Partage d'implémentation, implémentation du partage : une bibliothèque fonctorisée de diagrammes de décision binaires

Rapport de travaux de recherche par

Léo Andrès

6 mai 2019

Master 1 Informatique

Sous la direction de

Jean-Christophe Filliâtre

Résumé

Ce document est un rapport de mes travaux de recherche sur le sujet *TODO*, effectué dans le cadre de ma formation en Master 1 Informatique à l'Université Paris-Saclay, j'ai été encadré par Jean-Christophe Filliâtre.

Table des matières

1 Introduction

La logique propositionnelle est un formalisme largement utilisé en mathématiques et en informatique. Il n'existe probablement pas un seul programme informatique qui ne contienne des injonctions telles que *Si A, alors, faire B*. De même, en informatique théorique, le problème de satisfiabilité des formules propositionnelles est l'exemple typique donné de problème NP-complet. C'est en effet un problème difficile et énormément de travaux de recherches lui étant rattaché on été menés. Par exemple, au travers des SAT-solvers.

Les diagrammes de décision binaires sont une autre approche, moins répandue, quant à la façon de représenter les formules de la logique propositionnelle. Il est tout de même intéressant de noter que Donal Knuth a dit a propos des diagrammes de décision binaires qu'ils sont « one of the only really fundamental data structures that came out in the last twenty-five years. ». Profitons d'ailleurs de l'occasion pour mentionner [Knu11] dont la lecture a beaucoup aidé pour la compréhension des diagrammes de décision binaires.

On va ici introduire les diagrammes de décision binaires et leur lien avec la logique propositionnelle formellement. Puis, on présentera l'implémentation qui en a été réalisée. Pour finir, quelques applications seront détaillées. Le code de l'implémentation est disponible sur un dépôt git.

2 Les diagrammes de décision binaires

2.1 La logique propositionnelle

2.1.1 Les propositions

Une proposition est un énoncé pour lequel il fait sens de juger de la valeur de vérité. Par exemple, la phrase « OCaml est le plus beau langage au monde » est soit vraie, soit fausse. En revanche, pour la phrase « Ça va ? », il n'y a aucun sens à dire qu'elle est vraie ou fausse.

On notera $\bot$ la proposition correspondant à *faux* et $\top$ celle correspondant à *vrai*. Une proposition est donc un élément de l'ensemble $\{\top, \bot\}$, que l'on notera $\mathbb{B}$.

2.1.2 Les fonctions

Dès lors, il est possible de définir des fonctions de $\mathbb{B}$ vers $\mathbb{B}$. On a par exemple la *négation*, qui est la fonction qui à $\top$ associe $\bot$ et qui à $\bot$ associe $\top$. La négation d'une proposition P est notée $\neg$P. Par la suite, lorsque l'on écrira P sans plus de précision, P sera tenue pour vraie, et lorsque l'on écrira $\neg$P, elle sera tenue pour fausse.

De façon similaire, on peut définir des fonctions de $\mathbb{B}^2$ vers $\mathbb{B}$. Soient P et Q deux propositions, on a notamment la conjonction de P et Q, notée P $\wedge$ Q ; leur disjonction, notée P $\vee$ Q ; l'implication, notée P $\Rightarrow$ Q.

P $\wedge$ Q vaut $\top$ si et seulement si P et Q valent tous les deux $\top$. P $\vee$ Q vaut $\top$ si et seulement si au moins P ou Q vaut $\top$. P $\Rightarrow$ Q vaut $\bot$ si et seulement si P vaut $\top$ et que Q vaut $\bot$.

2.1.3 Encore ?

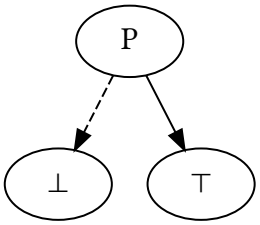
On ne détaillera pas plus la logique propositionnelle, et l'on considérera par la suite que le reste est connu. Le lecteur souhaitant en savoir plus peut se tourner vers ce cours de logique pour l'informatique.

2.2 Représentation graphique

En une phrase : les diagrammes de décision binaires permettent de représenter les fonctions de $\mathbb{B}^n$ dans $\mathbb{B}$. Les diagrammes de décision binaires peuvent être

représentés graphiquement, ce par quoi on commencera avant toute définition formelle, afin d'en facilitier la compréhension…

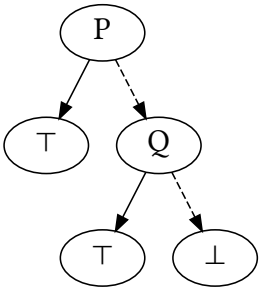
Représentons tout d'abord la formule P :



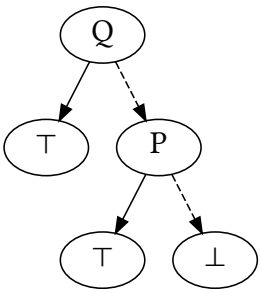
Un diagramme de décision binaire a donc un sommet « de départ », P ici. Un sommet a soit deux arcs sortants, soit aucun. S'il en a deux, alors, il contient une *variable propositionnelle* et dans ce cas, le « chemin à suivre » dépend de la valeur de vérité de la variable. Si la variable vaut $\bot$, on prend l'arc en pointillés, sinon, on prend l'autre. Si le sommet n'a aucun arc sortant, alors, il contient soit $\top$ soit $\bot$, ce qui correspond à la valeur de vérité de la formule représentée par ce diagramme dans le cas où on assigne aux variables les mêmes valeurs de vérités que celles que l'on a prises sur ce chemin.

