> typdef ]) #db -> int -> bool
```

Is the BDD index managed by the database ?

```
val varinfo_of_id : ([> typ ], [> typdef ]) #db ->
  int -> Var.t * info
```

Return the variable and `info` involved by the BDD index

```
val print_id :
  ([> typ ], [> typdef ]) #db ->
  Format.formatter -> int -> unit
```

Print the name associated to the single BDD index. If `name` is the name of the associated variable, and `I` the rank of the argument in the register, then it prints

- `name` in case of a Boolean variable
- `nameI` otherwise.

```
val info_of_var : ([> typ ], [> typdef ]) #db -> Var.t -> info
```

`info` associated to the variable in `db`

```
val reg_of_expr : expr -> Bdd.t array
```

Return the underlying array of BDD representing the expression

6.6 Conversion to expressions

```
module Expr :
  sig
```

Syntax tree for printing

```
type atom =
  | Tbool of Var.t * bool
```

variable name and sign

```
  | Tint of Var.t * int list
```

variable name and list of possible values

```
  | Tenum of Var.t * Var.t list
```

variable name, possibly primed, and list of possible labels

Atom

```
type term =
  | Tatom of atom
```

A true boolean indicates a primed variable

```
  | Texternal of int * bool
```

Unregistered BDD identifier and a Boolean for possible negation

```
  | Tcst of bool
```

Boolean constant

Basic term

```
  type conjunction =
    | Conjunction of term list
```

Conjunction of terms. Empty list means true.

```
    | Cfalse
```

Conjunction

```
  type disjunction =
    | Disjunction of conjunction list
```

Disjunction of conjunctions. Empty list means false

```
    | Dtrue
```

Disjunction

```
  val term_of_vint : Var.t -> Bddint.t -> Bddreg.Minterm.t -> term
  val term_of_venum :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Var.t -> Bddenum.t -> Bddreg.Minterm.t -> term
  val term_of_idcondb :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    MappeI.key * bool -> term
  val bool_of_tbool : Manager.tbool -> bool
  val mand : term list Pervasives.ref -> term -> unit
```

Raises Exit if false value

```
  val conjunction_of_minterm :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Manager.tbool array -> conjunction
  val disjunction_of_bdd :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Bdd.t -> disjunction
  val print_term :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Format.formatter -> term -> unit
  val print_conjunction :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Format.formatter -> conjunction -> unit
  val print_disjunction :
    ([> Bddvar.typ ], [> Bddvar.typdef ]) #Bddvar.db ->
    Format.formatter -> disjunction -> unit
end
```

Chapter 7

Module Bddoutput : Output of BDDs/MTBDDs

7.1 Types

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

BIte(idcond,idnodeThen,signElse,idnodeElse)

```
  | BTrue
```

Terminal case. Not needed in principle

BDD node

```
type inode =
  | IIte of int * int * int
```

IIte(idcond,idnodeThen,idnodeElse)

```
  | ICst of int
```

Leaf

IDD node

```
type 'a db = {
  mutable cond : SetteI.t ;
```

Reachable conditions

```
  mutable ileaf : SetteI.t ;
```

Reachable IDD leafs

```
  bhash : (Bdd.t, int) Hashtbl.t ;
  ihash : (Idd.t, int) Hashtbl.t ;
  mutable blastid : int ;
  mutable ilastid : int ;
```

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

```
  mutable bdef : bnode MappeI.t ;
```

Global BDDs graph

```
  mutable idef : (inode * 'a) MappeI.t ;
```

Global IDDs graph

`mutable ileafattr : int -> 'a ;`

Synthetized attribute for a leaf IDD node

`mutable inodeattr : int -> 'a -> 'a -> 'a ;`

Synthetized attribute for a non-leaf IDD node

`}`

Database

7.2 Functions

`val make_db : (int -> 'a) -> (int -> 'a -> 'a -> 'a) -> 'a db`

Create a database for printing BDDs/IDDs.

`make_db leafattr nodeattr` creates a database with the attribute functions

`val clear_db : 'a db -> unit`

Clear the database

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

Output the BDD and return its identifier

`val idattr_of_idd : 'a db -> Idd.t -> int * 'a`

Output the IDD and return its identifier

`val iter_cond : 'a db -> (int -> unit) -> unit`

Iterate the function on all the registered conditions

`val iter_cond_ordered : 'a db -> Manager.t -> (int -> unit) -> unit`

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

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

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

`val iter_idef_ordered : 'a db -> (int -> inode * 'a -> unit) -> unit`

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

Chapter 8

Module Arith : Expressions

