Rapport de projet

Programmation fonctionnelle

Nom et prénom du binôme :

➔RAHMANI Sylia
➔ KADRI TINHINANE

Objet du projet :

➔Réalisation d’un solveur booléen

Table des matières

1. INTRODUCTION :

Le présent rapport développe le projet d'implémentation en Ocaml un solveur booléen qui consiste en un algorithme permettant de calculer l'ensemble des solutions d'un système d'équations booléennes. l'objectif est de mettre en synergie et pratique les connaissances théoriques liées à la logique propositionnelle et à la programmation fonctionnelle en vue de réaliser un véritable projet pédagogique expérimental. Il permet également de renforcer la capacité à pister et corriger les erreurs dues au typage d'un langage.

Notre réflexion se déploie sur trois étapes ; il convient d'établir dans un premier temps, une présentation exhaustive du problème et de sa solution algorithmique en pseudo-code, suivie d'un listing commenté des types de fonctions Ocaml utilisées. Enfin, pour tester et interroger la pertinence et la maîtrise des fonctions retenues, des jeux d'essais nombreux et pertinents seront examinés et illustrés.

2. DIFFÉRENTES ÉTAPES DE RÉALISATION :

Le but de notre travail est de réaliser un solveur booléen pour pouvoir résoudre des équation et des système booléen.
Pour facilité l'implémentation de notre code, on a représenté les équations booléennes par des couples (x,y) telle que x représente l'équation elle même et y le résultat attendu.
Par exemples l'équation AND(V1,V2)= TRUE est représentée par le couple (AND(V1,V2),TRUE).
Le système d'équation est représenté par une liste, dont chaque maillon est une équation booléenne.
Deux types sont utilisés, un pour représenter les expression booléenne :

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

Un deuxième sert à représenter les arbres qui nous seront utiles pour la réalisation de la deuxième étape :

```
type arbre= Vide | Noeud of (eb*eb) * arbre * arbre ;;
```

Afin de mener à bien notre projet on a procéder par les étapes suivantes :

2.1. Afficher les variable de système :

Pour déterminer les variables de la première équation, on les ajoute à une liste vide (soit le nom de la liste l).
On détermine les variables de la deuxième équation si ils figurent déjà dans la liste l , on laisse la liste telle qu'elle est. Si non, on les ajoute a la liste l (ce sont des nouvelles variables qui ne figurent pas dans la première équation).
Pour réaliser cette algorithme on a implémenté deux fonctions récursives ; la première permet de déterminer l'ensemble de variable d'une équation , elle prend en argument une équation booléenne et une liste, elle renvoie une liste de variable (elle contient une seule occurrence de chaque variable présente dans le système).

```
let rec determine q l = match q with
        |V(a)-> if (appartient a l) then l else V(a)::l
        |AND(c,b)-> determine c (determine b l)
        |OR (c,b) -> determine c (determine b l)
        |XOR (c,b) -> determine c (determine b l)
        |NOT(c) -> (determine c l)
        |_ -> l;;
```

Type : val determine : eb -> eb list -> eb list = <fun>

Testes :

```
#let ll= determine ( OR( V(1),NOT(  AND(V(3), XOR(V(1),V(3)) ) )  )  ) [];;
 val ll : eb list = [V 1; V 3]
```

remarque:

« appartient » est une fonction récursive auxiliaire, elle prend en argument un élément q et une liste l, renvoi vrai si q appartient à l ,faux si non
son type est : val appartient : int -> eb list -> bool = <fun>
teste :#let e =appartient 30 [V(1);V(3);V(8)];;
val e : bool = false

La deuxième fonction permet d'avoir l'ensemble de variables de tous le système d'équation,elle prend en argument une liste d'équations et renvoie les variables qui se trouvent dans tous le système avec une seule occurrence .

```
et rec ensble_var l = match l with
        |[] ->[]
        |(a,b)::p -> determine a ( (determine b (ensble_var p)));;
```

Type : val ensble_var : (eb * eb) list -> eb list = <fun>
Teste :