2.2.1 Ordre

Prenons un autre exemple, voilà un diagramme représentant la proposition $P \vee Q$:



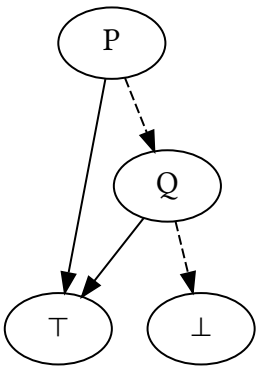
Il aurait aussi été possible de la représenter par celui-ci :



On va en fait se donner un *ordre* sur les variables et dès que l'on « aura le choix » entre deux variables, on prendra la plus petite des deux selon cet ordre. Par exemple, si l'on décide d'utiliser l'ordre lexicographique, alors, on aura le premier des deux diagrammes ci-dessus.

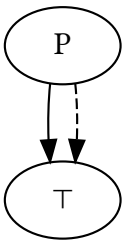
2.2.2 Non-redondance

Une seconde chose à prendre en compte est le fait que l'on va éviter toute redondance dans nos diagrammes. Si deux sommets mènent au même résultat dans tous les cas, on va les fusionner. Sur notre exemple précédent, on peut par exemple fusionner les deux sommets $\top$:

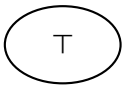


2.2.3 Simplification

Enfin, si les deux arcs sortants d'un sommet mènent à la même chose, on va tout simplement supprimer ce sommet. Par exemple, la proposition $P \vee \neg P$ ne sera pas représentée comme ceci :



Mais comme cela :



En fait, on a en quelque sorte développé la fonction, elle est « partiellement calculée ». Dans le cas d'une tautologie ou d'une contradiction, on aurait même complètement calculé la valeur de la proposition.

2.3 Représentation formelle

On utilise ici l'approche présentée dans [And97].

2.3.1 Opérateur `if-then-else`

On définit l'opérateur `if-then-else`, de $\mathbb{B}^3$ vers $\mathbb{B}$, et noté $P \to Q_1, Q_2$ ainsi :

$$P \to Q_1, Q_2 = (P \wedge Q_1) \vee (\neg P \wedge Q_2)$$

Ce qui se lit : « Si P, alors, Q_1, sinon, Q_2. ».

2.3.2 Forme normale `if-then-else`

On dit qu'une formule propositionnelle est en forme normale `if-then-else` (FNI) si elle est construite uniquement à partir de l'opérateur `if-then-else` et des constantes $\top$ et $\bot$; et de telle sorte que le premier opérande de l'opérateur `if-then-else` est toujours une variable. On recommande la lecture de [PW93] au lecteur curieux d'en apprendre plus à ce sujet.

2.3.3 Expansion de Shannon

On note $P[Q_1/Q_2]$ la proposition P dans laquelle on a substitué toutes les occurences de Q_1 par Q_2. On a alors :

$$P = Q \to P[Q/\top], P[Q/\bot]$$

C'est ce que l'on appelle l'expansion de Shannon de P par rapport à Q.

2.3.4 Passage en FNI

Pour n'importe quelle proposition, il est possible d'obtenir son équivalent en FNI. Il suffit pour cela d'appliquer l'expansion de Shannon de P par rapport à n'importe quelle variable Q de P, on aura alors $Q \rightarrow P[Q/\top], P[Q/\bot]$. Puis, on applique la même transformation à $P[Q/\top]$ et à $P[Q/\bot]$ et ce jusqu'à ce qu'il n'y ait plus de variables disponibles, on aura alors forcément soit $\top$ soit $\bot$.

Par exemple, la proposition $P \wedge Q$ peut être mise sous FNI ainsi :

$$P \wedge Q = P \rightarrow (\top \wedge Q), (\bot \wedge Q)$$

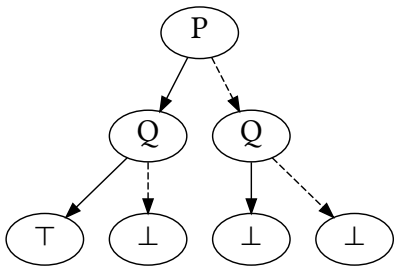
$$= P \rightarrow (Q \rightarrow (\top \wedge \top), (\top \wedge \bot)), (Q \rightarrow (\bot \wedge \top), (\bot \wedge \bot))$$

$$= P \rightarrow (Q \rightarrow \top, \bot), (Q \rightarrow \bot, \bot)$$

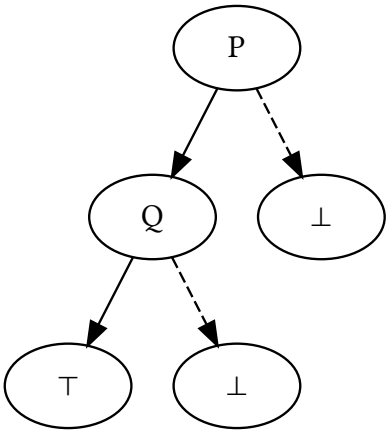
2.3.5 Passage en graphe orienté acyclique

À partir de maintenant, on ne poursuit non plus sur l'approche de [And97], mais sur celle de [Bry86].

