UNIVERSITÉ PARIS 13

Rapport du Devoir de Programmation fonctionnelle

SOLVEUR BOOLEEN

Binôme

Mohand Lounis **BENSEKHRI** 11710457

Lounis **LAHOUAZI** 11709378

Année Universitaire : 2018 / 2019

L2 informatique

M. CODOGNET

TABLE DES MATIERES

INTRODUCTION

Le sujet de ce projet consiste à créer un solveur booléen, qui servira à trouver l'ensemble des solutions d'un système d'équations, et cela en passant par plusieurs étapes.

Nous avons au cours de ce projet, mis en œuvre toutes les connaissances que nous avons acquis en termes de programmation fonctionnelle et du langage OCAML.

Le code source du projet est édité en OCAML, qui est l'un des langages de programmation fonctionnelle qui a comme définition de base « Dans un langage fonctionnel, les fonctions sont des citoyens de première classe. »

La problématique de ce projet consiste tous d'abord à définir un type expression booléenne qui sera constitué de variables, qui auront comme valeur TRUE ou FALSE, et notamment de constructeurs, pour faire les différentes manipulations de ce type. Nous avons comme constructeurs ; le AND, OR, XOR et NOT qui auront comme définition le et, ou, xor et non logique connues que nous aurons à définir dans le code source.

Ensuite, dès que le type est défini, on passe à la façon d'écrire ces équations car le type que nous venons de déclarer est juste une expression booléenne, et pour former une équation booléenne nous avons besoin de deux expressions booléennes une de chaque côté du égale « = », pour cela nous avons choisi de définir le type équation booléenne sous forme de couplet d'expression booléenne et cela sans recourir à créer un nouveau type, juste en se basant sur le fait qu'une équation booléenne est un couplet de deux expressions booléennes où la première partie du couplet est le côté gauche de l'équation et la seconde ; le côté droit. Avec cela on réussit à avoir la définition d'une équation, mais il nous manque un dernier petit détail celui de faire un système d'équations booléennes, et pour cela nous avons suivi le principe d'une liste ; c'est-à-dire que pour créer un système d'équations il nous suffit de mettre chaque équation dans la liste qui sera le système final à résoudre après.

Pour la première partie de déclaration et de typage c'est ce que nous avons fait, pour l'heure nous passons sur la façon de résoudre ce système.

Pour commencer nous avons besoin d'extraire toutes les variables que le système contient ensuite dès que c'est fait il faut calculer le nombre de ces variables pour générer une table de vérité pour toutes les variables, c'est-à-dire qu'il faudra pour chaque variable une valeur de vérité et cela on parcourant toute la table de vérité. Avec cela on réussit à avoir ce que l'on appellera un environnement de cas possibles pour la résolution de ce système.

Enfin, grâce à cet environnement de possibilité nous testeront toutes les valeurs des variables sur le système et cela bien évidement en définissant d'abord le rôle des différents connecteurs d'une expression booléenne dès que c'est fait on teste les différentes possibilités sur le système et pour chaque environnement vérifiant le système on l'ajoute à la liste des solutions.

Voilà un petit récapitulatif de la problématique et de la façon dont nous nous sommes pris pour la résoudre.

TYPES ET FONCTION OCAML

I. Types

Au cours de ce projet nous avons utilisé défini qu'un seul type celui déjà fourni qui est le type eb.

Type « eb » :

type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb | XOR of eb * eb | NOT of eb ;;

« eb » désigne une expression booléenne qui contient une infinité de variables constantes TRUE et FALSE, ainsi que les quatre (04) connecteurs AND, OR, XOR et NOT. Ce type nous permet de déclarer des variables qui seront des expressions booléennes qui nous permet de former des équations booléennes ; « expression booléenne = expression booléenne » ; et plusieurs équations forment un système d'équations booléennes que nous résoudrons grâce au solveur booléen.

Exemple :

Pour déclarer l'expression booléenne suivante :

V_1 AND V_2.

