Année : 2021/2022
Projet Programmation fonctionnel
Professeur (responsable du module) : M. Codonnet
Binôme : Abdou Lahat Sylla, 12011836 et Yousra Mahi Moussa,12012194

Rendu projet

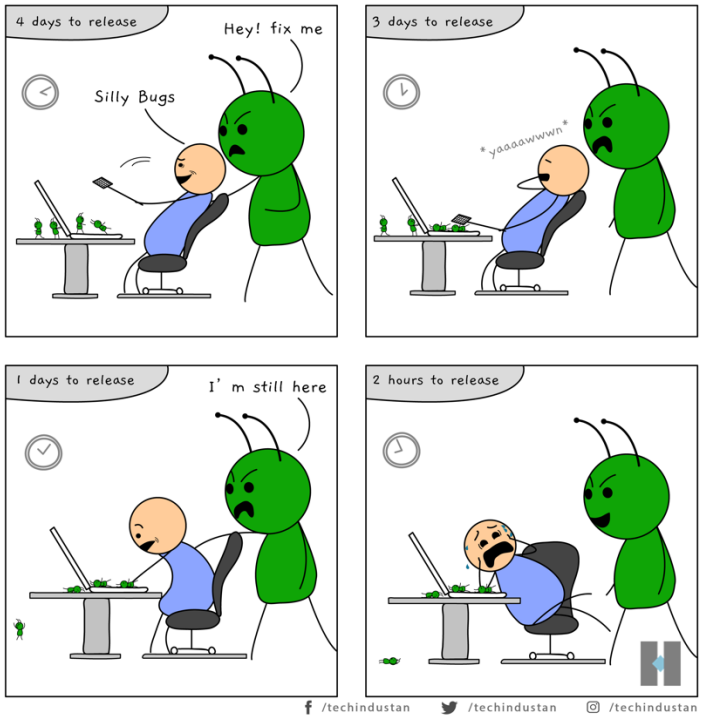
I / Rappel du sujet

Le but de ce projet est d'implémenter un solveur booléen dans le langage de programmation fonctionnel Ocaml.
Ce solveur prend en argument un système d'équations booléennes sous forme de liste et renvoie l'ensemble des solutions de ce système sous forme de liste de solutions.

II / Structure de données et les algorithmes mis en œuvre

Dans cette exercice la structure de donnée utiliser est une liste.
"Algorithme : Mot utilisé par les programmeurs quand ils ne veulent pas expliquer ce qu'ils ont fait."
L'implémentation de ce solveur s'est faite en plusieurs étapes suivants plusieurs algorithmes correctes même si leur efficacité n'étaient pas le plus grand de nos soucis.
Le premier algorithme utilisé est celui qui trouve tous les variable (V(i)) du système d'équations stockées sous forme de liste. Et ce premier algorithme a conduit à l'utilisation d'un autre algorithme qui fait la combinaison des variables V(i) dans la liste de l'algo précèdent avec les valeurs de vérités sous forme de couple (V(i), Valeur) et qui donne une nouvelle liste de liste contenant pour chaque variable V(i) sa valeur de taille 2 à la puissance la taille de la liste donnée par le premier algo.
Enfin le dernier algorithme après les deux précèdent était celui qui vérifie si chaque sous liste de la liste qui résulte du deuxième algorithme est solution du système et cette algorithme consistait à vérifier si l'expression de gauche et l'expression de droite étaient les mêmes après application pour sa V(i) sa valeur avant de le renvoyer si c'est une solution de l' équation et à la fin on a toute les solutions du deuxième algo qui vérifie le système sous forme de liste de liste.

III /Problème rencontrés et leurs solutions



Dans chacun des algorithmes cités et décrits plus haut on a rencontré pas mal de problème mais avec persistance et de l'abnégation on a su y remédier et trouver des solutions.

Le premier problème était au niveau du premier algorithme qui faisait bien le travail qu on attendait de lui mais le problème était qu'il recueillait une même variable V(i) plusieurs fois et ceci faussait le résultat en quelque sorte. Donc pour résoudre ce problème on a fait appel à un autre algorithme pour éliminer les éléments qui apparaissent deux fois dans la liste résultante du premier algorithme et ainsi tout marcha bien donc un problème de moins.

YOUUUHOUUUU !!!!

Tout se passait bien on a réussi l'implémentation du deuxième algorithme du premier coup ce qui était un exploit. Donc la fête continue ☺ .

