Implémentation et vérification d'une bibliothèque pour la manipulation des formules logiques dans l'outil *Why3*

Gabriel Desfrene

juin 2023

Je réalise actuellement un stage au sein de l'équipe Toccata. Mon stage porte sur la vérification formelle des programmes via la plateforme *Why3* [1]. Ce stage se déroule en deux parties [2]. La première est consacrée à l'implémentation d'une petite bibliothèque sur les diagrammes de décision binaire puis à sa vérification formelle à l'aide de *Why3*. La deuxième partie, pas encore réalisée à ce jour, portera sur l'implémentation d'une transformation logique décrite dans un article de recherche [3]. Cette transformation sera implémentée à l'aide de l'API *Why3* et pourra donc être employée lors de la vérification formelle des programmes dans la plateforme.

1 La plateforme *Why3*

Why3 est une plateforme pour la vérification déductive des programmes. Cette plateforme est utilisable à l'aide du langage *WhyML* qui sert à la fois à la spécification [1] et l'implémentation des programmes. À l'aide de ces informations, la plateforme crée des obligations de preuves. La vérification de ces obligations assure alors que le programme implémenté correspond à la spécification donnée. Ces obligations de preuves peuvent être exportées à des démonstrateurs automatiques qui se chargeront de la preuve, ou à des assistants de preuves.

Lors de ce stage, la version 1.6.0 de *Why3* a été utilisée avec les démonstrateurs automatiques suivants : *Z3* [4] (version 4.12.2), *CVC5* [5] (version 1.0.5) et *Alt-Ergo* [6] (version 2.4.2).

2 Diagrammes de Décision Binaire

On présente ici brièvement la construction ainsi que l'intérêt des diagrammes de décision binaire. On pourra se référer à l'introduction d'Henrik Reif Andersen [7] pour une construction plus rigoureuse.

Les Diagrammes de Décision Binaire ou BDD [2] sont une méthode pour représenter les formules de logique propositionnelle. Comme l'indique le nom, pour chaque formule logique, on construit un diagramme dans lequel chaque nœud représente un choix pour la valeur d'une variable. Tous les nœuds possèdent exactement deux fils, correspondant aux deux valeurs possibles de la variable. La flèche pleine représente le cas où la variable associée prend la valeur vraie. La flèche en pointillé correspond, elle, au cas où cette variable prend la valeur fausse. On réalise cette construction pour chaque valeur possible des variables. Une fois toutes les variables ayant une valeur fixée, on encode la valeur de la formule à l'aide des deux symboles $\top$ et $\bot$ représentant chacun la valeur vraie et fausse. La figure 1 donne un exemple d'un BDD pour la formule [3] $x \vee y$.

1. La spécification d'un programme consiste à décrire son comportement dans un système formel. Cette description peut ensuite être utilisée pour déduire des propriétés sur les résultats du programme par exemple.
2. De l'anglais *Binary Decision Diagram*.
3. Pour deux formules A et B, on note $\neg A$ la négation de A, $A \wedge B$ leur conjonction et $A \vee B$ leur disjonction.

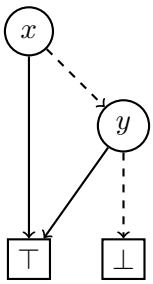

FIGURE 1 – Représentation de la formule $x \vee y$ par un BDD.

Cette méthode de représentation présente cependant un problème : une formule logique peut correspondre à deux BDD différents. C'est pourquoi on impose les conditions suivantes :

— On fixe un ordre arbitraire sur nos variables. Les variables de chaque BDD devront apparaître dans l'ordre croissant fixé. Un tel BDD est dit *ordonné*.
— On fusionne tous les nœuds qui représentent le même diagramme de décision. Dans ce cas, chaque nœud possédera deux fils nécessairement différents ; aucun nœud ne partagera la même variable et les deux mêmes fils avec un autre nœud du BDD. Un tel BDD est dit *réduit*.

Par abus de langage, BDD fait généralement référence à cette méthode de représentation des formules logiques en respectant les règles précédemment énoncées. La figure 2 illustre l'application de la règle de réduction.

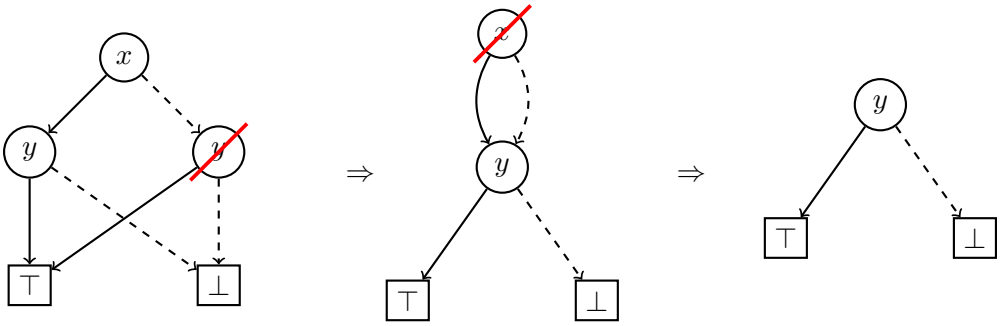

FIGURE 2 – Réduction d'un BDD.

Avec ces règles, les BDD possèdent une propriété très intéressante :

Chaque BDD représente de manière unique une formule de logique propositionnelle.