On écrit :

let exp = (AND (V(1), (V2)) ;;

Pour désigner un équation booléenne nous avons choisi de la caractériser par un couple de deux expression booléenne où la première est l'expression qui est à gauche du égale de l'équation et la seconde partie du couple l'expression qui est à droite du égale de l'équation.

Exemple :

Pour déclarer l'équation booléenne suivante :

V_1 OR V_2 = TRUE.

On écrit :

let equa = ((OR (V(1), V(2))), TRUE) ;;

Et ainsi pour désigner un système équation booléenne nous avons qu'à mettre plusieurs équations booléennes dans une liste, donc en trouve qu'un système d'équations booléennes est une liste de couples d'expressions booléennes.

Exemple :

Pour déclarer le système d'équations booléennes suivant :

{ V_1 OR V_2 = TRUE ; V_1 XOR V_3 = V_2 ; NOT (V_1 AND (V_2 AND V_3)) = TRUE }

On écrit :

let sys_equa = [((OR (V(1), V(2))), TRUE)] @ [(XOR(V(1), V(3)), V(2))] @ [(NOT(AND(V(1), (AND(V(2), V(3))))), TRUE)] ;;

II. Fonctions :

A propos des fonctions nous avons utilisé au total dans ce projet 12 fonctions pour permettre la résolution d'un système d'équations booléennes que nous avons expliqué par question.

- Question 01 :
  « Détermination de l'ensemble des variables du système d'équations. »
- Réponse 01 :

Pour déterminer l'ensemble des variables de notre système d'équations booléennes nous avons eu besoin de quatre (04) fonctions qui sont « ensVarEB », «appartient », « supRep » et « variableSysExp ».

La façon dont nous nous sommes pris pour trouver les variables du système est la suivante : On commence par expliquer à une expression booléenne ce qui est une variable en elle c'est-à-dire en prend que les V_i , dès que nous pouvons avoir les variables d'une expression cela est facile d'en avoir celles d'une équation et par la suite celles du système mais il faut faire attention en prenant une variable qu'une fois pas plus c'est pour cela que nous avons besoin des deux fonction appartient et supRep pour enlever toutes les occurrences des variables différentes.

➢ ensVarEb :

Cette fonction permet de déterminer l'ensemble des variables d'une expression booléenne qui a comme paramètre une expression booléenne et renvoie une liste de ces variables avec répétitions.

Algo :

fun ensVarEB e =
si e est une (variable V) en l'a met dans une liste.
si e est égal à TRUE on l'ajoute pas à la liste.
si e est égal à FALSE on l'ajoute pas à la liste.
si e est égal à AND (x, y) on concatène les deux appels sur x et y.
si e est égal à OR(x, y) on concatène les deux appels sur x et y.
si e est égal à XOR(x, y) on concatène les deux appels sur x et y.
si e est égal à NOT(x) on concatène les deux appels sur x et y.

Code :

```
let rec ensVarEB e =
  match e with
    |V(x)  -> e::[]
    |TRUE  -> []
    |FALSE -> []
    |AND(x, y) -> (ensVarEB x) @ (ensVarEB y)
    |OR(x, y)  -> (ensVarEB x) @ (ensVarEB y)
    |XOR(x, y) -> (ensVarEB x) @ (ensVarEB y)
    |NOT(x)    -> (ensVarEB x)
    ;;
```

➢ appartient :

Cette fonction permet de déterminer si un élement appartient à une liste et cela pour l'utiliser dans la fonction « supRep » qui enlève les occurrences des variables. Elle prend en paramètre la liste où chercher et l'élément à chercher.

Algo :

fun appartient l v =
si la liste est vide donc renvoi false.
Sinon si la tête de l est égal à v renvoi true sinon appel la fun sur le reste de l avec v.