Une proposition sous FNI peut être représentée comme un arbre binaire, sur notre exemple précédent, cela donnerait le résultat suivant :



On peut alors réduire l'arbre, c'est-à-dire que tout noeud ayant deux sous-arbres identiques sera supprimé :



Enfin, il est possible de passer à une représentation sous forme de graphe. Un arbre binaire n'étant qu'un cas particulier de graphe. Il est alors possible de réduire encore ce graphe en faisant en sorte de fusionner tous les sous-graphes isomorphes.

Le résultat final est un graphe orienté acyclique. La preuve d'absence de cycles se base sur le fait que chaque sommet est soit $\top$, soit $\bot$, soit étiqueté par une variable propositionnelle x et possède deux arcs sortants, menant à deux sommets distincts y et y' tels que $x < y$ et $x < y'$. S'il existait un cycle passant par les sommets $x_1, \ldots, x_n$, on aurait alors :

$$x_1 < x_2 < \ldots < x_n < x_1$$

Ce qui est bien évidemment contradictoire.

2.3.6 Résumé formel

Un diagramme de décision binaire est donc un graphe orienté acyclique, donc les sommets sont soit des sommets sans arcs sortants étiquetés par $\top$ ou $\bot$, soit des sommets étiquetés par des variables propositionnelles et possédants deux arcs sortants.

Un diagramme de décision binaire est dit *ordonné* si tous les chemins dans le graphes respectent un ordre donné sur l'étiquetage des sommets.

Un diagramme de décision binaire (ordonné) est dit réduit s'il respecte les deux propriétés suivantes :

unicité il n'existe pas deux sommets avec la même étiquette et les mêmes successeurs

non-redondance il n'existe pas de sommet avec les mêmes deux successeurs

2.4 Lemme de canonicité

Pour toute fonction $f : \mathbb{B}^n \to \mathbb{B}$, il existe, pour un ordre donné sur les variables propositionnelles, *un unique* diagramme de décision binaire ordonné réduit.

3 Implémentation

Toute notre implémentation s'est faite en utilisant le langage OCaml. On supposera ici que le lecteur est familier de ce langage.

3.1 Implémentation naïve

3.1.1 Types

On commence par se donner un type représentant les diagrammes de décision binaires :

```
type t =
  | True
  | False
  | Node of int * t * t
```

Un diagramme de décision binaire est soit `True` soit `False` soit un noeud étiqueté par un entier et ayant deux fils qui sont eux-mêmes des diagrammes de décision binaire.

3.1.2 Ordre sur les variables

On se donne un ordre sur les variables :

```
let get_order = function
  | True | False -> max_int
  | Node (v, _, _) -> v
```

Pour comparer deux diagrammes de décision binaire, on se contentera ensuite de récupérer leur image par `get_order` et d'utiliser la comparaison sur les entiers d'OCaml.

3.1.3 Smart constructor, fonctions auxiliaires et constantes

On introduit ensuite un *smart constructor* permettant de créer un nouveau sommet :

```
let node v l h =
  if v >= get_order l || v >= get_order h then invalid_arg
  ↪  "node";
  if l = h then l else Node (v, l, h)
```

On garantit à travers lui le respect de l'ordre et la non-redondance.

On va introduire des fonctions `view` et `fview`, inutiles pour le moment, mais que l'on modifiera ensuite, cela permettra de ne pas avoir à redéfinir d'autres fonctions.

```
let view bdd = bdd

let fview f bdd = view bdd |> f
```

On définit deux constantes correspondant à $\top$ et $\bot$:

```
let true_bdd = True
let false_bdd = False
```

3.1.4 Fonctions sur les diagrammes

On peut alors ensuite définir l'équivalent des fonctions propositionnelles sur nos diagrammes de décision binaires, par exemple la négation :

```
let rec neg =
  fview (function
  | True -> false_bdd
  | False -> true_bdd
  | Node (v, l, h) -> node var (neg l) (neg h))
```

Pour les fonctions $\mathbb{B}^2 \to \mathbb{B}$, on va commencer par définir des fonctions d'ordre supérieur `comb_comm` et `comb` qui permettent de définir un opérateur respectivement commutatif et quelconque à partir d'une fonction booléenne d'OCaml, seul le code de la version commutative est donné ici, l'autre ne servant que pour définir l'implication :

```
let of_bool = function
  | true -> true_bdd
  | false -> false_bdd

let rec comb_comm op x y =
  match view x, view y with
  | Node (v1, l1, h1), Node (v2, l2, h2) when v1 = v2 ->
      node v1 (comb_comm l1 l2) (comb_comm h1 h2)
  | Node (v1, l1, h1), Node (v2, _, _) when v1 < v2 ->
      node v1 (comb_comm l1 y) (comb_comm h1 y)
  | Node (_, _, _), Node (v2, l2, h2) ->
      node v2 (comb_comm x l2) (comb_comm x h2)
  | True, Node (v, l, h) | Node (v, l, h), True ->
      node v (comb_comm l true_bdd) (comb_comm h true_bdd)
  | False, Node (v, l, h) | Node (v, l, h), False ->
      node v (comb_comm l false_bdd) (comb_comm h false_bdd)
  | False, False -> of_bool (op false false)
  | False, True | True, False -> of_bool (op true false)
  | True, True -> of_bool (op true true)
```

On peut alors facilement définir plusieurs des opérateurs de la logique propositionnelle :

```
let conj = comb_comm (fun x y -> x && y)
let disj = comb_comm (fun x y -> x || y)
let eq   = comb_comm (fun x y -> x = y)
```

On a donc de quoi construire des diagrammes ordonnés et non-redondants. Il reste alors à implémenter la propriété d'unicité.