Cette propriété permet alors de résoudre de nombreux problèmes en très peu d'opérations :

— Tester l'équivalence de deux BDD est possible en un nombre constant d'opérations.
— Savoir si un BDD est satisfiable [1] ou une tautologie [2] est également possible en nombre constant d'opérations.
— Déterminer une affectation satisfiant un BDD est linéaire en son nombre de nœuds.
— Compter le nombre d'affectations satisfiant un BDD est également linéaire en son nombre de nœuds.

On procède alors à l'implémentation de ces opérations en *WhyML*.

3 Création d'une bibliothèque pour la manipulation de BDD

On implémente en *WhyML* une bibliothèque permettant de manipuler des BDD, dans l'optique de la vérifier formellement. La totalité du code produit, près de 800 lignes, est disponible sur un dépôt GitHub [8].

1. On dit qu'une formule logique est satisfiable s'il existe une affectation de chaque variable qui lui donne la valeur vraie.
2. Une formule logique est une tautologie si elle prend la valeur vraie dans toutes les interprétations possibles.

3.1 Architecture de la bibliothèque

Dans un souci d'optimisation des opérations effectuées, on procède au *Hash Consing* [1] des BDD construits [9]. Ce procédé permet d'assurer que chaque nœud est créé, au plus, une seule fois. Avant de construire un nouveau nœud, on vérifie s'il n'a pas été précédemment construit. Si c'est le cas, aucune allocation de mémoire n'est nécessaire, il suffit de retourner l'objet existant. Dans le cas contraire, un nouveau nœud est créé puis renvoyé.

Puisque cette méthode assure que chaque nœud est créé au plus une fois, il devient possible de numéroter son ordre d'apparition au long du programme. Cet entier, unique à chaque nœud construit, identifie de manière unique un BDD. Ainsi, dans le cas où la technique du *Hash Consing* est employée, l'égalité structurelle [2] des objets coïncide avec l'égalité physique [3] qui coïncide également avec l'égalité des identifiants de chaque BDD. Ainsi, pour chaque diagramme u et v construit suivant cette technique :

$$u \texttt{ == } v \iff u \texttt{ = } v \iff id(u) = id(v)$$

Vu ses propriétés, l'identifiant de chaque diagramme peut être utilisé comme clef de hachage. Cela assure alors un calcul de cette clef en un temps constant, puisque chaque diagramme est stocké en mémoire avec son identifiant. De plus, puisque chaque diagramme possède un identifiant unique, il n'y a pas de collision possible.

Cette technique assure également un partage maximal lors de la construction des BDD. Ainsi, pour deux BDD donnés, s'ils partagent un sous-diagramme similaire, c'est-à-dire deux sous-diagrammes structurellement égaux, alors ces deux BDD feront en réalité référence au même sous-diagramme en mémoire. Le calcul des résultats des opérations possibles sur les BDD pourra donc toujours se faire de manière optimale, sous réserve de *mémoïser* [4] chaque résultat.

3.2 Représentation des Diagrammes

Afin de représenter les BDD en *WhyML*, on utilise le type algébrique suivant :

```
1 type bdd =
2 | Bottom
3 | Top
4 | N int63 bdd bdd Peano.t (* N (var_id, true_branch, false_branch, node_id) *)
```

Deux types de données peu courants sont ici utilisés pour représenter des nombres.

- Le type `int63` est ici utilisé pour représenter les variables. On l'utilise afin de travailler sur les mêmes entiers que l'interprète OCaml : les entiers de 63bits signés. L'usage de ces entiers est possible à l'aide du module `mach.int.Int63` de la bibliothèque standard de *Why3* qui inclut également toutes les théories nécessaires pour raisonner sur ce type.
- Le type `Peano.t` correspond, comme son nom l'indique, à des entiers de Peano. Deux fonctions logiques sont alors disponibles : la valeur `zero` ainsi la fonction `succ` renvoyant le successeur d'un entier. Ce type, disponible dans la bibliothèque `mach.int.Peano`, donne la possibilité d'incrémenter une variable sans avoir à justifier le non-dépassement d'entier (*integer overflow*) [10]. En effet, en utilisant seulement l'opération `succ`, il est nécessaire de réaliser 2^{62} opérations en OCaml pour arriver à un dépassement d'entier. Cela prendrait quelques mois, voire quelques années sur ordinateur moderne.

L'usage de ces types permet une extraction du code en OCaml avec des entiers natifs : `int`. On appellera taille d'un BDD le nombre de nœuds distincts qui le composent.

1. L'algorithme *Hash-Life* de Bill Gosper, effectuant de manière très optimisée les calculs pour le *Jeu de la vie* de Conway, est connu pour utiliser cette technique.
2. Égalité portant sur les valeurs des objets, ici calculée de manière récursive. Elle correspond à l'opérateur de comparaison == en OCaml.
3. Égalité portant sur la position dans la mémoire de chaque objet. C'est l'opérateur de comparaison = en OCaml.
4. Technique utilisée en programmation où chaque résultat d'une fonction est enregistré afin de ne pas avoir à refaire ce calcul.