Code :

```
let rec appartient l v =
  match (l,v) with
```

```
|([], V(x)) ->  false
|([], _)    ->  false
|(a::b, V(x)) ->  if (a = V(x))
                then true
                else (appartient b v)
|(a::b, _)    ->  false
;;
```

- **supRep**

Cette fonction permet de supprimer un élément répété dans une liste, le but de cette fonction est d'enlever les répétions dans la liste fournie par la fonction "ensVarEB", en utilisant la fonction « appartient ». Elle prend en paramètre une liste et la met à jour en laissant que les éléments qui sont différents.

Algo :

fun supRep l =

si la liste est vide renvoi vide.

sinon si la tête de la liste n'appartient pas au reste on l'ajoute au résultat et en rappel la fun sur le reste de l, si la tête appartient appel directement sur y sans l'ajouter à la liste résultante.

Code :

```
let rec supRep l =
  match l with
  |[]   -> []
  |x::y ->  if ((appartient y x) == false)
          then [x] @ (supRep y)
          else (supRep y)
  ;;
```

- **variableSysExp**

Cette fonction permet de déterminer l'ensemble des variables d'un système d'expressions. Si le système est vide elle déclenche une exception en envoyant un message d'erreur. Elle prend en paramètre un système d'équations booléennes et renvoie la liste de ces variables sans répétition.

Algo :

fun variableSysExp l =

si l est vide déclencher l'exception

sinon on appelle la fun ensVarEB sur le couple de la tête de l et en enlève les répétition avec supRep et on l'ajoute a la liste final des variable et on applique le processus tant que le reste de l n'est pas égal à 0.

Code :

```
let rec variableSysExp l =
  match l with
  |[] -> failwith "Le systeme est vide !"
  |(x1, x2)::q -> let l1 = ensVarEB (x1) in
                let l2 = ensVarEB (x2) in
                let l3 = supRep (l1@l2) in
                if (List.length q != 0)
                then supRep(l3@variableSysExp q)
                else l3
  ;;
```

- Question 02 :

« Génération de tous les environnements possibles. (Un environnement est une liste de couples (variable, valeur), en l'occurrence de couples (variable booléenne, valeur de vérité)). »

- Réponse 02 :

Pour déterminer l'ensemble des environnement possibles pour les variables du système nous avons utilisé quatre (04) fonctions qui sont « ajoutFalseTrueT », « tableDeVerite », « couple » et « creatEnvironment ».

La façon dont nous nous sommes pris pour générer cet environnement est la suivante : Tout d'abord il faut générer une table de vérité pour les valeurs des variables du système, donc en ayant le nombre de variable on génère la table de vérité grâce aux deux fonctions « ajoutFalseTrueT » et « tableDeVerite » et dès que la table est prête qui sera sous forme de liste de liste de valeur TRUE, FALSE nous créant l'environnement des possibilités et cela en prenant la liste des variables et pour chaque liste dans la table de vérité en donne ces valeur pour la liste des variables et en créée une liste de couple variable, valeur de vérité et cela avec la fonction couple et en continue le processus tant que la table de vérité n'est pas vide, et à chaque fois en crée une liste de couple qui sera au final l'environnement des possibilité variable, valeur de vérité.

- **ajoutFalseT**

Cette fonction permet de dupliquer la tête d'une liste et ajoute pour la 1ère TRUE, la seconde FALSE, et s'auto appel sur le reste et cela jusqu'à la fin de la liste. Si la liste est vide, elle renvoie une liste de listes. Elle prend en paramètre une liste et renvoie une liste dupliquée avec pour chaque élément TRUE et pour son suivant FALSE.

Algo :

fun ajoutFalseTrueT l =

si l est vide renvoi une liste de liste vide.

Sinon si l a un seul elt duplique l'element et les met dans deux liste dif et pour chaque un ajoute TRUE , et FALSE pour l'autre.

Sinon si l contient plus d'un elt fait la meme chose que le second cas et appel la fun ajoujFalseTrueT sur le reste de l.