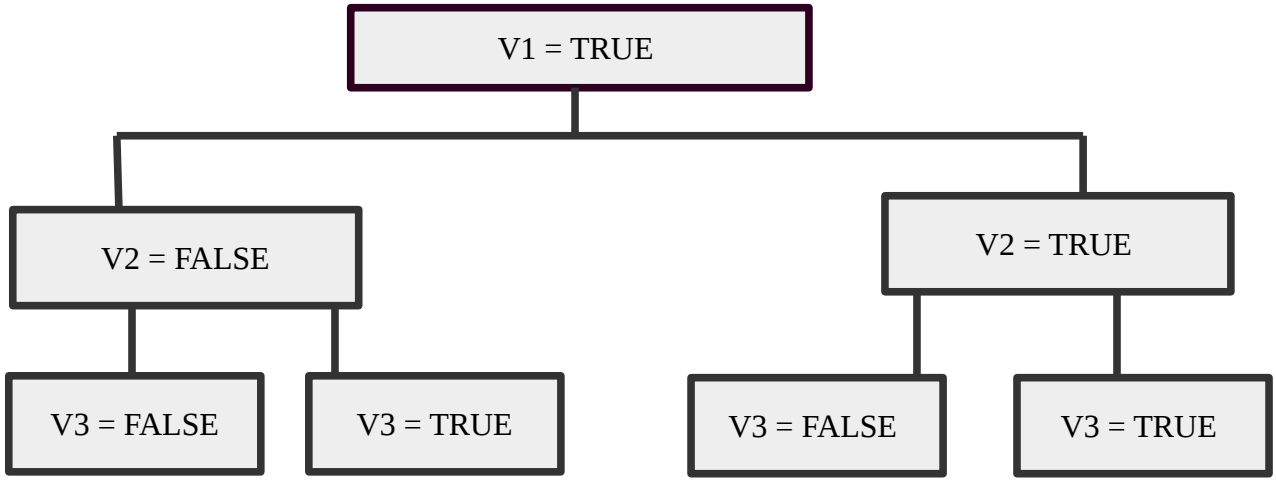
```
 #let r= ensble_var [(OR(V(1),V(2)),TRUE);(XOR(V(0),V(3)),V(1)) ; (NOT (AND(V(5),
                (AND(V(6),V(1))))),TRUE)];;
    val r : eb list = [V 2; V 0; V 3; V 5; V 6; V 1] *)
```

2.2. Afficher toutes les interprétations possibles:

Une interprétation est une liste de couples de la forme (lettre,bool).
Afficher toutes les interprétations possible signifie, afficher tous les combinaisons possibles entre l'ensemble des lettres {V1,V2,….,Vn} avec l'ensemble {TTRUE,FALSE}.
Le schéma ci-dessous récapitule un exemple d'une combinaison avec trois variables{V1,V2,V3}. Dans ce cas de figure, on a créé deux arbres ; le premier avec une racine fixant V1 à TRUE ainsi que le second fixe V1 à FALSE. Chaque racine aura deux fils d'une autre variable V2 de valeur de

vérité différente (V2=FALSE,V2=TRUE). La même logique est appliquée pour les variables V2 constituant alors des pères pour les fils V3 portant des valeurs différentes
(V3 = TRUE et V3= FALSE).
En parcourant les quatre branches de premier arbre, cela nous permet d'avoir toutes les combinaison dont la racine v1 a la valeur TRUE. On aura ainsi la liste T suivante :

T1=[[V1=TRUE, V2=FALSE,V3=FALSE] ; [V1=TRUE,V2=FALSE,V3=TRUE]
[V1=TRUE,V2=TRUE,V3=TRUE] ; [V1=TRUE,V2=TRUE,V3=FALSE]]



Pour réaliser cette arbre on a implémenter une fonction récursive « generateur_racine_true » qui génère un arbre de racine TRUE à partir d'une liste de valeurs V(n).

```
let rec generateur_racine_TRUE l a = match l with
                |[] -> a
                |V(n)::q-> generateur_racine_TRUE q (inserer_arbre_TRUE n a);;
```
Type val generateur_racine_TRUE : eb list -> arbre -> arbre = <fun>