```
type typ = [ ‘Int | ‘Real ]
class type [> typ ] db =
  object

    method typ_of_var : Var.t -> ([> Arith.typ ] as ’a)
  end
```

8.1 Expressions

8.1.1 Linear expressions

```
module Lin :
  sig

    type term = Mpqf.t * Var.t
    type expr = {
      cst : Mpqf.t ;
      lterm : term list ;
    }
    val normalize : expr -> expr
    val compare_lterm : term list -> term list -> int
    val compare : expr -> expr -> int
    val var : Var.t -> expr
    val cst : Mpqf.t -> expr
    val add : expr -> expr -> expr
    val sub : expr -> expr -> expr
    val scale : Mpqf.t -> expr -> expr
    val negate : expr -> expr
    val support : expr -> Var.Set.t
    val rename : expr -> Var.t Var.Map.t -> expr
    val normalize_as_constraint : expr -> expr
    val print : Format.formatter -> expr -> unit
  end
```

8.1.2 Polynomial expressions

```
module Poly :
  sig
    type varexp = Var.t * int
    type monomial = varexp list
    type term = Mpqf.t * monomial
    type expr = term list
    val compare_varexp : varexp -> varexp -> int
    val compare_monomial : monomial -> monomial -> int
    val normalize_monomial : monomial -> monomial
    val normalize : expr -> expr
    val normalize_full : expr -> expr
    val compare : expr -> expr -> int
    val cst : Mpqf.t -> expr
    val var : Var.t -> expr
    val add : expr -> expr -> expr
    val sub : expr -> expr -> expr
    val scale : Mpqf.t * monomial -> expr -> expr
    val mul : expr -> expr -> expr
    val div : expr -> expr -> expr
    val negate : expr -> expr
    val support : expr -> Var.Set.t
    val rename : expr -> Var.t Var.Map.t -> expr
    val normalize_as_constraint : expr -> expr
    val print : Format.formatter -> expr -> unit
  end
```

8.1.3 Tree expressions

```
module Tree :
  sig
    type binop =
      | Add
      | Sub
      | Mul
      | Div
      | Mod
    type expr =
      | Cst of Mpqf.t
      | Ref of Var.t
      | Binop of binop * expr * expr
      | Other of Var.t * expr list
    val support : expr -> Var.Set.t
    val rename : expr -> Var.t Var.Map.t -> expr
    val print : Format.formatter -> expr -> unit
  end
```

8.1.4 Tree expressions

8.1.5 Conversions

```
val lin_of_poly : Poly.expr -> Lin.expr
val lin_of_tree : Tree.expr -> Lin.expr
val poly_of_tree : Tree.expr -> Poly.expr
val tree_of_lin : Lin.expr -> Tree.expr
val tree_of_poly : Poly.expr -> Tree.expr
```

8.1.6 General expressions and operations

```
type expr =
  | Lin of Lin.expr
  | Poly of Poly.expr
  | Tree of Tree.expr
val var : [> typ ] #db -> Var.t -> expr
val cst : Mpqf.t -> expr
val add : expr -> expr -> expr
val sub : expr -> expr -> expr
val mul : expr -> expr -> expr
val div : expr -> expr -> expr
val gmod : expr -> expr -> expr
val other : Var.t -> expr list -> expr
val negate : expr -> expr
val support : expr -> Var.Set.t
val rename : expr -> Var.t Var.Map.t -> expr
val normalize : expr -> expr
val equal : expr -> expr -> bool
val hash : expr -> int
val compare : expr -> expr -> int
val normalize_as_constraint : expr -> expr
val is_dependent_on_integer_only : [> typ ] #db -> expr -> bool
val typ_of_expr : [> typ ] #db -> expr -> [ `Int | `Real ]
val print : Format.formatter -> expr -> unit
val print_typ : Format.formatter -> [> typ ] -> unit
```

8.2 Constraints