Mais arrivé au troisième algorithme on a encore eu un problème qui était plus difficile à résoudre que le précèdent. Mais comme on dit « programmer dosn't matter bugs ».
On s'est accroché et on a cherché à trouver des solutions et finalement on a trouvé une manière de le résoudre ce problème. En effet le problème en question était à la fin de l'implémentation de cet algorithme en Ocaml on a trouvé les bonnes solutions renvoyées mais à chaque fois y avais des listes vides qui étaient rajouter dans la liste de solution et ceci faussait le résultat puisqu' une liste vide n'est pas solution du système donc ils n'avaient pas leur place dans la solution finale. Donc après plusieurs heures on a finalement trouvé un moyen d'éliminer ces liste vides dans la solution finale. Et même pour implémenter cet algorithme pour supprimer les listes vides on a eu à galérer un peu et on a fait appel à un autre algorithme qui tester l'égalité de deux listes pour que ça marche et que les listes vides soient supprimées.

IV / Types et fonctions Ocaml

On a utilisé pas mal de type dans cet exercice. Le type principal est le liste 'eb' comme « expression booléenne » contenant de variables V(i) avec les constantes booléennes « Vrai » et « FAUX » ainsi que les connecteurs qui sont au cœur de ce projet à savoir « AND, OR, XOR, NAND, NOT ».
On a aussi utilisé le type 'list' et ses primitives. En effet on a beaucoup utilisé les types :
-'eb list : pour les listes de type 'eb ;
-(eb * eb) list : pour les listes de couples de type 'eb
- a list : qui est utiliser dans ce cas avec les 'eb grâce au polymorphisme de ce type list
- a list list : qui est aussi utiliser pour les listes de liste avec le 'eb grâce au polymorphisme
- bool : qu'on a utilisé dans une des fonctions qui teste l'égalité de deux listes
-(eb * eb) list list : pour les listes de listes de couples 'eb comme par exemple le type de la solution finale renvoyé par le solveur .

En ce qui concerne les fonctions Ocaml on en a utilisé beaucoup et ça nous a beaucoup faciliter le travail car on n'avait pas besoin de redéfinir certaines fonctions il suffisait juste de faire le bon appel.
Parmis ces fonctions on a :

- List.length : qui nous renvoie la taille d'une liste donnée en argument
- List.rev : qui renverse une liste donnée en argument
- @ et ':' : la concaténation et le cons
- List.map : qui prend deux arguments et qui applique au premier chaque élément de la deuxieme
- List.concat : qui concatène deux listes et renvoie une liste contenant les éléments des deux listes

- List.combine : qui transforme un pair de listes en liste de pair

V/ Le code et les jeux d'essaies

```
(* Projet solver booleenes 2021/2022 Programmation fonctionnel  :
Binome : Abdou Lahat Sylla 12011836 et Yousra Mahi Moussa 12012194
*)

type eb = V of int | VRAI | FAUX | AND of eb * eb | OR of eb * eb | XOR
        of eb * eb | NAND of eb * eb | NOT of eb ;;

let test = [(OR(V(1),V(2)),VRAI) ; (XOR(V(1),V(3)),V(2));(NAND(V(1),AND(V(2),V(3))),VRAI)];;

let rec contient e = function
  | [] -> false
  | t::q when t = e -> true
  | _::q -> contient e q
;;

let rec concatenation l1 l2 = match l1 with
   [] -> l2
  |x::ll1 -> x::(concatenation ll1 l2);;

let elimine_doublon l =
  let rec aux acc = function
    | [] -> List.rev acc
    | t::q when contient t acc -> aux acc q
    | t::q -> aux (t::acc) q in
  aux [] l

(** 1: determination de l enssemble des variables booleenes du systeme *)

(** fonction delete_exp qui fait ressortir les expression V(i) dans une operation (OR,AND,NOT,...) et renvoie la liste des V(i) *)
let rec delete_exp e = match e with
  |FAUX -> []
  |V(i) -> [e]
  |AND(x,y) ->  delete_exp x  @ delete_exp y
  |XOR(x,y) ->  delete_exp x  @ delete_exp y
  |NAND(x,y)->  delete_exp x  @ delete_exp y
  |NOT x -> delete_exp x
  |VRAI -> []
  |OR(x,y) ->  delete_exp x  @ delete_exp y;;