On définit également deux prédicats d'égalités `eq0` et `eq1`. Le premier travaille sur la racine du nœud en comparant leurs identifiants. Le second travaille à l'étage suivant. Le code en figure 3 décrit l'implémentation de ces prédicats.

```
(* Equality of rank 0 *)
let predicate eq0 (t1 : bdd) (t2 : bdd) : bool =
  eq (id t1) (id t2)

(* Equality of rank 1 *)
let predicate eq1 (t1 : bdd) (t2 : bdd) : bool =
  match t1, t2 with
    | Top, Top | Bottom, Bottom -> True
    | N var_1 true_1 false_1 _, N var_2 true_2 false_2 _ ->
          var_1 = var_2 && eq0 true_1 true_2 && eq0 false_1 false_2
    | _ -> False
  end
```

FIGURE 3 – Implémentation des prédicats `eq0` et `eq1`.

Un type `affectation` est également introduit ainsi qu'une fonction `value`. Ce type représente une fonction associant à chaque variable une valeur booléenne. La fonction `value` permet d'évaluer un BDD selon une affectation, son implémentation est décrite en figure 4.

```
(* Value of a BDD with a given affectation *)
let rec function value (f : affectation) (input : bdd) : bool =
  variant { input } (* For the termination proof *)
  match input with
    | Top -> True
    | Bottom -> False
    | N var_id true_b false_b _ ->
        if f var_id then value f true_b else value f false_b
  end
```

FIGURE 4 – Implémentation de la fonction `value`.

3.3 Modélisation d'une table de Hachage

La bibliothèque standard de *Why3* ne possède pas de théorie sur les tables de hachage prenant en paramètre un prédicat d'égalité. On modélise donc le comportement d'une telle structure de données.

Cette modélisation assure que toutes les clefs égales selon le prédicat d'égalité fourni pointent vers la même valeur. On raisonnera donc sur les classes d'équivalence de notre prédicat afin de prouver les diverses spécifications dans lesquelles cette structure interviendra.

Cette modélisation a été réalisée afin de garantir une extraction du code possible à l'aide du foncteur `Hashtbl.Make` d'OCaml. On trouvera en annexe A l'énoncé précis de cette modélisation.

3.4 Implémentation et Preuve

Maintenant que le type de données est défini et que l'on possède une modélisation des tables de hachage convenable, on commence à implémenter cette bibliothèque. On détaillera seulement les preuves concernant la création des nœuds ainsi que le calcul de la négation d'un BDD.

3.4.1 Création des nœuds : le module `BDD`

L'objectif de cette partie est d'implémenter et de prouver une fonction `create_node` qui, pour deux BDD réduits et ordonnés et une variable, renvoie un nouveau BDD, toujours réduit et ordonné, correspondant aux arguments. C'est également cette fonction qui assure le *Hash Consing* des BDD créés. Le lecteur pourra se référer au code disponible en annexe B pour une lecture en profondeur de l'implémentation et de la preuve. On exposera seulement les idées générales utilisées.

On introduit un nouveau type, `hctable`, dont l'objectif est de stocker chaque instance des BDD créés lors de l'exécution :

```
type hctable = { tbl : HCMap.t bdd; mutable next_id : Peano.t }
```

Le champ `tbl` est une table de hachage ayant pour clef des objets de type `bdd`. Cette table associe tous les nœuds possédant la même variable ainsi que les deux mêmes sous-diagrammes vers leur unique représentant, marqué par son identifiant unique. Le prédicat d'égalité `eq1` qui ignore l'identifiant des éléments comparés est donc utilisé pour cette table. L'unique représentant de chaque classe d'équivalence du prédicat `eq1` est reconnu par le fait qu'il est le seul à être associé à lui-même. C'est ce que le prédicat `mem_tbl` représente :

```
(* A value is mem_tbl in the table if it is in the table and linked to itself *)
predicate mem_tbl (tbl : HCMap.t bdd) (k : bdd) =
  HCMap.is_in tbl k /\ HCMap.val_of tbl k = k
```

Le champ `next_id` correspond au prochain identifiant disponible pour la création d'un nouveau nœud. On impose des invariants au type `hctable` pour assurer le caractère réduit et ordonné des BDD créés. Par exemple, le fait que les variables de chaque nœud apparaissent dans l'ordre croissant est traduit comme il suit :

```
(* Each BDD mem_tbl in table is Ordered [True Branch] *)
invariant { forall u : bdd. mem_tbl tbl u -> is_node (true_branch u) ->
              var u < var (true_branch u) }

(* Each BDD mem_tbl in table is Ordered [False Branch] *)
invariant { forall u : bdd. mem_tbl tbl u -> is_node (false_branch u) ->
              var u < var (false_branch u) }
```

De la même manière, des invariants sont introduits pour assurer le caractère réduit des BDD présents dans notre table. Pour des raisons de simplicité, on impose que tous les BDD présents dans la table soient uniquement des nœuds. La présence d'un nœud dans un objet du type `hctable` assurant sa construction correcte, on introduit le prédicat `well_formed` qui assure que le BDD donné, que ce soit un nœud ou une feuille ($\top$ ou $\bot$), est bien réduit et ordonné.

```
(* A value is well-formed (ie a ROBDD) if it is mem_tbl in the hash-consing table
   or a leaf *)
predicate well_formed (tbl : HCMap.t bdd) (k : bdd) =
  is_leaf k \/ mem_tbl tbl k
```

Deux lemmes importants sont ensuite énoncés. Ces lemmes assurent que les deux égalités introduites précédemment, `eq0` et `eq1`, coïncident avec l'égalité logique de *Why3* pour des diagrammes présents dans une table de *Hash Consing* :

```
(* With the hash-consing, for well-formed BDD, eq1 is the same as
   the logical equality *)
lemma eq1_match_logic :
  forall hc : hctable, u v : bdd. well_formed hc.tbl u -> well_formed hc.tbl v ->
    u = v <-> eq1 u v

(* With the hash-consing, for well-formed BDD, eq0 is the same as
   the logical equality *)
lemma eq0_match_logic :
  forall hc : hctable, u v : bdd. well_formed hc.tbl u -> well_formed hc.tbl v ->
    u = v <-> eq0 u v
```