Teste :
```
#let g= generateur_racine_TRUE ([V(1);V(2);V(3)]) Vide;;

 val g : arbre =
  Noeud ((V 1, TRUE),
    Noeud ((V 2, TRUE), Noeud ((V 3, TRUE), Vide, Vide),
          Noeud ((V 3, FALSE), Vide, Vide)),
                Noeud ((V 2, FALSE), Noeud ((V 3, TRUE), Vide, Vide),
                      Noeud ((V 3, FALSE), Vide, Vide)))
```

Remarque :
inserer_arbre_true est une fonction qui prend en paramètre un nœud et l'insère dans un arbre donné en paramètre, ce dernier a comme racine une variable de valeur de vérité vrai.

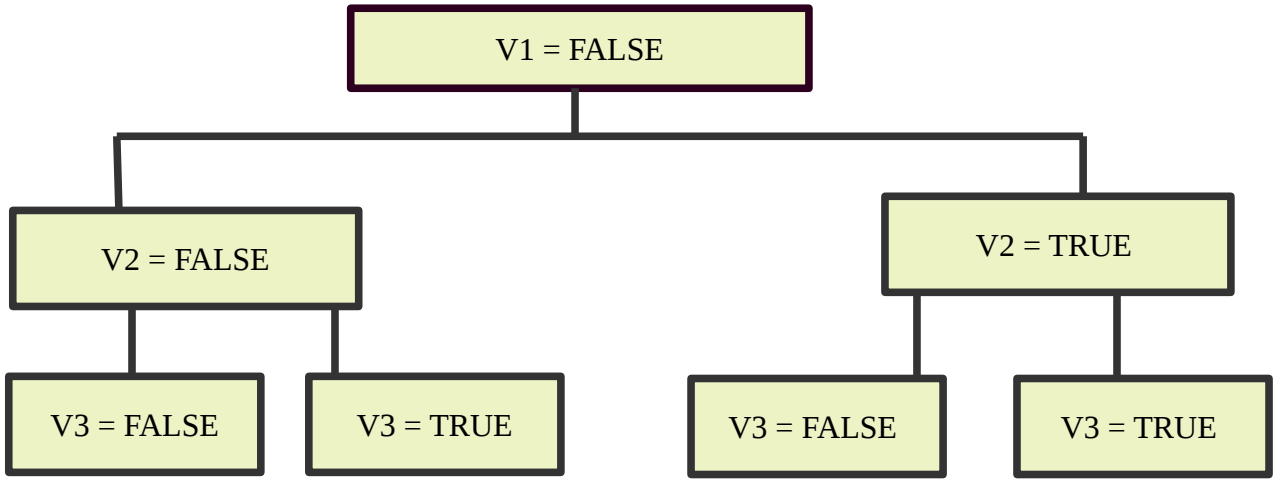
```
let rec inserer_arbre_TRUE n a = match a with
        |Vide-> Noeud((V(n),TRUE),Vide,Vide)
        |Noeud(x,fg,fd) ->if(fg=Vide) then
                (Noeud(x,Noeud((V(n),TRUE),Vide,Vide),Noeud((V(n),FALSE),Vide,Vide)) )
```

```
        else
         Noeud(x, (inserer_arbre_TRUE n fg), (inserer_arbre_TRUE n fd));;
```

Type :val inserer_arbre_TRUE : int -> arbre -> arbre = <fun>

t

On parcourant le deuxième arbre branche par branche, on aura toutes les combinaisons linéaires dont v1 a comme valeur de vérité FALSE. On aura ainsi la liste suivante :
T1= [[V1=FALSE,, V2=FALSE,V3=FALSE] ; [V1=FALSE,,V2=FALSE,V3=TRUE]
[V1=FALSE,,V2=TRUE,V3=TRUE] ; [V1=FALSE,,V2=TRUE,V3=FALSE]]



pour réaliser cette arbre on a implémenter une fonction récursive « generateur_racine_false» qui génère un arbre de racine (variable,FALSE) à partir d'une liste de valeurs V(n)

```
let rec generateur_racine_FALSE l a = match l with
                |[] -> a
                |V(n)::q-> generateur_racine_FALSE q (inserer_arbre_FALSE n a);;
```
Type : val generateur_racine_FALSE : eb list -> arbre -> arbre = <fun>