Code :

```
let rec ajoutFalseTrueT l =
  match l with
    |[] -> [[]]
    |x::[] -> [x @ [TRUE]] @ [x @ [FALSE]]
    |x::q  -> (x @ [TRUE])::(x @ [FALSE])::(ajoutFalseTrueT q)
    ;;
```

- **tableDeVerite**

Cette fonction permet de générer une table de vérité et cela en lui passant en paramètre le nb de variable "n", cela pour générer 2^n cas possible pour les valeurs de vérité pour les n variables.

Algo :

fun tableDeVerite n =

si n = 0 renvoi une liste vide.

sinon si n = 1 renvoi [[TRUE] ; [FALSE]].

Sinon crée une liste de liste en appelant la fun tableDeVerite sur n-1 et en lui appliquant la fun ajoutFalseTrueT à chaque fois.

Code :

```
let rec tableDeVerite n =
```

```
match n with
  |0 -> []
  |1 -> [[TRUE]; [FALSE]]
  |_ -> let aux = tableDeVerite (n-1) in ajoutFalseTrueT aux
  ;;
```

- **couple**

Cette fonction permet de mettre chaque deux éléments du même indice de deux liste dans un couple qui sera ajouté à la liste final, si l'une des deux listes est vide elle renvoie la liste vide. Elle prend en paramètre deux listes, et renvoie une liste avec les couples formées.

Algo :

fun couple l1 l2 =

si l1 ou l2 vide renvoi vide.

Sinon prendre les deux tetes de l1 et l2 et les mettre dans un couple a ajouter dans la liste finale que l'on concatène avec l'appel de couple sur les deux reste de l1 et l2.

Code :

```
let rec couple l1 l2 =
  match (l1,l2) with
    |([], l2) -> []
    |(l1, []) -> []
    |(x1::q1, x2::q2) -> (x1, x2)::(couple q1 q2)
    ;;
```

- **creatEnvironment**

Cette fonction est la fonction final en effet elle permet de générer tous les environnements possibles qui sont une liste de couple (variable, valeur de vérité). Elle prend en paramètre deux liste ; la table de vérité des variable et la liste des variables et renvoie l'environnement des possibilités.

Algo :

fun creatEnvironment l1 l2 =

si l1 vide renvoi une liste de liste vide.

Si l a un seul elt on appel couple sur l2 et elt.

Sinon si l a plus d'un elt en appel couple sur l'elt et l2 et la concatène avec l'appel de creatEnvironment sur le reste de l1 et l2.

Code :

```
let rec creatEnvironment l1 l2 =
  match l1 with
    |[] -> [[]]
    |x::[] ->  [(couple l2 x)]
    |x::q  ->  couple l2 x :: (creatEnvironment q l2)
    ;;
```

- Question 03 :

« Evaluation de la satisfaction d'une équation donnée dans un environnement donné. (On évaluera chacun des deux membres, gauche et droit, et on vérifiera l'égalité de ces deux valeurs.) »

- Réponse 03 :

Enfin pour déterminer l'ensemble des solutions de notre système d'équations booléennes nous avons rajouté quatre (04) fonctions qui sont : « convVarVal », « simplificationEquation », « testMiniEnvSurSys », et « solSysEq ».

La façon dont nous nous sommes pris pour déterminer l'ensemble des solutions du système est la suivante : Tout d'abord nous avons défini une fonction qui simplifie une expression booléenne et cela en définissant le rôle des différents connecteurs AND, OR, XOR et NOT et cela pour pourvoir à la fonction convVarVal qui teste un environnement de possibilités sur une expression booléenne pour avoir à la fin une valeur pour l'expression. Avec en généralise le processus sur le système et cela en testant pour chaque équation ensuite pour le système tous entier chaque liste dans l'environnement des possibilités et dès qu'on trouve qu'un environnement vérifie le système on l'ajoute à la liste des solutions.

➢ simplificationEquation

Cette fonction permet de tester une expression booléenne et la simplifier en une valeur de vérité ou variable et cela en définissant les différents connecteurs du type eb. Elle prend en paramètre une expression booléenne et renvoie une expression booléenne.