3.2 Implémentation du partage physique

La démarche expliquée ici repose sur celle présentée dans [CF06]. Le concept principal est celui du *hashconsing*, ou partage maximal.

Elle est surtout utile lorsqu'on a affaire à des structures de données récursives. Le but est d'être capable de ne jamais recréer deux fois la même valeur

en mémoire, tout en préservant la sûreté des types d'OCaml et en utilisant une approche modulaire.

L'idée principale est la suivante. Lorsque l'on veut créer une valeur, on regarde si on a déjà une valeur équivalente en mémoire et si oui, on réutilise cette dernière.

3.2.1 Modèle d'exécution d'OCaml

En OCaml, les valeurs sont soit des entiers, soit des pointeurs vers des entiers, c'est par exemple aussi le cas en Java. Les entiers permettent en effet d'encoder divers types. Ainsi, contrairement à ce qui peut se faire en C, une structure ou une valeur d'un type récursif ne sont jamais copiés entièrement lors de l'appel d'une fonction, on ne passe qu'un pointeur vers notre valeur, ou bien la valeur elle-même si l'ensemble des valeurs qu'elle peut prendre peut être encodé sur un seul entier.

Pour notre type de diagramme de décision binaire, on peut par exemple imaginer que la valeur `False` est un entier valant 0, `True` un entier valant 1 et `Node (v, l, h)` un pointeur vers une structure contenant successivement un pointeur vers les trois valeurs v, l et h.

Pour distinguer si une valeur est un pointeur ou non, OCaml utilise une technique quelque peu particulière. En fait, les valeurs ne sont pas codées sur 32 ou 64 bits, mais sur 31 ou 63, le bit restant servant à distinguer les pointeurs.

3.2.2 Hashconsing basique

On commence par définir une fonction `hc` qui va regarder si on a déjà construit une valeur. Si oui, on renverra donc cette valeur déjà construite. Sinon, on l'ajoute à la table et l'on renvoie cette valeur. L'égalité utilisée est celle par défaut, c'est-à-dire l'égalité structurelle d'OCaml, il en va de même pour la fonction de hachage, une fonction générique fournie par OCaml est utilisée.

```
let hc =
  let tbl = Hashtbl.create 256 in
  fun x ->
    try Hashtbl.find tbl x
    with Not_found -> Hashtbl.add tbl x x; x
```

Ainsi, on peut redéfinir les fonctions définies précedemment afin de garantir que toutes nos valeurs passent par notre fonction `hc` :

```
let true_bdd = hc True
let false_bdd = hc False

let node v l h =
  if v >= get_order l || v >= get_order h then invalid_arg
  ↪  "node";
  if l = h then l else hc (Node (v, l, h))
```

Ce faisant, on peut garantir que pour deux valeurs x et y auxquelles on a appliqué le hashconsing, on a :

$$x = y \Leftrightarrow x == y$$

On pourrait alors maintenant remplacer l'égalité structurelle par l'égalité physique dans notre fonction `hc`, le temps de comparaison ne dépendrait alors plus de la taille de notre valeur - c'est un type récursif - mais serait constant.

OCaml permet de faire cela simplement, il suffit d'instancier un module en utilisant le foncteur de table de hachages fourni par OCaml :

```
module H = Hashtbl.Make(struct
  type t = t (* (1) *)
  let equal x y = match x, y with
  | True, True | False, False -> true
  | Node (v1, l1, h1), Node (v2, l2, h2) ->
      v1 == v2 && l1 == l2 && h1 == h2
  | _ -> false
  let hash = Hashtbl.hash
)
```

La ligne avec le commentaire `(1)` n'est pas valide en OCaml, il faudrait simplement utiliser un autre nom pour le type, seulement, pour des raisons de clarté, on garde le même ici, le type `t` du côté droit faisant référence à celui définit précédemment. On redéfinit ensuite notre fonction `hc` ainsi :

```
let hc =
  let tbl = H.create 256 in
  fun x ->
    try H.find tbl x
    with Not_found -> H.add tbl x x; x
```

3.2.3 Surêté du typage

Si l'on travaille sur un type `t`, on veut pouvoir parler du type `t` sur lequel on a appliqué le *hashconsing* comme étant un type différent. Une des raisons à cela est de s'assurer que le typechecker d'OCaml ne nous laisse pas mélanger des valeurs hashconsées avec d'autres qui ne le sont pas, ce qui signifierait qu'on aurait oublié d'appliquer le hashconsing à un endroit.

On va commencer par définir un type polymorphe `'a hash_consed` :

```
type 'a hash_consed = {
  node: 'a;
  tag: int;
}
```

Ainsi, une valeur v de type `t` après *hashconsing* sera simplement l'enregistrement OCaml ayant son champ `node` valant v et dotée d'un champ `tag` supplémentaire, lequel enregistrement aura le type `t hash_consed`. Le but du champ `tag` est donné ci-après.