Ces lemmes sont prouvés sans difficulté par les démonstrateurs automatiques. On passe désormais à l'implémentation de la fonction permettant la création d'un nouveau nœud. Les deux sous-diagrammes passés en argument doivent être bien formés. De plus, la variable donnée en argument ainsi que les variables des sous-diagrammes doivent respecter l'ordre des variables imposé. La fonction assure ensuite la construction correcte du nœud, sa présence dans la table de *Hash Consing* ainsi que la validité de cette dernière. Le code, nettoyé de ses (nombreuses) préconditions et postconditions, est présenté en figure 5.

```
let create_node (hc : hctable) (v : int63) (t : bdd) (f : bdd) : bdd =
  (* Préconditions et postconditions disponibles en annexe *)
  if eq0 t f then
    t
  else
    let bdd_candidate = N v t f Peano.zero in
    try
      HCMap.find hc.tbl bdd_candidate
    with HCMap.NotFound ->
      let new_node = N v t f hc.next_id in
      HCMap.put hc.tbl bdd_candidate new_node;
      hc.next_id <- Peano.succ hc.next_id;
      new_node
    end
```

FIGURE 5 – Code sans préconditions ni postconditions de la fonction `create_node`.

La preuve du comportement correct de cette fonction est réalisée sans difficulté par les démonstrateurs automatiques, en quelques secondes. Le tableau 1 résume le statut [1] des preuves réalisées par chaque démonstrateur pour prouver la spécification du module `BDD`.

1. Les cellules orange signifient que le démonstrateur n'a pas pu vérifier l'énoncé dans les limites imposées de temps (5.00 s) et de mémoire (1 Go). Les cellules grises signifient que le démonstrateur a abandonné la preuve.

Proof obligations	Alt-Ergo 2.4.2	CVC5 1.0.5	Z3 4.12.3
VC for hctable	0.02	0.03	0.01
bdd_is_unique	1.99	0.03	0.01
eq1_match_logic	0.06	0.03	0.01
eq0_match_logic	0.02	0.03	0.01
VC for create_hctable	0.02	0.03	0.01
VC for create_node	(5.00s)	3.82	0.27

TABLE 1 – Bilan des preuves automatiques pour le module BDD.

3.4.2 Négation d'un BDD : le module `Not`

L'objectif de cette partie est de calculer de manière optimale la négation d'un BDD. Afin de vérifier que le résultat renvoyé par la fonction calculant la négation, `apply`, on introduit un prédicat `is_not_bdd` décrivant le lien entre deux BDD, dont l'un est la négation de l'autre :

```
predicate is_not_bdd (not_bdd : bdd) (orig_bdd : bdd) =
  forall f : affectation. value f not_bdd = not value f orig_bdd
```

Vu le partage opéré lors de la construction de nos diagrammes, *mémoïser* leur négation peut être très intéressant dans le cas où de nombreux calculs de négation sont effectués. On introduit donc le type `memoMap` qui s'assure d'enregistrer les résultats de ces calculs. Des invariants lui sont associés afin de prouver le contrat de la fonction `apply` :

```
type memoMap = { hc : hctable; m : MemoMap.t bdd }

(* Top and Bottom are not in the memoisation table *)
invariant { not MemoMap.is_in m Top /\ not MemoMap.is_in m Bottom }

(* All BDD in the memoisation table have their id < than the next_id of
   the hash consing Table *)
invariant { forall u : bdd. MemoMap.is_in m u -> id u < hc.next_id }

(* All key of the memoisation table have their value in the hash consing table *)
invariant { forall u : bdd. MemoMap.is_in m u ->
              well_formed hc.tbl (MemoMap.val_of m u) }

(* All values are images by not operation of their key *)
invariant { forall u : bdd. mem_tbl hc.tbl u -> MemoMap.is_in m u ->
              is_not_bdd (MemoMap.val_of m u) u }

(* The variable of the keys are <= than the variable of the value *)
invariant { forall u : bdd. mem_tbl hc.tbl u -> MemoMap.is_in m u ->
              is_node (MemoMap.val_of m u) -> var u <= var (MemoMap.val_of m u) }
```

Le champ `hc` correspond à une table de *Hash Consing*, comme défini dans la partie précédente. Il a premièrement été inclus dans ce type dans le but d'écrire les invariants à respecter. Sa présence s'est ensuite avérée algorithmiquement intéressante pour la création de nouveaux nœuds dans la fonction `apply`.

Le second champ, `m`, correspond à une table de hachage utilisant le prédicat d'égalité `eq0`. Elle enregistre les résultats calculés pour une future réutilisation si nécessaire. L'identifiant de chaque BDD est utilisé comme clef de hachage vu ses propriétés.