```
module Condition :
  sig

    type typ =
      | EQ
      | SUPEQ
      | SUP
      | NEQ
    type t = typ * Arith.expr
```

```
    val make :
      < typ_of_var : Var.t -> [> `Int | `Real ]; .. > ->
      typ ->
      Arith.expr -> [ `Arith of t | `Bool of bool ]
    val negate : < typ_of_var : Var.t -> [> `Int | `Real ]; .. > ->
      t -> t
    val support : t -> Var.Set.t
    val print : Format.formatter -> t -> unit
    val compare : t -> t -> int
  end
```

Chapter 9

Module CondDD : Utility functions

```
type typ = [ `Benum of Var.t | `Bint of bool * int | `Bool | `Int | `Real ]
type typdef = Bddvar.typdef
type cond = [ `Arith of Arith.Condition.t ]
type ('a, 'b) ddexpr =
  | Leaf of 'a
  | Mtbdd of 'b
module HashCond :
Hashhe.S  with type key = cond
module DHashCond :
DHashhe.S  with module HashX=HashCond and module HashY=HashheIB
class type db =
  object
    inherit Bddvar.db [6.1]
    val v_cond : CondDD.DHashCond.t
    val mutable v_careset : Bdd.t
    method cond : CondDD.DHashCond.t
    method careset : Bdd.t
    method set_careset : Bdd.t -> unit
    method cond_of_idb : int * bool -> CondDD.cond
  end
class make_db : Manager.t -> db
val print_cond : Format.formatter -> cond -> unit
val print_typ : Format.formatter -> typ -> unit
val print_typdef : Format.formatter -> typdef -> unit
val print_db : Format.formatter -> #db -> unit
val cond_support : cond -> Var.Set.t
val condition :
  leaf_of_id:(int -> 'a) ->
  cond_of_leaf:('a -> [ `Arith of Arith.Condition.t | `Bool of bool ]) ->
  #db -> ('a, 'b) ddexpr -> Bdd.t
val substitute_ddexpr :
  substitute_leaf:((#db as 'a) ->
```

```
                    'b -> 'c Var.Map.t -> ('b, 'd) ddexpr) ->
  guardleafs:('d -> (Bdd.t * 'b) array) ->
  ite:('a ->
       Bdd.t ->
       ('b, 'd) ddexpr ->
       ('b, 'd) ddexpr -> ('b, 'd) ddexpr) ->
  'a ->
  ('b, 'd) ddexpr ->
  Bdd.t array option -> 'c Var.Map.t -> ('b, 'd) ddexpr
val compute_careset : #db -> unit
```

Chapter 10

Module ArithDD : Operations

```
module Add :
Mtbdd.S  with type leaf = Arith.expr
type t = (Arith.expr, Add.t) CondDD.ddexpr
val of_expr : [> ‘Arith of t ] -> t
val to_expr : t -> [> ‘Arith of t ]
val to_add : #CondDD.db -> t -> Add.t
```

Operations

```
val var : #CondDD.db -> Var.t -> t
val cst : Mpqf.t -> t
val add : #CondDD.db -> t -> t -> t
val mul : #CondDD.db -> t -> t -> t
val sub : #CondDD.db -> t -> t -> t
val div : #CondDD.db -> t -> t -> t
val gmod : #CondDD.db -> t -> t -> t
val other : #CondDD.db -> Var.t -> t list -> t
val negate : #CondDD.db -> t -> t
val ite : #CondDD.db -> Bdd.t -> t -> t -> t
val condition : #CondDD.db -> Arith.Condition.typ -> t -> Bdd.t
```

Tests