3.2.4 Hashconsing fonctorisé

Pour pouvoir réutiliser notre implémentation du hashconsing et en faire une bibliothèque plutôt qu'un *design pattern*, on va définir un foncteur `Make`, qui à partir d'un module ayant la signature `Hashtbl.HashedType` va produire un module ayant la signature suivante :

```
module Make(H: HashedType) : sig
  type t
  val create: int -> t
  val hashcons: t -> H.t -> H.t hashconsed
end
```

La fonction `create` permet à l'utilisateur de créer une nouvelle « table de hashconsing », qui appliquée à la fonction `hashcons` produira une fonction permettant d'obtenir la version hashconsée de n'importe quelle valeur de type `H.t`.

```
module Make(H: Hashtbl.HashedType) = struct

  module E = Hashtbl.Make(H)
  type t = E.t

  let create n = E.create n
```

```
  let hashcons =
    let gen =
      let count = ref(-1) in
      fun () -> incr count; !count
    in
    fun tbl k ->
      try E.find tbl k
      with Not_found ->
        let v = {
          tag = gen();
          node = k;
        } in
        E.add tbl k v; v

end
```

On en profite pour attribuer au champs `tag` une valeur unique, qu'on utilisera plus tard. Il faut aussi noter qu'on n'utilise plus deux fois `v`, mais une seule : en tant que valeur associée à une clé. On place notre type `'a hash_consed` et notre foncteur `Make` dans un module appelé `Hashcons`. On peut alors obtenir une version hashconsée de nos diagrammes :

```
module HBdd = Hashcons.Make(
  type t =
    | True
    | False
    | Node of int * t * t
  let equal = ... (* même définition que précédemment *)
  let hash = function
    | False -> 0
    | True -> 1
    | Node (v, l, r) ->
      19 * (19 * v + l.tag) + r.tag + 2

)
```

On a utilisé le champ `tag` pour fournir une fonction de hachage en temps constant ! Désormais, le hashconsing d'un diagramme se fait en temps constant. On peut alors redéfinir nos fonctions `view` et `hc` de la manière suivante :

```
let view bdd = bdd.node
let hc = Hbdd.hashcons (Hbdd.create 256)
```

Toutes nos anciennes fonctions sur les diagrammes restent alors correctes !

3.2.5 Hashconsing avec éphéméron

En OCaml, la mémoire n'est pas gérée par le programmeur, mais par le *garbage collector*. L'un des rôles du *garbage collector* est de désallouer la mémoire lorsqu'elle n'est plus utilisée. Pour cela, l'une des techniques les plus simples [1] il peut par exemple compter le nombre de pointeurs vers une valeur afin de savoir si celle-ci est toujours utilisée. Cela a l'avantage d'éviter de faire de nombreuses erreurs : fuites mémoires, double libération de pointeurs et autres réjouissances. Cela se fait au prix d'une baisse des performances.

Il y a aussi un second inconvénient au fait de passer par un *garbage collector*. Si l'on crée une table de hachage dont les clés sont des pointeurs pour par exemple implémenter la mémoïsation d'une fonction, le fait d'avoir un pointeur dans notre table empêchera le garbage collector de libérer cette valeur.

Notre implémentation du hashconsing présente aussi ce problème ! Si un diagramme peut être libéré, il ne le sera pas parce qu'il est présent dans la table de hashconsing…

Pour répondre à ce problème, une structure de donnée a été présentée dans [Hay97] : les éphémérons. Cette structure est plutôt générale. Ainsi, le cas particulier d'un éphéméron à une seule clé est en fait une *weak hash-table*. Il s'agit d'une table de hachage dont les clés sont des pointeurs faibles. Un pointeur faible est un pointeur traité de manière spéciale par le garbage collector : si une valeur n'est accessible que par des pointeurs faibles, alors, le garbage collector pourra libérer la mémoire occupée par cette valeur. L'entrée correspondante dans la table sera alors supprimée.

Les éphémérons ont été introduits dans OCaml 4.03. Comme les tables de hachages, il en existe une interface fonctorisée, et il existe même un foncteur pour le cas particulier des *weak hash-table*. On peut très simplement redéfinir notre foncteur `Hashcons` pour utiliser un éphéméron plutôt qu'une table de hachage :

```
module Make(H: Hashtbl.HashedType) = struct

  module E = Ephemeron.K1.Make(H)

  type key = H.t
  type data = key hash_consed
  type t = data E.t

  let create n = E.create n
```

1. Qui n'est pas celle employée par le garbage collector d'OCaml.

```
  let hashcons =
    let gen =
      let count = ref(-1) in
      fun () -> incr count; !count
    in
    fun tbl k ->
      try E.find tbl k
      with Not_found ->
        let v = {
          tag = gen();
          node = k;
        } in
        E.add tbl k v; v

end
```

Il nous a suffit de changer quatre lignes pour y parvenir. L'implémentation présentée dans [CF06] datant d'avant l'introduction des éphémérons dans OCaml, une grande quantité de code y est nécessaire pour arriver au même résultat.

3.3 Mémoïsation

On a gagné de la place en mémoire, en utilisant le partage physique, mais, on peut en fait faire *beaucoup* mieux, en mémoïsant nos opérations sur les diagrammes.

Le concept de la mémoïsation est plutôt simple : avant de calculer le résultat de l'application d'une fonction à des arguments, on regarde si on a déjà appliqué la fonction à ces mêmes arguments, si oui, on utilise ces arguments, si non, on calcule le résultat et on l'ajoute à une table de hachage avec les arguments comme clé.

Il est en fait même possible de créer une fonction permettant de mémoïser n'importe quelle fonction, ce qui nous évite d'implémenter la mémoïsation pour chaque fonction.