La fonction récursive `apply` est ensuite implémentée. Sa terminaison est montrée à l'aide du variant `b` : le BDD donné en argument décroît structurellement à chaque appel récursif. Le code, nettoyé de ses (nombreuses) préconditions et postconditions, est présenté en figure 6.

```
let rec apply (map : memoMap) (b : bdd) : bdd =
  (* Préconditions et postconditions disponibles en annexe *)
  variant { b }
  match b with
    | Bottom -> Top
    | Top -> Bottom
    | _ ->
      try
        MemoMap.find map.m b
      with MemoMap.NotFound ->
          let new_tb = apply map (true_branch b) in
          let new_fb = apply map (false_branch b) in
          let new_value = create_node map.hc (var b) new_tb new_fb
          in
          MemoMap.put map.m b new_value;
          new_value
      end
```

FIGURE 6 – Code sans préconditions ni postconditions de la fonction `apply` du module `Not`.

Comme précédemment, munie des correctes préconditions et postconditions, la validité ainsi que la terminaison de la fonction `apply` est montrée sans grande difficulté par les démonstrateurs automatiques. Le tableau 2 résume le statut des preuves réalisées par chaque démonstrateur pour prouver la spécification du module `Not`.

Proof obligations	Alt-Ergo 2.4.2	CVC5 1.0.5	Z3 4.12.3
VC for memoMap	0.01	0.02	0.02
VC for init_memo_map	0.01	0.02	0.03
VC for apply	(5.00s)	(5.00s)	1.67

TABLE 2 – Bilan des preuves automatiques pour le module `Not`.

3.5 Bilan des travaux réalisés

La bibliothèque implémentée et prouvée est bien plus complète que ce qui a été développé ici. Un rapport des preuves est disponible [11]. Les méthodes implémentées et prouvées sont :

- L'opérateur `create_node` assure la création de diagrammes réduits et ordonnés, le tout avec un partage maximal assuré par l'utilisation du *Hash Consing*.
- L'opération de négation d'un BDD est disponible et prouvée dans le module `Not`. L'utilisation de la *mémoïsation* assure une complexité au pire linéaire en la taille du diagramme.

— Le calcul du nombre de nœuds distincts d'un diagramme est possible via la fonction `size` du module `Size`. La complexité de cette opération est linéaire en la taille du diagramme.
— Le module `BinOp` fournit une implémentation pour toutes les opérations binaires. Une fonction du type `bool -> bool -> bool`, décrivant l'opération à réaliser, est juste nécessaire pour l'instanciation de ce module. L'algorithme implémenté garantit une complexité au pire linéaire selon le produit de la taille des diagrammes d'entrée.
— La fonction `is_sat` du module `Sat` permet de déterminer si un BDD est satisfiable en temps constant.
— La fonction `any_sat` du module `AnySat` renvoie une affectation des variables satisfiant un BDD sous la forme d'un tableau.
— La fonction `count_sat` du module `CountSat` renvoie le nombre d'affectations satisfiant un BDD.

On insiste sur le fait que toutes ces fonctions sont prouvées à l'aide de la plateforme *Why3*.

4 Un exemple : le Problème des N-Reines

Un exemple pour l'utilisation de cette bibliothèque a été réalisé : la (traditionnelle) résolution du problème des N-Reines. Le problème que l'on se propose de résoudre peut être posé de la manière suivante :

> Soit n un entier. En respectant les règles du jeu d'échecs, est-il possible de placer n reines sur un échiquier de $n \times n$ cases, sans qu'elles se menacent mutuellement ? Si oui, donner une solution possible ainsi que le nombre total de solutions.

On encode ce problème dans la logique propositionnelle afin de construire le BDD correspondant. On introduit les n^2 variables $V_{i,j}$ représentant la présence d'une reine à la case (i, j). Le tableau 3 décrit la reformulation du problème dans ce modèle.

Il y a au moins une reine par ligne :	$\bigwedge_{i=0}^{n} \left(\bigvee_{j=0}^{n} V_{i,j} \right)$
Il n'y a pas deux reines sur la même colonne :	$\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{n} \bigwedge_{\substack{j'=0 \\ j' \neq j}}^{n} V_{i,j} \Rightarrow \neg V_{i,j'}$
Il n'y a pas deux reines sur la même ligne :	$\bigwedge_{j=0}^{n} \bigwedge_{i=0}^{n} \bigwedge_{\substack{i'=0 \\ i' \neq i}}^{n} V_{i,j} \Rightarrow \neg V_{i',j}$
Il n'y a pas deux reines sur la même diagonale :	$\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{n} \bigwedge_{\substack{d=\max(-i,-j) \\ d \neq 0}}^{\min(n-i,n-j)} V_{i,j} \Rightarrow \neg V_{i+d,j+d}$
Il n'y a pas deux reines sur la même antidiagonale :	$\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{n} \bigwedge_{\substack{d=\max(-i,j-n+1) \\ d \neq 0}}^{\min(n-i,j+1)} V_{i,j} \Rightarrow \neg V_{i+d,j-d}$

TABLE 3 – Une reformulation possible du problème des N-Reines dans la logique propositionnelle.