Teste :
```
#let g= generateur_racine_FALSE ([V(1);V(2);V(3)]) Vide;;
 val g : arbre =
  Noeud ((V 1, FALSE),
         Noeud ((V 2, TRUE), Noeud ((V 3, TRUE), Vide, Vide),
```

```
Noeud ((V 3, FALSE), Vide, Vide)),
        Noeud ((V 2, FALSE), Noeud ((V 3, TRUE), Vide, Vide),
                Noeud ((V 3, FALSE), Vide, Vide)))
```

Remarque :
inserer_arbre_FALSE est une fonction qui prend en argument un arbre qui a pour racine une variable de valeur de vérité FALSE et un nœud ,et insert ce dernier dans l'arbre.

```
let rec inserer_arbre_FALSE n a = match a with
        |Vide-> Noeud((V(n),FALSE),Vide,Vide)
        |Noeud(x,fg,fd) ->if(fg=Vide) then
(Noeud(x,Noeud((V(n),TRUE),Vide,Vide),Noeud((V(n),FALSE),Vide,Vide)) )  else
                        Noeud(x, (inserer_arbre_FALSE n fg), (inserer_arbre_FALSE n fd));;
```

Type : val inserer_arbre_FALSE : int -> arbre -> arbre = <fun>

on concaténant les deux listes T1 et T2 obtenu précédemment on aura une liste de toutes les interprétations possibles de toutes les variable de système.

Soit T= CONCAT T1 T2 ;

```
T=  [[V1=FALSE,, V2=FALSE,V3=FALSE] ;       [V1=FALSE,,V2=FALSE,V3=TRUE]
        [V1=FALSE,,V2=TRUE,V3=TRUE] ;   [V1=FALSE,,V2=TRUE,V3=FALSE] ;
      [V1=FALSE,, V2=FALSE,V3=FALSE] ;    [V1=FALSE,,V2=FALSE,V3=TRUE]
        [V1=FALSE,,V2=TRUE,V3=TRUE] ;   [V1=FALSE,,V2=TRUE,V3=FALSE]]
```

Pour avoir les listes T1 et T2 on a implémenter une fonction récursive auxiliaire « arbredeliste » qui prend en argument un arbre et une liste et génère tous les chemins existants de la racine jusqu'à toutes les feuilles ,et mets le tous dans une liste qui sera renvoyée à la fin .

```
let rec arbredeliste a l= match a with
        | Vide -> l
        | Noeud(e,Vide,Vide) -> e::[]
        | Noeud (e,fg,fd) -> e::(arbredeliste fg (e::l))@l@e:: (arbredeliste fd (e::l));;
```

Type :val arbredeliste : arbre -> (eb * eb) list -> (eb * eb) list = <fun>

Teste :

```
#let k=arbredeliste (generateur_racine_TRUE ([V(1);V(2);V(3)]) Vide) [];;

 val k : (eb * eb) list =
        [(V 1, TRUE); (V 2, TRUE); (V 3, TRUE); (V 1, TRUE); (V 2, TRUE);
        (V 3, FALSE); (V 1, TRUE); (V 2, FALSE); (V 3, TRUE); (V 1, TRUE); (V 2, FALSE);
         (V 3, FALSE)]
```

Pour avoir la dernier liste T, il suffit de concaténer les deux listes calculées précédemment , ce qui est l'objectif de la fonction « concat » qui prend en argument deux listes et les concatène et renvoie une liste.

```
let  rec concat l1 l2 =match l1 with

                |[]->l2

                |x1::ll1 -> x1:: concat ll1 l2 ;;
```