Algo :

```
fun simplificationEquation e =
    si e est une V(x) renvoi V(x)
    si e = TRUE renvoi TRUE
    si e = FALSE renvoi FALSE
    si c'est un AND(x, y)
            si x = TRUE appel la fun sur y
            sinon       si x = FALSE
                        renvoi FALSE
                        sinon appel la fun sur AND ( fun x, fun y).
    si c'est un OR(x, y)
            si x = false   appel fun sur y
            sinon       si x = true renvoi true
                        sinon appel la fun sur OR ( fun x, fun y).
    si c'est un XOR(x, y)
            si x = false appel fun sur y
            sinon       si x = true appel fun sur not y
                        sinon appel fun sur XOR ( fun x, fun y).
    si c'est un NOT(x, y)
            si x = false renvoi true
            sinon       si x = true renvoi false
                        sinon appel fun sur not (fun x).
```

Code :

```
let rec simplificationEquation e =
  match e with
    |V(x)  -> V(x)
    |TRUE  -> TRUE
    |FALSE -> FALSE
    |AND(x, y) -> if(x = TRUE)
            then simplificationEquation y
```

```
                else  if(x = FALSE)
                    then FALSE
                    else simplificationEquation(AND(simplificationEquation x,
    simplificationEquation y))
      |OR(x, y) -> if(x = FALSE)
                then simplificationEquation y
                else if (x = TRUE)
                    then TRUE
                    else simplificationEquation(OR(simplificationEquation x,
    simplificationEquation y))
      |XOR(x, y) -> if(x = FALSE)
                then simplificationEquation y
                else if (x = TRUE)
                    then simplificationEquation(NOT y)
                    else simplificationEquation(XOR(simplificationEquation x,
    simplificationEquation y))
      |NOT(x) -> if(x = FALSE)
              then TRUE
              else if(x = TRUE)
                  then FALSE
                  else simplificationEquation(NOT(simplificationEquation x))
      ;;
```

- **convVarVal**

Cette fonction permet de convertir chaque variable à sa valeur en la prenant d'une seule liste de l'environnement des possibilités, et donne la valeur finale de l'expression booléenne. Elle prend en paramètre une seule liste d'environnement de possibilités et une expression booléenne, et renvoie le résultat de l'expression booléenne par cette liste de valeurs.

Algo :

```
fun convVarVal env e =
    si env vide renvoi e
    si env non vide & e = V(x)      si e = la variable de env renvoi sa valeur de vérité
                                    sinon appel la fun sur le reste de env et V(x).
    si e = TRUE renvoi TRUE
    si e = FALSE renvoi FALSE
    si env non vide et e = AND(x, y) renvoi AND(fun env x, fun env y).
    si env non vide et e = OR(x, y) renvoi OR(fun env x, fun env y).
    si env non vide et e = XOR(x, y) renvoi XOR(fun env x, fun env y).
    si env non vide et e = NOT(x) renvoi NOT(fun env x).
```

Code :

```
    let rec convVarVal env e =
      match (env, e) with
        |([], x) -> x
        |((a,b)::q, V(x)) ->    if (a = V(x))
                                then b
                                else convVarVal q (V(x))
        |(env, TRUE)  -> TRUE
```

```
|(env, FALSE) -> FALSE
|((a, b)::q, AND(x, y)) -> AND(convVarVal env x, convVarVal env y)
|((a, b)::q, OR(x, y))  -> OR(convVarVal env x, convVarVal env y)
|((a, b)::q, XOR(x, y)) -> XOR(convVarVal env x, convVarVal env y)
|((a, b)::q, NOT(x))    -> NOT(convVarVal env x)
;;
```

- **testMiniEnvSurSys**

Cette fonction permet de tester la compatibilité d'une liste d'environnement de possibilité sur un système d'équations booléennes, et cela en vérifiant pour chaque équation du système. Elle prend en paramètre une seule liste d'environnement de possibilité et un système d'équations booléennes et renvoie la valeur du système par la liste.