Avec cette reformulation, il suffit d'instancier le module `BinOp` pour les opérations $\vee$, $\wedge$ et $\Rightarrow$. Construire le BDD associé au problème est alors immédiat. L'appel à la fonction `is_sat` sur le diagramme construit détermine si le problème possède une solution. Les fonctions `count_sat` et `any_sat` permettent respectivement de compter le nombre de solutions et d'extraire une solution du problème. On récapitule dans le tableau 4 les résultats et quelques statistiques pour plusieurs exécutions. Une solution au problème des 8-Reines est également disponible en figure 7.

n	Taille du diagramme	Nombre de solutions	Temps d'exécution (en s.)
1	1	1	0.000003
2	0	0	0.000028
3	0	0	0.000163
4	29	2	0.000801
5	167	10	0.008143
6	129	4	0.026549
7	1099	40	0.104563
8	2451	92	0.453080
9	9557	352	1.989143
10	25945	724	8.119704
11	94822	2680	29.022493
12	435170	14200	116.713563

Table 4 – Performances de la bibliothèque pour le problème des N-Reines

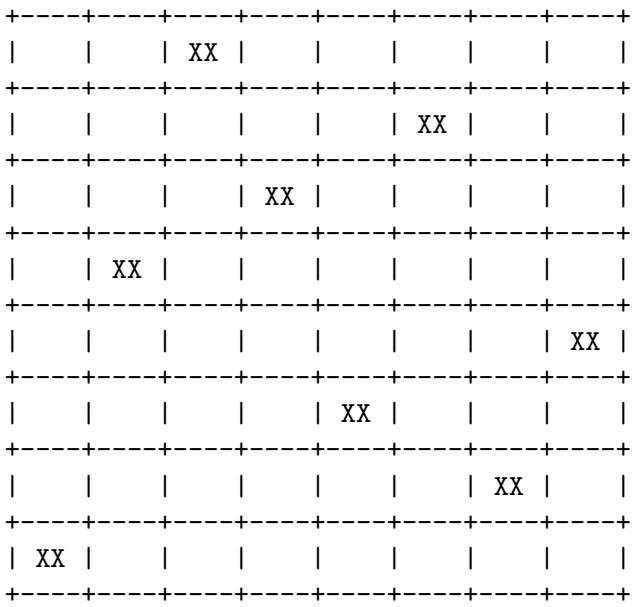

Figure 7 – Solution donnée par la bibliothèque pour $n = 8$

5 Conlusion

L'implémentation et la preuve d'une bibliothèque pour la manipulation des diagrammes de décision binaire furent de très bons exercices, tant sur le plan formel qu'informatique. La première partie de ce stage m'a permis d'obtenir une très bonne maîtrise de l'outil *Why3* pour la preuve d'algorithme.

Cette bibliothèque, bien que minimale, a été prouvée avec succès via la plateforme *Why3*. On notera cependant que l'implémentation proposée de la méthode `count_sat` est assez simpliste. Ceci est dû à la complexité des preuves nécessaires à l'implémentation d'une version optimisée. La taille du contexte devenant importante, il devient difficile pour les démonstrateurs automatiques d'aboutir. L'usage d'un outil tel que *Coq* pourrait permettre la preuve d'une version bien plus efficace, toujours via la plateforme *Why3*.

Je remercie vivement Jean-Christophe Filliâtre et Andrei Paskevich pour leur encadrement et leur aide au cours de mon stage.

Références

[1] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond et Andrei Paskevich. *Why3*. Version 1.6.0. Mars 2023. url : https://why3.lri.fr/.

[2] *Proposition de stage*. url : https://gabriel.desfrene.fr/robdd/sujet-stage.pdf.

[3] Martin Clochard. « Preuves taillées en biseau ». In : *vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*. Gourette, France, jan. 2017. url : https://inria.hal.science/hal-01404935.

[4] Microsoft Research. *Z3 Theorem Prover*. Version 4.12.2. Mai 2023. url : https://github.com/Z3Prover/z3.

[5] Clark Barrett et Cesare Tinelli. *CVC5*. Version 1.0.5. Mars 2023. url : https://cvc5.github.io/.

[6] OCamlPro. *Alt-Ergo*. Version 2.4.2. Août 2022. url : https://alt-ergo.ocamlpro.com/.

[7] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams*. Oct. 1997. url : https://www.cs.utexas.edu/~isil/cs389L/bdd.pdf.

[8] Gabriel Desfrene, Jean-Christophe Filliâtre et Andrei Paskevich. *ROBDD-in-Why3*. Juin 2023. url : https://github.com/desfreng/ROBDD-in-Why3.

[9] Jean-Christophe Filliâtre et Sylvain Conchon. « Type-safe modular hash-consing ». In : *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. ACM, 2006, p. 12-19. doi : 10.1145/1159876.1159880.

[10] Martin Clochard, Jean-Christophe Filliâtre et Andrei Paskevich. « How to Avoid Proving the Absence of Integer Overflows ». In : *Verified Software : Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*. T. 9593. Lecture Notes in Computer Science. Springer, 2015, p. 94-109. doi : 10.1007/978-3-319-29613-5_6.

[11] *Rapport des preuves effectuées*. url : https://gabriel.desfrene.fr/robdd/why3session.html.

Ce document est disponible en version numérique à l'adresse suivante :
https://gabriel.desfrene.fr/robdd/rapport.pdf