On va donc créer une petite bibliothèque de mémoïsation possédant un foncteur :

```
module type S = sig
  type t
  val memo: ((t -> 'a) -> t -> 'a) -> t -> 'a
  val memo2: ((t -> t -> 'a) -> t -> t -> 'a) -> t -> t -> 'a
end
```

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

module Make (H: Hashtbl.HashedType) = struct
  type t = H.t

  module Hash = Hashtbl.Make(H)
  let memo ff =
    let tbl = Hash.create 512 in
    let rec f k =
      try Hash.find tbl k
      with Not_found ->
        let v = ff f k in
        Hash.add tbl k v;
        v
    in
    f

  module Hash2 = ...
  let memo2 ff = ...
end
```

En fait, cela permet à l'utilisateur de mémoïser n'importe quelle fonction mais en plus, en lui permettant de fournir lui-même les fonctions d'égalité et de hachage à utiliser. On ne donne ici que le code pour la version à un argument, le reste étant similaire et disponible dans le code complet. On place ce code dans un module nommé `MemoFunct`.

On peut alors l'instancier pour nos diagrammes :

```
module Memo = MemoFunct.Make(struct
  type t = t hash_consed
  let equal = (==)
  let hash b = b.tag
end)
```

Ce qui nous permet alors de mémoïser toutes nos fonctions précédentes, par exemple pour la fonction `neg` :

```
let neg = Memo.memo (fun neg ->
  fview (function
  | True -> false_bdd
  | False -> true_bdd
  | Node (v, l, h) ->
      node v (neg l) (neg h)))
```

Le corps de la fonction reste identique, seul le début change légèrement.

Le point intéressant est que la mémoïsation se fait elle aussi en temps constant : on utilise l'égalité physique pour comparer deux valeurs et on se contente de récupérer le champ `tag` pour hacher une valeur.

3.4 Fonctorisation des implémentations précédentes

Cette section mais surtout, le code associé, fait un usage intensif des foncteurs d'OCaml et de son système de modules. Le lecteur qui n'est pas familier avec ce système peut se reporter vers [Ler00] qui en fait une présentation détaillée. Il est important de noter que ce système de module est extrêmement puissant et permet beaucoup de choses. Il est d'ailleurs, en théorie, adaptable à n'importe quel langage pour lequel on dispose d'un *typechecker*.

Afin de pouvoir mesurer l'impact sur l'usage mémoire et le temps de calcul du hashconsing et de la mémoïsation, on a créé un foncteur `Bdd.Make` paramétré par un module contenant un module de hashconsing et un module de mémoïsation.

On peut alors lui donner des « faux » modules de mémoïsation et de hashconsing, ou bien des vrais et ainsi obtenir quatre implémentations différentes à partir d'une seule.

Le code ne sera pas recopié ici, mais est disponible sur le dépôt du projet.

La chose la plus intéressante à noter a été la définition de la signature du module par lequel est paramétré notre foncteur :

```
module type S = sig

  type hidden
  type view =
    | False
    | True
    | Node of int * hidden * hidden

  module Hash: Hash.HashedType with type t = hidden
  module Mem: MemFunct.S with type t = hidden

  val hc: view -> hidden
  val view: hidden -> view
end
```

On a dû définir le type `view` comme étant partiellement opaque : il dépend du type `hidden` mais impose les variants à fournir. Il n'est pas possible de faire autrement, sans quoi, le compilateur n'accepte pas le type, pensant qu'il est cyclique.

3.5 SAT

On peut utiliser nos diagrammes pour implémenter les fonctions classiques proposées par un SAT-solver. Cela se fait en effet de façon extrêment simple et efficace une fois le diagramme construit.

3.5.1 `is_sat`

La fonction `is_sat` qui indique si la formule propositionnelle représentée par un diagramme est satisfiable se fait par exemple en temps constant :

```
let is_sat = fview (function
  | False -> false
  | _ -> true)
```

Si le diagramme est réduit à $\bot$, alors, la formule n'est pas satisfiable, sinon, soit il est réduit à $\top$ et est donc toujours satisfiable, soit il existe au moins une assignation des variables propositionnelles atomiques telle que la formule est satisfiable - sinon elle aurait été réduite à $\bot$. Ce problème, lorsqu'appliqué directement aux formules propositionnelles est NP-complet. Ici, toute la difficulté a été déportée vers la construction du diagramme.

3.5.2 `any_sat`

La fonction `any_sat` qui donne une assignation quelconque aux variables telle que la formule est vraie se fait aussi très simplement :

```
let any_sat =
  let rec aux assign = fview (function
    | False -> None
    | True -> Some assign
    | Node (v, l, h) -> (match aux assign l with
      | None -> aux ((v, true) :: assign) h
      | Some assign -> Some ((v, false) :: assign)))
  in aux []
```

On descend dans notre diagramme en prenant toujours la branche correspondant à l'assignation $\bot$ pour la variable courante. Si on a trouvé une assignation, alors, on l'utilise, sinon, on essaie avec l'autre branche, celle correspondant à $\top$. Si notre diagramme est réduit à $\bot$, alors on aura `None`, sinon, on aura forcément une assignation correcte.

3.5.3 `all_sat`

La fonction `all_sat` est identique, sauf qu'on parcourt toutes les branches et on garde l'ensemble des assignations valides.