Type : val concat : 'a list -> 'a list -> 'a list = <fun>

Teste :

```
# let z=concat (arbredeliste (generateur_racine_TRUE ([V(1);V(2);V(3)]) Vide) []) (arbredeliste
        (generateur_racine_FALSE ([V(1);V(2);V(3)]) Vide) []);;

val z : (eb * eb) list =
        [(V 1, TRUE); (V 2, TRUE); (V 3, TRUE); (V 1, TRUE); (V 2, TRUE);
         (V 3, FALSE); (V 1, TRUE); (V 2, FALSE); (V 3, TRUE); (V 1, TRUE);
        (V 2, FALSE); (V 3, FALSE); (V 1, FALSE); (V 2, TRUE); (V 3, TRUE);
        V 1, FALSE); (V 2, TRUE); (V 3, FALSE); (V 1, FALSE); (V 2, FALSE);
        V 3, TRUE); (V 1, FALSE); (V 2, FALSE); (V 3, FALSE)]
```

La fonction « listeComplet » résume tous ce qui vient d'être fait jusqu'à présent, elle prend en argument une liste de variables et renvoie toutes les interprétations possibles.

```
let rec listeComplet l=concat (arbredeliste (generateur_racine_TRUE l Vide) []) (arbredeliste
                        (generateur_racine_FALSE l Vide) []);;
```

Type :eb list -> (eb * eb) list = <fun>

teste :

```
#let t=listeComplete [V(1);V(2);V(3)];;
val t : (eb * eb) list =
        [(V 1, TRUE); (V 2, TRUE); (V 3, TRUE); (V 1, TRUE); (V 2, TRUE);
        (V 3, FALSE); (V 1, TRUE); (V 2, FALSE); (V 3, TRUE); (V 1, TRUE);
        (V 2, FALSE); (V 3, FALSE); (V 1, FALSE); (V 2, TRUE); (V 3, TRUE);
        (V 1, FALSE); (V 2, TRUE); (V 3, FALSE); (V 1, FALSE); (V 2, FALSE);
        (V 3, TRUE); (V 1, FALSE); (V 2, FALSE); (V 3, FALSE)]
```

2.3. Ensemble d'interprétations qui satisfait le système :

Dans cette partie, nos algorithmes sont inspirés de nos connaissances en logique propositionnelle,et plus précisément des définitions suivantes :

- Une formule A est satisfaisable s'il existe au moins une interprétation I qui satisfait A.
- Un ensemble de formules Δ est satisfaisable s'il existe au moins une interprétation I telle que I satisfait Δ.
- Une formule A est valide si toute interprétation satisfait A.

- Un ensemble de formules Δ est valide si toute formule de Δ est valide

Pour calculer toutes les interprétations qui satisfaient le système d'équation booléenne on a suivie un enchaînement en cascade qui se déploie comme suit :

2.3.1. Implémentation des fonctions booléennes

Il s'agit d'implémenter les fonctions booléennes AND ,OR , XOR, NOT qui vont nous servir pour l'évaluation d'une expression booléenne .

```
let rec evaluer_NOT p =  match p with
                        |TRUE-> FALSE
                        |FALSE->TRUE;;

let rec evaluer_OR (a,b)= match (a,b) with
                        |(TRUE,_) -> TRUE
                        |(_,TRUE) ->TRUE
                        |_ -> FALSE;;

let rec evaluer_AND (a,b)= match (a,b) with
                        |(TRUE,TRUE) -> TRUE
                        |_-> FALSE

let rec evaluer_XOR (a,b) = if a=b then FALSE else TRUE;;
```

2.3.2. Interprétation d'une lettre propositionnelle

Cette partie a pour objectif de renvoyer l'interprétation d'une variable passée en argument.

```
let rec valeur_de_virite e l =
        match l with
        | []-> failwith "erreur"
        | (m,v)::r -> if (e = m) then v else valeur_de_virite e r ;;
```

Type : 'a -> ('a * 'b) list -> 'b = <fun>

Teste :

```
#let l =[(V(1),TRUE);(V(2),FALSE)];;
#let b = valeur_de_virite (V(1)) l ;;
#b;;
val b : eb = TRUE
```