A Module Hashtbl

```
module Hashtbl
  use map.Map

  type key

  val predicate eq key key : bool

  type t 'a = abstract {
    mutable contents: map key 'a ;
    mutable domain: map key bool
  }
  invariant {
    (       (* All similar key have the same status in the table's domain *)
      forall u v : key. eq u v -> domain u = domain v
    ) /\ ( (* If two key are similar and one is in the table,
              they map to the same value *)
      forall u v : key. eq u v /\ domain u ->
          Map.([]) contents u =  Map.([]) contents v
    )
  }
  by { contents = (fun _ -> any 'a) ; domain = (fun _ -> false) }

  function is_in (h: t 'a) (k: key) : bool =
    h.domain k

  function val_of (h : t 'a) (k : key) : 'a = Map.([]) h.contents k

  val create (_u : unit) : t 'a
    ensures { forall k : key. is_in result k = false }

  val put (h: t 'a) (k: key) (v: 'a) : unit
    writes { h }
    ensures { forall k' : key. eq k k' -> is_in h k' /\ val_of h k' = v }
    ensures { forall k' : key. not eq k k' -> val_of h k' = val_of (old h) k'
                /\ is_in h k' = is_in (old h) k' }

  exception NotFound

  val find (h: t 'a) (k: key) : 'a
    ensures { is_in h k /\ val_of h k = result }
    raises  { NotFound -> not is_in h k }

  val mem (h: t 'a) (k: key) : bool
    ensures { result = is_in h k }
end
```

B Module BDD

```
  module BDD
  use int.Int
  use mach.peano.Peano
  use mach.int.Int63
  use export BDDType

  use HashConsingHashtbl as HCMap

  (* A value is mem_tbl in the table if it is in the table and it is
     linked to itself *)
  predicate mem_tbl (tbl : HCMap.t bdd) (k : bdd) =
    HCMap.is_in tbl k /\ HCMap.val_of tbl k = k

  (* A value is well-formed (ie a ROBDD) if its mem_tbl in the hash-consing
```

```
15        table or it's a leaf *)
16    predicate well_formed (tbl : HCMap.t bdd) (k : bdd) =
17      is_leaf k \/ mem_tbl tbl k
18
19    type hctable = { tbl : HCMap.t bdd; mutable next_id : Peano.t }
20
21    (* Top and Bottom are not in the Table *)
22    invariant { not HCMap.is_in tbl Top /\ not HCMap.is_in tbl Bottom }
23
24    (* Each BDD mem_tbl in table is Ordered [True Branch] *)
25    invariant { forall u : bdd. mem_tbl tbl u -> is_node (true_branch u) ->
26                  var u < var (true_branch u) }
27
28    (* Each BDD mem_tbl in table is Ordered [False Branch] *)
29    invariant { forall u : bdd. mem_tbl tbl u -> is_node (false_branch u) ->
30                  var u < var (false_branch u) }
31
32    (* No BDD mem_tbl in table has the same branch twice *)
33    invariant { forall u : bdd. mem_tbl tbl u -> true_branch u <> false_branch u }
34
35    (* Each BDD in the domain is in the same equivalence-class of eq1 to its
36       associated value *)
37    invariant { forall u : bdd. HCMap.is_in tbl u -> eq1 u (HCMap.val_of tbl u) }
38
39    (* Each BDD mem_tbl in table has a unique id *)
40    invariant { forall u v : bdd. mem_tbl tbl u /\ mem_tbl tbl v /\ u <> v ->
41                  (id u).v <> (id v).v }
42
43    (* Each BDD mem_tbl in table has a id lower than next_id *)
44    invariant { forall u : bdd. mem_tbl tbl u -> (id u).v < next_id.v }
45
46    (* Transitivity of true_branch for the BDD mem_tbl in table *)
47    invariant { forall u : bdd. mem_tbl tbl u ->
48                  well_formed tbl (true_branch u) }
49
50    (* Transitivity of false_branch for the BDD mem_tbl in table *)
51    invariant { forall u : bdd. mem_tbl tbl u ->
52                  well_formed tbl (false_branch u) }
53
54    (* Next_id begin at 2 *)
55    invariant { next_id >= 2 }
56
57    (* Each BDD mem_tbl in table have an id greater than 2 *)
58    invariant { forall u : bdd. mem_tbl tbl u -> id u >= 2 }
59
60    by { tbl = HCMap.create (); next_id = Peano.succ Peano.one }
61
62    (* Each BDD is now uniquily present in the table *)
63    lemma bdd_is_unique :
64      forall hc : hctable, u v : bdd. mem_tbl hc.tbl u /\ mem_tbl hc.tbl v ->
65          var u = var v -> true_branch u = true_branch v ->
66            false_branch u = false_branch v ->
67              u = v
68
69    (* All BDD well-formed in the 'old_hc' Hash-consing table are still
70       well-formed in the newer one *)
71    predicate still_well_formed (old_hc new_hc : hctable) =
72      forall u : bdd. well_formed old_hc.tbl u -> well_formed new_hc.tbl u
73
74    (* With the hash-consing, for well-formed BDD, eq1 is the same as the
75       logical equality *)
76    lemma eq1_match_logic :
77      forall hc : hctable, u v : bdd. well_formed hc.tbl u ->
78        well_formed hc.tbl v -> u = v <-> eq1 u v
79
80    (* With the hash-consing, for well-formed BDD, eq0 is the same as the
```

```
81        logical equality *)
82    lemma eq0_match_logic :
83      forall hc : hctable, u v : bdd. well_formed hc.tbl u ->
84        well_formed hc.tbl v -> u = v <-> eq0 u v
85
86    (* Create an empty hctable *)
87    let create_hctable () : hctable =
88      { tbl = HCMap.create () ; next_id = Peano.succ Peano.one }
89
90    (* From two bdd and a variable, we create a new BDD in the table *)
91    let create_node (hc : hctable) (v : int63) (t : bdd) (f : bdd) : bdd =
92      (* If t is a node, var t > v *)
93      requires { is_node t -> var t > v }
94
95      (* If f is a node, var f > v *)
96      requires { is_node f -> var f > v }
97
98      (* t is well-formed in the hash-consing table *)
99      requires { well_formed hc.tbl t }
100
101     (* f is well-formed in the hash-consing table *)
102     requires { well_formed hc.tbl f }
103
104     (* The old well-formed bdd are still well-formed *)
105     ensures { still_well_formed (old hc) hc }
106
107     (* All well-formed BDD in the current hash-consing table where well-formed
108        in the old hash-consing table or they are the result *)
109     ensures { forall u : bdd. u <> result ->
110                 well_formed (old hc).tbl u = well_formed hc.tbl u }
111
112     (* The result is well-formed *)
113     ensures { well_formed hc.tbl result }
114
115     (* If t <> f, var result = v *)
116     ensures { t <> f -> var result = v }
117
118     (* If t = f and is_node t, var result = var t *)
119     ensures { t = f -> is_node t -> var t = var result }
120
121     (* If result is a node, var result <= v *)
122     ensures { is_node result -> var result >= v }
123
124     (* Interpretation of a ROBDD *)
125     ensures { forall a : affectation.
126                 value a result = if a v then value a t else value a f }
127
128     (* result was either well_formed in the old hash-table and the next_id
129        hasn't change or result have as id the old next_id *)
130     ensures { well_formed (old hc).tbl result /\ hc.next_id = old hc.next_id \/
131                 id result = (old hc).next_id = hc.next_id - 1 }
132
133     if eq0 t f then
134       t
135     else
136       let bdd_candidate = N v t f Peano.zero in
137       try
138         HCMap.find hc.tbl bdd_candidate
139       with HCMap.NotFound ->
140         let new_node = N v t f hc.next_id in
141         HCMap.put hc.tbl bdd_candidate new_node;
142         hc.next_id <- Peano.succ hc.next_id;
143         new_node
144       end
145   end
```