```
let all_sat bdd =

  let add_assign v b = function
    | None -> None
    | Some assign -> Some ((v, b)::assign)
  in

  let rec aux assign = fview (function
    | False -> [None]
    | True -> [Some assign]
    | Node (v, l, h) ->
        let add_assign = add_assign v in
        let aux = aux assign in
        (List.map (add_assign false) (aux l)) @ List.map
        ↪  (add_assign true) (aux h))
  in

  List.fold_left (fun acc -> function
    | None -> acc
    | Some assign -> assign::acc
  ) [] (aux [] bdd)
```

Il faut noter que l'on ne considère que les variables qui apparaissent dans la formule.

3.5.4 `random_sat`

La fonction `random_sat` est elle aussi similaire. Elle calcule la taille des deux sous-diagrammes, et va vers l'un ou l'autre de façon aléatoire avec une probabilité proportionnelle à la taille de chacun.

3.5.5 `count_sat`

Le cas de `count_sat` est un peu plus compliqué car l'utilisateur doit saisir le nombre de variables existantes. On utilise alors le fait que les variables sont ordonnées pour calculer à chaque fois le nombre de variables qui n'apparaissent pas dans la formule, et on multiplie correctement le nombre possible de solutions en faisant un peu d'arithmétique. On a ici aussi dû utiliser de la mémoïsation : en effet un sous-diagramme peut apparaître plusieurs fois dans notre parcours.

3.6 Outils utilisés

En plus du langage OCaml, plusieurs outils ont été utilisés pour notre implémentation : le *build system* Dune, le générateur de parser Menhir, le framework de test alcotest, bisect_ppx pour la couverture du code et enfin, landmarks pour le profilage du code. Merci à leurs auteurs.

4 Applications

4.1 Un petit langage de formules propositionnelles

Puisque les diagrammes de décision binaires représentent des formules propositionnelles, il est naturel de vouloir créer un langage pour les formules propositionnelles. On se donne le type de syntaxe abstraite suivant :

```
type t =
  | True
  | False
  | Var of string
  | Neg of t
  | And of t * t
  | Or of t * t
  | Imp of t * t
  | Eq of t * t
  | BigAnd of t Seq.t
  | BigOr of t Seq.t
```

On écrit un lexer et un parser, qui permettent d'écrire des formules ressemblant à cela : `(x0 || (x2 => x3) <=> x4 || !x1 => true || false`.

On suppose que le types de nos expression est dans un module `E` et que l'on dispose d'une fonction `int_of_var`. Alors, pour créer un diagramme de décision binaire à partir d'une expression, on procède ainsi :

```
let rec of_expr = function
  | E.True -> true_bdd
  | E.False -> false_bdd
  | E.Var v -> node (int_of_var v) false_bdd true_bdd
  | E.Neg e -> neg (of_expr e)
  | E.And (e1, e2) -> conj (of_expr e1) (of_expr e2)
  | E.Or (e1, e2) -> disj (of_expr e1) (of_expr e2)
  | E.Imp (e1, e2) -> imp (of_expr e1) (of_expr e2)
  | E.Eq (e1, e2) -> eq (of_expr e1) (of_expr e2)
```

```
10  | E.BigAnd e -> Seq.fold_left (fun acc el -> conj (of_expr
    ↪  el) acc) true_bdd e
    | E.BigOr e -> Seq.fold_left (fun acc el -> disj (of_expr
    ↪  el) acc) false_bdd e
```

Le code complet contient toute une série de benchmarks permettant de comparer les 4 implémentations différentes de nos diagrammes de décision binaires.

4.2 Problème des N-reines

4.2.1 Présentation

Le problème des N-reines est le suivant. Supposons que l'on dispose d'une grille de taille $N \times N$. On veut disposer une reine sur chaque ligne, soit un total de N reines et ce, de telle sorte que chacune des reines soit à l'abri des autres selon les règles employées aux échecs. C'est-à-dire, pour une reine r donnée : il n'y a aucune autre reine sur la même ligne, aucune autre reine sur la même colonne et aucune autre reine sur l'une des diagonales passant par la case où se trouve r.

4.2.2 Implémentation et résultats

On va chercher ici à calculer l'ensemble des solutions possibles pour un N donné et les compter. L'implémentation consiste à exprimer les contraintes présentées précédemment directement en un diagramme de décision binaire et est relativement simple. Lorsque l'on demande le résultat pour $N = 8$, on obtient bien 92 solutions et ce, instantanément [2]. Lorsque l'on passe à $N = 9$, on obtient 352 en environ deux secondes et enfin, à partir de $N = 10$ on obtient 724 mais le temps de calcul dépasse les dix secondes.

4.3 Numberlink

4.3.1 Présentation du jeu

Numberlink est un jeu de puzzle japonais, popularisé récemment par une version disponible sur téléphone. On a une grille avec des nombres dans certaines cases. Chaque nombre apparaît exactement deux fois. Il faut alors relier les paires de nombres identiques sans que les chemins ne se croisent et de telle sorte qu'il y ait au moins un chemin passant par chacune des cases.

2. Aucun benchmark précis n'a été réalisé.

4.3.2 Représentation du problème

Pour résoudre ce problème, on va le représenter sous une forme plus générale, utilisant des graphes non-orientés plutôt qu'une grille. On a aussi écrit un parser et un lexer qui permettent de passer d'une représentation textuelle d'une grille donnée à notre version sous forme de graphe.

4.3.3 Difficulté

Le problème dans le cas des graphes revient au problème du matching de chemins disjoints, qui est NP-complet. En effet, la version avec une seule paire de nombres à relier revient à trouver un chemin Hamiltonien entre deux sommets du graphe, ce qui est déjà un problème NP-complet.

4.3.4 Réduction vers SAT

En reprenant l'approche donnée par Vicent Pilaud dans un projet de programmation de l'École polytechnique, on a réduit le problème vers SAT, et ce, en utilisant notre petit langage d'expressions propositionnelles présenté précedemment. En effet, contrairement au sujet de programmation évoqué ci-dessus, on utilisera non pas un SAT-solver, mais notre implémentation des diagrammes de décision binaire.

Il s'agit de générer les formules correspondant aux contraintes présentées dans le sujet, ce qui n'est pas complètement trivial, mais s'écrit de façon assez agréable lorsque l'on s'y prend correctement. Par exemple, la contrainte qui nous dit que pour tout chemin `i`, l'ensemble des sommets consécutifs sur le chemin `i` sont adjacents dans notre graphe, s'écrit ainsi :