2.3.3. Interprétation d'une expression booléenne

La fonction « valeur_de_verite_exp » prend en argument une expression booléenne et une liste de couple (variable,valeur de vérité) et renvoie l'interprétation de cette expression en fonction des valeurs de vérités passés en argument.

```
let rec valeur_de_verite_exp a i =
        match a with
        |TRUE->TRUE
        |FALSE->FALSE
        |V(n)-> valeur_de_virite (V(n)) i
        |NOT(x)-> evaluer_NOT(valeur_de_verite_exp x i)
        |AND(x,y)-> evaluer_AND(valeur_de_verite_exp x i,valeur_de_verite_exp y i)
        |OR(x,y)-> evaluer_OR (valeur_de_verite_exp x i ,valeur_de_verite_exp y i)
        |XOR(x,y)-> evaluer_XOR(valeur_de_verite_exp x i ,valeur_de_verite_exp y i);;
```

Type :val valeur_de_verite_exp : eb -> (eb * eb) list -> eb = <fun>

Teste :

```
#let al = NOT(AND(V(1),V(2)));;
#let d = valeur_de_verite_exp al [(V(1),TRUE);(V(2),FALSE)];;
 #d;;
val al : eb = NOT (AND (V 1, V 2))
val d : eb = TRUE
- : eb = TRUE
```

2.3.4 Déterminer si une équation est satisfaite ou pas

Comme les équations de notre système sont représentées par des couple (A,B), les valeur de vérités de v1,v2,… ,vn satisfaient l'équation A=B si et seulement si l'interprétation de A en fonction des valeurs de vérités de v1,v2,..,vn est égale à interprétation de B en fonction des valeurs de vérités de v1,v2,..,vn.
La fonction « équation_est_satisfaite » prend en argument une équation booléenne et une liste de (variable,valeur de vérités) renvoie TRUE si l'équation est satisfaite pas les valeurs de vérités des variables passer en argument, FALSE si non.

```
let rec equation_est_satisfaite (a,b) i = if((valeur_de_verite_exp a i)= (valeur_de_verite_exp b i))
                                                                    then true
                                                                else false;;
```

Type :val equation_est_satisfaite : eb * eb -> (eb * eb) list -> bool = <fun>
Teste :

```
#let e = equation_est_satisfaite (al,TRUE) [(V(1),TRUE);(V(2),FALSE)];;
#e;;
val e : bool = true
- : bool = true

#let f =equation_est_satisfaite (al,FALSE) [(V(1),TRUE);(V(2),FALSE)];;
f;;

val f : bool = false
- : bool = false
```

2.3.5 Déterminer si un système d'équation booléenne est satisfais

Une interprétation I satisfait le système S si et seulement si elle satisfait toutes les équations de S.
la fonction «systeme_est_satisfait » prend en argument un système d'équations(liste de couple (a,b)) et une interprétation I,renvoie TRUE si I satisfait S, FALSE si non.

```
let rec systeme_est_satisfait l i =
        match l with
        |[] -> FALSE
        |(a,b)::[]->if (equation_est_satisfaite (a,b) i) then TRUE else FALSE
        |(a,b)::q -> if(equation_est_satisfaite (a,b) i) then systeme_est_satisfait q i
                                                        else             FALSE;;
```

Type :
Teste :

```
#let sys = systeme_est_satisfait [(AND(V(1),V(2)),TRUE);(OR(V(1),V(1)),V(1))] [(V(1),TRUE);
(V(2),FALSE)];;
#sys;;
val sys : eb = FALSE
- : eb = FALSE

#let sy = systeme_est_satisfait [(AND(V(1),V(1)),TRUE);(OR(V(1),V(1)),V(1))] [(V(1),TRUE);
(V(2),FALSE)];;
#sy;;

val sy : eb = TRUE
- : eb = TRUE
```

```
#let ss = systeme_est_satisfait [(AND(V(1),V(1)),TRUE);(OR(V(2),V(2)),V(1))] [(V(1),TRUE);
(V(2),FALSE)];;
#ss;;
val ss : eb = FALSE
- : eb = FALSE
```

2.3.6 Déterminer l'ensemble des solutions

Dans la deuxième partie de ce projet, on a on a généré une liste de liste de toutes les interprétations possibles des différentes variables de système.