```
val supeq : #CondDD.db -> t -> Bdd.t
val sup : #CondDD.db -> t -> Bdd.t
val eq : #CondDD.db -> t -> Bdd.t
val cofactor : t -> Bdd.t -> t
val tdrestrict : t -> Bdd.t -> t
val support_cond : #CondDD.db -> t -> Bdd.t
val support_leaf : #CondDD.db -> t -> Var.Set.t
val substitute_expr :
  #CondDD.db -> Arith.expr -> [> ‘Arith of t ] Var.Map.t -> t
val substitute_cond :
  #CondDD.db ->
  Arith.Condition.t -> [> ‘Arith of t ] Var.Map.t -> Bdd.t
val print : #CondDD.db -> Format.formatter -> t -> unit
```

Chapter 11

Module Formula : Boolean expressions

```
type cond = CondDD.cond
type typ = CondDD.typ
type typdef = CondDD.typdef
type expr = [ `Arith of ArithDD.t
  | `Benum of Bddenum.t
  | `Bint of Bddint.t
  | `Bool of Bdd.t ]
type leaf = [ `Arith of Arith.expr ]
class type db =
  object
    method add_typ : Var.t -> Formula.typdef -> unit
    method add_var : Var.t -> Formula.typ -> unit
    method bddincr : int
    method bddindex : int
    method careset : Bdd.t
    method cond : CondDD.DHashCond.t
    method cond_of_idb : int * bool -> Formula.cond
    method idcondvar : Var.t MappeI.t
    method manager : Manager.t
    method mem_typ : Var.t -> bool
    method mem_var : Var.t -> bool
    method print_external_idcondb : Format.formatter -> int * bool -> unit
    method print_order : Format.formatter -> unit
    method set_bddincr : int -> unit
    method set_bddindex : int -> unit
    method set_careset : Bdd.t -> unit
    method set_idcondvar : Var.t MappeI.t -> unit
    method set_print_external_idcondb :
      (Format.formatter -> int * bool -> unit) -> unit
    method set_typdef : Formula.typdef Var.Map.t -> unit
```

```
    method set_varinfo : Bddvar.info Var.Map.t -> unit
    method set_varset : Bdd.t Var.Map.t -> unit
    method set_vartyp : Formula.typ Var.Map.t -> unit
    method typ_of_var : Var.t -> Formula.typ
    method typdef : Formula.typdef Var.Map.t
    method typdef_of_typ : Var.t -> Formula.typdef
    method var_of_idcond : int -> Var.t
    method varinfo : Bddvar.info Var.Map.t
    method varset : Bdd.t Var.Map.t
    method vartyp : Formula.typ Var.Map.t
  end

class make_db : Manager.t -> CondDD.db
val typ_of_expr : expr -> typ
val print_cond : Format.formatter -> cond -> unit
val print_typ : Format.formatter -> typ -> unit
val print_typdef : Format.formatter -> typdef -> unit
val print_expr : #db -> Format.formatter -> expr -> unit
val print_bdd : #db -> Format.formatter -> Bdd.t -> unit
val print_db : Format.formatter -> #db -> unit
module Bool :
  sig
    type t = Bdd.t
    val of_expr : Formula.expr -> t
    val to_expr : t -> Formula.expr
    val dtrue : #Formula.db -> t
    val dfalse : #Formula.db -> t
    val of_bool : #Formula.db -> bool -> t
    val var : #Formula.db -> Var.t -> t
    val dnot : #Formula.db -> t -> t
    val dand : #Formula.db -> t -> t -> t
    val dor : #Formula.db -> t -> t -> t
    val xor : #Formula.db -> t -> t -> t
    val nand : #Formula.db -> t -> t -> t
    val nor : #Formula.db -> t -> t -> t
    val nxor : #Formula.db -> t -> t -> t
    val eq : #Formula.db -> t -> t -> t
    val ite : #Formula.db ->
      t -> t -> t -> t
    val is_included_in : ?sem:bool -> #Formula.db -> t -> t -> bool
    val is_inter_empty : ?sem:bool -> #Formula.db -> t -> t -> bool
    val exists : ?sem:bool -> #Formula.db -> Var.t list -> t -> t
    val forall : ?sem:bool -> #Formula.db -> Var.t list -> t -> t
    val print : #Formula.db -> Format.formatter -> t -> unit
```