```
let f paths positions vertices edges n =

  let vertices' = vertices |> without_phantom in

  for_all paths (fun path ->
  for_all positions (fun pos ->
  if pos < (n - 1) then begin
    for_all vertices' (fun vertex ->
    for_all vertices' (fun vertex' ->
      if are_adjacent edges vertex vertex' then E.True
      else E.Neg (E.And
        (mk vertex path pos,
         mk vertex' path (pos + 1)))))
  end else
    E.True))
```

Après avoir encodé l'ensemble de notre problème ainsi, on peut utiliser la fonction de translation vers les diagrammes de décision binaires pour construire le diagramme correspondant à une instance donnée. On pourra alors ensuite voir immédiatement s'il existe une solution par exemple, ou bien, demander une solution au hasard. On pourra aussi vérifier qu'une solution donnée en est bien une.

4.3.5 Résultats

Malheureusement, on arrive rapidement à des formules propositionnelles dont la taille est trop importante. La construction de nos diagrammes de décision binaires est alors bien trop lente pour être utilisée. En effet, contrairement à un SAT-solver, le diagramme de décision binaire construit toutes les solutions possibles. Une solution serait de modifier l'encodage vers SAT, en utilisant par exemple une autre représentation du problème, comme cela est d'ailleurs suggéré dans le sujet de Vincent Pilaud.

5 Conclusion

De la représentation des entiers dans le modèle d'exécution d'OCaml à la bonne façon de définir un type partiellement opaque pour permettre la fonctorisation, ces travaux de recherche ont été l'occasion d'étudier de très nombreux aspects du langage OCaml. Cela dans le but de modéliser un formalisme mathématique : la logique propositionnelle. Ces aller-retours permanents entre théorie et pratique ont été une chance, ils m'ont en effet permis d'apprendre énormément sur le sujet des diagrammes de décision binaires et diverses techniques telles que le hashconsing, la mémoïsation, ou même des structures de données telles que les éphémérons.

Je tiens particulièrement à remercier Jean-Christophe pour son encadrement. Les mercredi après-midis passés en sa compagnie ont été des plus enrichissants ! Il m'a dit aimer enseigner et je dois bien avouer que cela se ressent dans sa façon d'expliquer les choses. De plus, il est intarissable sur une infinité de sujets passionnants, ce qui mène souvent à de nombreuses disgressions [3], il faut alors employer la technique du *backtracking* [4] pour reprendre le fil, mais c'est toujours un plaisir. Alors Jean-Christophe, merci !

3. Pourquoi en 2006 fallait-il utiliser un nombre premier pour initialiser la taille d'une table de hachage alors qu'aujourd'hui il est préférable d'utiliser une puissance de deux ? Pour le savoir, demandez à Jean-Christophe !

4. Serait-ce là l'origine de son pseudonyme sur GitHub ?

Références

[Knu11] Donald E. Knuth. *The Art of Computer Programming, volume 4A : Combinatorial Algorithms, Part 1.* 1st. Addison-Wesley Professional, 2011 (cf. p. 2).

[And97] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams.* Lectures notes from Technical University of Denmark. http://www.cs.utexas.edu/~isil/cs389L/bdd.pdf. 1997 (cf. p. 6, 7).

[PW93] Christine Paulin-Mohring et Benjamin Werner. « Synthesis of ML Programs in the System Coq ». In : *J. Symb. Comput.* 15.5-6 (mai 1993), p. 607-640. issn : 0747-7171. doi : 10.1016/S0747-7171(06)80007-6. url : http://dx.doi.org/10.1016/S0747-7171(06)80007-6 (cf. p. 6).

[Bry86] Randal E. Bryant. « Graph-Based Algorithms for Boolean Function Manipulation ». In : *IEEE Trans. Comput.* 35.8 (août 1986), p. 677-691. issn : 0018-9340. doi : 10.1109/TC.1986.1676819. url : http://dx.doi.org/10.1109/TC.1986.1676819 (cf. p. 7).

[CF06] Sylvain Conchon et Jean-Christophe Filliâtre. « Type-Safe Modular Hash-Consing ». In : *ACM SIGPLAN Workshop on ML*. Portland, Oregon, sept. 2006. url : http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.pdf (cf. p. 11, 17).

[Hay97] Barry Hayes. « Ephemerons : A New Finalization Mechanism ». In : *SIGPLAN Not.* 32.10 (oct. 1997), p. 176-183. issn : 0362-1340. doi : 10.1145/263700.263733. url : http://doi.acm.org/10.1145/263700.263733 (cf. p. 16).

[Ler00] Xavier Leroy. « A modular module system ». In : *Journal of Functional Programming* 10.3 (2000), p. 269-303 (cf. p. 19).