Parmi ces interprétations les seules qui constituera l'ensemble des solutions ce sont celles qui satisfaient le système, c'est à dire satisfaient toutes les équations de système.Et dés qu'une interprétation ne satisfait pas une équation on l'élimine sans même avoir besoin de vérifier si elle satisfait ou pas les autres équations restantes.

```
let rec ensemble_de_solution l i = (*l=liste d'equation i=liste de toute les combinaison possible*)
    match (l,i) with
    |(((a,b)::[]),t::[])->if ((equation_est_satisfaite (a,b) t)= TRUE) then t::[] else []
    | (d,b::t)->if((systeme_est_satisfait d b) == TRUE) then b::(ensemble_de_solution d t)
                else ensemble_de_solution d t
    | (_,_)-> [] ;;
```

Type : (eb * eb) list -> (eb * eb) list list -> (eb * eb) list list = <fun>

Teste :

```
#let c=ensemble_de_solution [(OR(V(1),V(2)),TRUE);(XOR(V(1),V(3)),V(2));
(NOT(AND(V(1),AND(V(2),V(3)))),TRUE)] [[(V 1, TRUE); (V 2, TRUE); (V 3, TRUE)];[(V 1,
TRUE); (V 2, TRUE); (V 3, FALSE)];[(V 1, TRUE); (V 2, FALSE); (V 3, TRUE)];[(V 1, TRUE);
(V 2, FALSE); (V 3, FALSE)];[(V 1, FALSE); (V 2, TRUE); (V 3, TRUE)];[(V 1, FALSE); (V 2,
TRUE); (V 3, FALSE)];[(V 1, FALSE); (V 2, FALSE); (V 3, TRUE)];[(V 1, FALSE); (V 2,
FALSE); (V 3, FALSE)]];;

val c : (eb * eb) list list =
  [[(V 1, TRUE); (V 2, TRUE); (V 3, FALSE)];
   [(V 1, TRUE); (V 2, FALSE); (V 3, TRUE)];
   [(V 1, FALSE); (V 2, TRUE); (V 3, TRUE)]]
```

a fin d'éviter de faire ce genre d'appel de fonction on a créer une fonction « projet » qui prend en argument en argument un système d'équation et renvoie l'ensemble des solution .

```
let rec projet l = ensemble_de_solution l ( divise_liste(  triT(        ensble_var(l)))  (listeComplete
                        (triT( ensble_var l))));;
```

Type : val projet : (eb * eb) list -> (eb * eb) list list = <fun>

Teste :

```
#let v= projet [(OR(V(1),V(2)),TRUE);(XOR(V(1),V(3)),V(2));
        (NOT(AND(V(1),AND(V(2),V(3)))),TRUE)];;

val v : (eb * eb) list list =
          [[(V 1, TRUE); (V 2, TRUE); (V 3, FALSE)];
          [(V 1, TRUE); (V 2, FALSE); (V 3, TRUE)];
          [(V 1, FALSE); (V 2, TRUE); (V 3, TRUE)]]
```

Remarque :

lors de la réalisation de la fonction « ensemble_de_solution », on a été obliger de découper la liste de toutes les interprétations des variables de système en sous liste dont chaque variable figure une seul fois.pour réaliser ce là plusieurs fonctions auxiliaires ont été implémenter,qui sont :

```
let rec decoupTT l p = match (l,p) with
                |([],_)->[],[]
                |(V(x)::ll,V(t)) -> let a,b =decoupTT ll p in if x<t then V(x)::a,b else a ,V(x)::b
                |_->[],[];;
```

Type :val decoupTT : eb list -> eb -> eb list * eb list = <fun>*)

```
triT  est une fonction qui trie une liste
let rec triT l=match l with
        []->[]
        |x::ll -> let a,b=decoupTT ll x in (triT a)@(x::(triT b));;
```

Type :val triT : eb list -> eb list = <fun>

Teste :

```
#let r=triT( ensble_var [(OR(V(1),V(2)),TRUE);(XOR(V(0),V(3)),V(1)) ; (NOT (AND(V(5),
        (AND(V(6),V(1))))),TRUE)]);;
val r : eb list = [V 0; V 1; V 2; V 3; V 5; V 6]
```