Algo :

fun testMiniEnvSurSys env syseq =
si syeq est vide renvoi env
sinon si pour la première equation du syseq l'expression de gauche = l'expression de droite en appel la fun sur le reste du sys sinon on renvoie nul c'est-à-dire que l'environnement ne vérifie pas le sys.

Code :

```
let rec testMiniEnvSurSys env syseq =
  match syseq with
    |[] -> env
    |(x, y)::q -> if((simplificationEquation (convVarVal env x)) = (simplificationEquation
(convVarVal env y)))
                then testMiniEnvSurSys env q
                else []
    ;
```

- **solSysEq**

Cette fonction est la fonction final en effet elle permet vérifier tous les environnements généré dans toutes les équations du système. Elle prend en argument l'environnement de possibilités et le système d'équations booléennes, et renvoi la liste des solutions du systèmes.

Algo :

fun solSysEq listEnv syseq =
si listEnv est vide renvoie liste vide
sinon teste chaque liste d'env sur le sys avec testMiniEnvSurSys si c'est vide on appel sur le reste de env si non en ajoute a la liste des sol et on appel sur le reste de env.

Code :

```
let rec solSysEq listEnv syseq =
  match listEnv with
    |[] -> []
    |x::q -> if((testMiniEnvSurSys x syseq) = [])
          then solSysEq q syseq
          else (testMiniEnvSurSys x syseq)::(solSysEq q syseq)
    ;;
```

JEUX D'ESSAIS

Au niveau des jeux d'essais nous avons testé notre solveur booléen sur plusieurs systèmes d'équations booléennes qui sont toutes différentes d'une manière à englober tous les cas possibles, et cela en changeant le nombre de variables et le nombre d'équations. Et pour valider les résultats du solveur nous les avons toutes testés à quelques une à la main et surtout sur l'essai n°3 qui est une tautologie.

Remarque à propos de l'exemple du sujet :

Nous avons aussi remarqué que lors de l'exemple fourni on nous a donné deux solutions or que notre solveur booléen nous donne trois et nous avons vérifié la troisième donné à la main et cela fonctionne avec le système de l'exemple.

let exemple = [(OR(V(1), V(2)), TRUE); (XOR(V(1), V(3)), V(2)); (NOT(AND(V(1), (AND(V(2), V(3))))), TRUE)] ;;

Solutions données par le solveur :

[[(V 1, TRUE); (V 2, TRUE); (V 3, FALSE)];
[(V 1, TRUE); (V 2, FALSE); (V 3, TRUE)]; => vérifie le système !
[(V 1, FALSE); (V 2, TRUE); (V 3, TRUE)]]

Les essais :

- **Premier essaie: «** 3 variables et 3 équations ».

let essaie1 = [(OR(V(1), V(3))), TRUE; (AND(V(2), (NOT(V(3))))), TRUE; (NOT(XOR(V(1), V(2)))), TRUE] ;;
let sol_essaie1 = solveur_booleen essaie1 ;;

- Cela nous donne une seule solution qui est:

[[(V 3, FALSE); (V 1, TRUE); (V 2, TRUE)]

- **Second essaie: «** 4 variables et 4 équations. »

let essaie2 = [(OR(V(1), NOT(V(2))), TRUE); (XOR(V(3),V(4)), TRUE); (OR(V(4),V(1)), TRUE); (OR(V(2),NOT(V(1))), TRUE)] ;;
let sol_essaie2 = solveur_booleen essaie2 ;;

- Cela nous donne deux solutions qui sont:

[(OR (V 1, NOT (V 2)), TRUE); (XOR (V 3, V 4), TRUE); (OR (V 4, V 1), TRUE); (OR (V 2, NOT (V 1)), TRUE)]

- **Troisième essaie: «** 3 variables et 1 seule équation. »

let essaie3 = [(OR(V(1), (AND(V(2), V(3))))), (AND((OR(V(1), V(2))), (OR(V(1), V(3)))))] ;;
let sol_essaie3 = solveur_booleen essaie3 ;;

- Cela nous donne 8 solutions comme tous les cas sont vérifié on peut dire que l'équation est une tautologie. Ces solutions est toute la table de vérité.