```
  end

module Bint :
  sig

     type t = Bddint.t
     val of_expr : Formula.expr -> t
     val to_expr : t -> Formula.expr
     val of_int : #Formula.db -> [> `Tbint of bool * int ] -> int -> t
     val var : #Formula.db -> Var.t -> t
     val neg : #Formula.db -> t -> t
     val succ : #Formula.db -> t -> t
     val pred : #Formula.db -> t -> t
     val add : #Formula.db -> t -> t -> t
     val sub : #Formula.db -> t -> t -> t
     val shift_left : #Formula.db -> int -> t -> t
     val shift_right : #Formula.db -> int -> t -> t
     val scale : #Formula.db -> int -> t -> t
     val ite : #Formula.db -> Bdd.t -> t -> t -> t
     val zero : #Formula.db -> t -> Bdd.t
     val eq : #Formula.db -> t -> t -> Bdd.t
     val eq_int : #Formula.db -> t -> int -> Bdd.t
     val supeq : #Formula.db -> t -> t -> Bdd.t
     val supeq_int : #Formula.db -> t -> int -> Bdd.t
     val sup : #Formula.db -> t -> t -> Bdd.t
     val print : #Formula.db -> Format.formatter -> t -> unit
  end

module Benum :
  sig

     type t = Bddenum.t
     val of_expr : Formula.expr -> t
     val to_expr : t -> Formula.expr
     val var : #Formula.db -> Var.t -> t
     val ite : #Formula.db -> Bdd.t -> t -> t -> t
     val eq : #Formula.db -> t -> t -> Bdd.t
     val eq_label : #Formula.db -> t -> Var.t -> Bdd.t
     val print : #Formula.db -> Format.formatter -> t -> unit
  end

module Arith :
  sig

     type t = ArithDD.t
     val of_expr : [> `Arith of t ] -> t
     val to_expr : t -> [> `Arith of t ]
     val var : #Formula.db -> Var.t -> t
     val cst : Mpqf.t -> t
```

```
    val add : #Formula.db -> t -> t -> t
    val mul : #Formula.db -> t -> t -> t
    val sub : #Formula.db -> t -> t -> t
    val div : #Formula.db -> t -> t -> t
    val gmod : #Formula.db -> t -> t -> t
    val other : #Formula.db -> Var.t -> t list -> t
    val negate : #Formula.db -> t -> t
    val ite : #Formula.db -> Bdd.t -> t -> t -> t
    val condition : #Formula.db -> Arith.Condition.typ -> t -> Bdd.t
    val supeq : #Formula.db -> t -> Bdd.t
    val sup : #Formula.db -> t -> Bdd.t
    val eq : #Formula.db -> t -> Bdd.t
    val substitute_expr :
      #Formula.db ->
      Arith.expr -> [> `Arith of t ] Var.Map.t -> t
    val substitute_cond :
      #Formula.db ->
      Arith.Condition.t -> [> `Arith of t ] Var.Map.t -> Bdd.t
    val print : #Formula.db -> Format.formatter -> t -> unit
  end

val eq : #db -> expr -> expr -> Bool.t
val ite : #db -> Bool.t -> expr -> expr -> expr
val compose_of_substitution :
  #db ->
  expr Var.Map.t -> Bdd.t array option * expr Var.Map.t
val substitute : #db -> expr -> expr Var.Map.t -> expr
val var : #db -> Var.t -> expr
val cofactor : expr -> Bdd.t -> expr
val tdrestrict : expr -> Bdd.t -> expr
val support_cond : #db -> expr -> Bdd.t
val vectorsupport_cond : #db -> expr list -> Bdd.t
val support : #db -> expr -> Var.Set.t
val print_expr : #db -> Format.formatter -> expr -> unit
val print_bdd : #db -> Format.formatter -> Bdd.t -> unit
val cleanup : #db -> SetteI.t -> expr list -> unit
```