« enleve_premiere_liste » est une fonction qui extrait autant de nombre de premiers éléments qu'il y a de variables et les renvoie dans une liste*)

```
let rec enleve_premiere_liste l ll= match l with
                |[] ->[]
                |a::q -> match ll with
                                |o::p ->o::enleve_premiere_liste q p
                                |_->[];;
```

Type :val enleve_premiere_liste : 'a list -> 'b list -> 'b list = <fun>

Test :

```
#let f=enleve_premiere_liste  [V(1);V(2);V(3);V(8)] (listeComplete [V(1);V(2);V(3)]);;
  val f : (eb * eb) list = [(V 1, TRUE); (V 2, TRUE); (V 3, TRUE); (V 1, TRUE)]
```

supprime_premiere_liste est une fonction qui supprime autant de nombre de premiers éléments qu'il y a de variables et renvoie la nouvelle liste .

```
let rec supprime_premiere_liste ll lll= match (ll,lll) with
                                |([],p)->p
                                |(a::q,b::w) ->supprime_premiere_liste q w
                                |_ ->lll;;
```

Type :val supprime_premiere_liste : 'a list -> 'b list -> 'b list = <fun>

Test :

```
#let rec s= supprime_premiere_liste [V(1);V(2);V(3)] (listeComplete [V(1);V(2);V(3)]);;

val s : (eb * eb) list =
          [(V 1, TRUE); (V 2, TRUE); (V 3, FALSE); (V 1, TRUE); (V 2, FALSE);
            (V 3, TRUE); (V 1, TRUE); (V 2, FALSE); (V 3, FALSE); (V 1, FALSE);
           (V 2, TRUE); (V 3, TRUE); (V 1, FALSE); (V 2, TRUE); (V 3, FALSE);
           (V 1, FALSE); (V 2, FALSE); (V 3, TRUE); (V 1, FALSE); (V 2, FALSE);
            (V 3, FALSE)]
```

« divise_liste » « est une fonction qui divise la liste en sous listes d'interprétation possible et renvoie une liste de sous listes*)

```
let rec divise_liste ll lll= match lll with
                    |[]->[]
                    |_ ->(enleve_premiere_liste ll lll) :: divise_liste  ll (supprime_premiere_liste ll lll);;
```

Type : val divise_liste : 'a list -> 'b list -> 'b list list = <fun>

teste :

```
#let y= divise_liste [V(1);V(2);V(3)] (listeComplete [V(1);V(2);V(3)]);;

  val y : (eb * eb) list list =
         [[(V 1, TRUE); (V 2, TRUE); (V 3, TRUE)];
           [(V 1, TRUE); (V 2, TRUE); (V 3, FALSE)];
           [(V 1, TRUE); (V 2, FALSE); (V 3, TRUE)];
          [(V 1, TRUE); (V 2, FALSE); (V 3, FALSE)];
           [(V 1, FALSE); (V 2, TRUE); (V 3, TRUE)];
           [(V 1, FALSE); (V 2, TRUE); (V 3, FALSE)];
          [(V 1, FALSE); (V 2, FALSE); (V 3, TRUE)];
           [(V 1, FALSE); (V 2, FALSE); (V 3, FALSE)]]
```

3. conclusion

Ce projet nous a permis de consolider nos connaissance et mieux se familiariser avec le langage OCAMl, en plus d’être un projet pédagogique, il est aussi ludique et nous a donné beaucoup de liberté dans le code et dans la conception.
De points de vue de la gestion de projet, nous avons confirmé le fait que la communication est primordiale lorsque l’on travaille en groupe, un dialogue par mail ou messagerie instantanée ne remplacera jamais une entrevue en face à face. Il faut toujours réussir à motiver l’autre par les idées que l’on apporte et réfléchir avant de se lancer dans une voie.
Du point de vue de notre cahier de charge, ce projet est objectivement une réussite.Tous les objectifs principaux sont été atteints.