[[(V 2, TRUE); (V 1, TRUE); (V 3, TRUE)];
[(V 2, TRUE); (V 1, TRUE); (V 3, FALSE)];
[(V 2, TRUE); (V 1, FALSE); (V 3, TRUE)];
[(V 2, TRUE); (V 1, FALSE); (V 3, FALSE)];

[(V 2, FALSE); (V 1, TRUE); (V 3, TRUE)];
[(V 2, FALSE); (V 1, TRUE); (V 3, FALSE)];
[(V 2, FALSE); (V 1, FALSE); (V 3, TRUE)];
[(V 2, FALSE); (V 1, FALSE); (V 3, FALSE)]]

- **Quatrième essaie: «** 3 variables et 4 équations. »

let essaie4 = [(XOR(V(1), V(2)), TRUE); (XOR(V(3),V(4)), TRUE); (AND(V(1),V(3)), TRUE); (NOT(AND(V(1),NOT(V(5)))), TRUE)] ;;
let sol_essaie4 = solveur_booleen essaie4 ;;

- ✓ Cela nous donne une seule solution qui est:

[[(V 2, FALSE); (V 4, FALSE); (V 3, TRUE); (V 1, TRUE); (V 5, TRUE)]]

- **Cinquième essaie: «** 5 variables et 4 équations. »

let essaie5 = [(AND(V(1), V(3)), TRUE); (OR(V(3),NOT(V(4))), TRUE); (XOR(V(1),V(3)), TRUE); (NOT(OR(V(1),NOT(V(5)))), TRUE)] ;;
let sol_essaie5 = solveur_booleen essaie5 ;;

- ✓ Cela nous donne aucune solution :

[]

- **Sixième essaie: «** 7 variables et 5 équations. »

let essaie6 = [(OR(V(1), V(3)), TRUE); (OR(V(3),NOT(V(4))), TRUE); (OR(V(1),V(3)), TRUE); (NOT(OR(V(1),NOT(V(5)))), TRUE); (NOT(AND(V(1), (AND(V(6), V(7))))), TRUE)] ;;
let sol_essaie6 = solveur_booleen essaie6 ;;

- ✓ Cela nous donne 8 solutions qui sont:

[[(V 4, TRUE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, TRUE); (V 7, TRUE)];
(V 4, TRUE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, TRUE); (V 7, FALSE)];
[(V 4, TRUE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, FALSE); (V 7, TRUE)];
[(V 4, TRUE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, FALSE); (V 7, FALSE)];
[(V 4, FALSE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, TRUE); (V 7, TRUE)];
[(V 4, FALSE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, TRUE); (V 7, FALSE)];
[(V 4, FALSE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, FALSE); (V 7, TRUE)];
[(V 4, FALSE); (V 3, TRUE); (V 5, TRUE); (V 1, FALSE); (V 6, FALSE); (V 7, FALSE)]]

CONCLUSION

En conclusion, nous tenons à dire que ce petit projet, ou devoir de Programmation Fonctionnelle pour implémenter un solveur booléen en langage OCAML, est à la fois intéressant et très enrichissant car il nous a permis de mettre en œuvre nos connaissances déjà acquises en programmation fonctionnelle, en particulier le langage OCAML. Il nous a notamment permis de gagner en confiance dans programmation. Mais surtout ce projet nous a encore plus mis dans un climat de travail en équipe ce qui implique l'entre aide des deux membres du binôme et ainsi faciliter le travail et surtout nous préparer pour nos vies professionnelles au sein d'une équipe de développeurs.