C Module `Not`

```
module Not
  use BDD
  use int.Int
  use BDDAssociationMap as MemoMap

  (* Test if 'not_bdd' is the correct value of the 'orig_bdd' bdd by the
     not operation *)
  predicate is_not_bdd (not_bdd : bdd) (orig_bdd : bdd) =
    forall f : affectation. value f not_bdd = not value f orig_bdd

  type memoMap = { hc : hctable; m : MemoMap.t bdd }

  (* Top and Bottom are not in the memoisation table *)
  invariant { not MemoMap.is_in m Top /\ not MemoMap.is_in m Bottom }

  (* All BDD in the Memoisation Table are in the Hashconsing Table *)
  invariant { forall u : bdd. MemoMap.is_in m u -> id u < hc.next_id }

  (* All BDD in the Memoisation Table have their value in the
     Hashconsing Table *)
  invariant { forall u : bdd. MemoMap.is_in m u ->
                well_formed hc.tbl (MemoMap.val_of m u) }

  (* All values are images of the not operation of their key *)
  invariant { forall u : bdd. mem_tbl hc.tbl u -> MemoMap.is_in m u ->
                is_not_bdd (MemoMap.val_of m u) u }

  (* The variable of the keys are <= than the variable of the value *)
  invariant { forall u : bdd. mem_tbl hc.tbl u ->
                MemoMap.is_in m u ->
                  is_node (MemoMap.val_of m u) ->
                    var u <= var (MemoMap.val_of m u) }

  by { hc = create_hctable (); m = MemoMap.create () }

  let init_memo_map (hc : hctable) : memoMap =
    { hc = hc ; m = MemoMap.create () }

  (** Memoised unary operation *)
  let rec apply (map : memoMap) (b : bdd) : bdd =
    (* The input BDD must be well-formed. *)
    requires { well_formed map.hc.tbl b }

    (* The result is well-formed *)
    ensures { well_formed map.hc.tbl result }

    (* If the input is a node, then it is in the table *)
    ensures { is_node b -> MemoMap.is_in map.m b }

    (* If the input is a node, then its associated value in the table is
       the result *)
    ensures { is_node b -> MemoMap.val_of map.m b = result }

    (* If the result is a node, then the input must be a node and var result
       is >= than var b *)
    ensures { is_node result -> var b <= var result /\ is_node b }

    (* All the old well-formed bdd in map.hc are still well-formed in map.hc *)
    ensures { still_well_formed (old map.hc) map.hc }

    (* All old BDD in the table are concerved and their value too *)
    ensures { forall u : bdd. MemoMap.is_in (old map).m u ->
                MemoMap.is_in map.m u /\
                  MemoMap.val_of (old map).m u = MemoMap.val_of map.m u }
```

```
65
66      (* No changes have been made to the memo_table if the input is a leaf *)
67      ensures { is_leaf b -> forall u : bdd.
68                  MemoMap.is_in (old map).m u = MemoMap.is_in map.m u }
69
70      (* The result is the result of the not operation *)
71      ensures { is_not_bdd result b }
72
73      (*  The next_id of the hash-consing table can only increase *)
74      ensures { map.hc.next_id >= (old map).hc.next_id }
75
76      variant { b }
77      match b with
78        | Bottom -> Top
79        | Top -> Bottom
80        | _ ->
81          try
82            MemoMap.find map.m b
83          with MemoMap.NotFound ->
84              let new_tb = apply map (true_branch b) in
85              let new_fb = apply map (false_branch b) in
86              let new_value = create_node map.hc (var b) new_tb new_fb
87              in
88              MemoMap.put map.m b new_value;
89              new_value
90          end
91      end
92  end
```