```

```
(* fonction extraire_var qui extrait dans un systeme donné en argument tous les variable booleenes V(i) et renvoie la liste de ces variables *)
let rec extraire_var sys = match sys with
  |[] -> []
  |(l,s)::v  -> elimine_doublon (concatenation (delete_exp l @ delete_exp s) (extraire_var v))
;;

(*let variable = extraire_var test;;*)

(** 2: generation des environnement possibles *)

let rec remplir_resultat listVar n = match n with
    0 -> [[]]
  |_ -> match listVar with
      [] -> [[]]
    |u::v -> let x = remplir_resultat listVar (n-1) in
        List.concat(List.map (fun x -> List.map (fun listVar -> listVar@x)listVar)x)
;;
(*fonction pour faire les combinaisons des valeur de verité *)
let rec combinaison var list = match list with [] -> [] | u :: v -> List.combine var u :: (combinaison var v) ;;

(*fonction pour remplir la table de verité *)
(*let table = remplir_resultat ([[VRAI];[FAUX]]) (List.length variable);;*)

(*let environnement = combinaison variable table;;*)

(** 3)  evaluation de la satisfaction d une equation donné dans un environnement donnée *)

let rec equal l v = match (l,v) with
    ([],[]) -> true
  |([],l) -> false
  |(l,[]) -> false
  |((w::x),(y::z)) -> if List.length l != List.length v then false else if w=y then equal x z else false
;;

let rec supprimerVide v = match v with
    []::_ -> []
  |[] -> []
  | (x::y) :: li -> if equal (x::y) [] then supprimerVide  li else (x::y) :: supprimerVide li ;;

let rec valassocie var env =  match  env with
    [] -> raise(Not_found)
  |(l,s)::v -> if l = var then s else (valassocie var v )
;;

let eval_and  l s =  if l = VRAI then s else FAUX;;
let eval_or l s = if l = VRAI then VRAI else s;;
let eval_xor l s = if l != s then VRAI else FAUX;;
let eval_not l = if l = VRAI then FAUX else VRAI;;
let eval_nand l s = eval_not (eval_and l s);;

let rec app_env e env = match e with
    FAUX -> FAUX
  |VRAI -> VRAI
  |V(i) -> valassocie e  env
  |AND(x,y) ->  eval_and (app_env x env)   (app_env y env)
  |XOR(x,y) ->  eval_xor (app_env x env)   (app_env y env)
  |NAND(x,y)->  eval_nand (app_env x env)   (app_env y env)
  |NOT x -> eval_not (app_env x env)
  |OR(x,y) ->  eval_or (app_env x env )   (app_env y env)
;;
(*fonction qui verifie les solution V(i) dans un sous environnement*)
let rec solver sys env = match sys with
    [] -> env
  |(l,s)::v -> let x = app_env l env in let y = app_env s env in if x=y then solver v env else []
;;

let rec solution sys env = match env with
    [] -> []
  |e::x -> ( supprimerVide [solver sys e]) @ ( supprimerVide (solution sys x))
;;

(** fonction pour resoudre tout systeme *)
let resolver sys = solution sys (combinaison (extraire_var sys) (remplir_resultat ([[VRAI];[FAUX]]) (List.length (extraire_var sys)) ) ) ;;
```

LES TESTS :

```
resolver [(OR(V(1),V(2)),FAUX) ; (XOR(V(1),V(3)),V(2));(NAND(V(1),AND(V(2),V(3))),VRAI)] ;;
- : (eb * eb) list list = [[(V 1, FAUX); (V 2, FAUX); (V 3, FAUX)]]
(** exemple  de cas ou on a une seule solution*)
resolver [(OR(V(1),V(2)),VRAI) ; (XOR(V(1),V(3)),V(2));(NAND(V(1),AND(V(2),V(3))),VRAI)] ;;
- : (eb * eb) list list =
[[(V 1, FAUX); (V 2, VRAI); (V 3, VRAI)];
 [(V 1, VRAI); (V 2, FAUX); (V 3, VRAI)];
 [(V 1, VRAI); (V 2, VRAI); (V 3, FAUX)]]
 (* le systeme donné avec le sujet la solution est la bonne *)
resolver [(OR(V(1),V(2)),FAUX) ; (XOR(V(1),V(3)),V(2));(NAND(V(1),AND(V(2),V(3))),FAUX)] ;;
- : (eb * eb) list list = [] (*dans ce cas on a Pas de solution pour ce systeme*)
```