Année 2018/2019

Projet de programmation fonctionnelle

Noms de binôme :

TATIBOUET Kevin
AID Lamia

UNIVERSITÉ PARIS 13 | USPC Université Sorbonne Paris Cité

Sommaire

1. Cahier des charges

On désire implémenter en Ocaml un solveur booléen, c'est à dire un algorithme permettant de calculer l'ensemble des solutions d'un système d'équations booléennes.
Pour ce faire on disposait du type suivant :
type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb | XOR of eb * eb | NOT of eb

L'algorithme :

Les principales fonctionnalités de l'algorithme sont les suivantes :

1) Détermination de l'ensemble des variables du système d'équations.
2) 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)).
3) Évaluation 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.)

2. Les types

Pour ce faire nous utilisons deux types:
- type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb | XOR of eb * eb | NOT of eb ;; qui représente une expression booléenne.
- type eq = EQ of eb * eb;; un couple d'eb représentant une equation (membre gauche et membre droit, en effet on peut ne pas tenir compte du "=" car c'est forcement le membre du milieu des equations que nous utilisons).

3. Les fonctions

filtrer : Cette fonction retire tous les doublons d'une liste passée en paramètre.
C'est une fonction récursive.

Entrée : Liste l (ordonnable)

Sortie : Sous-liste de l

```
fonction filtrer l
        si l == [] alors []
        sinon si l == a::[] alors a
                sinon si l == a::r alors
                                                si a == (premier_element l)
                                                alors filtrer l
                                                sinon [a]@(filtrer l)
```

unionPartiel : Cette fonction nous sert d'union mais en vérité elle sert surtout a concatener deux listes triées en une seule liste triée.
C'est une fonction récursive.

Entrée : Liste a (ordonnable) et liste b (ordonnable)

Sortie : Union trié (sans doublons) de a et b

```
fonction unionPartiel a b
        si a == [] alors filtrer b
        si a == (d1::l1) alors
                                        si b == [] alors filtrer a
                                        si b == (d2::l2) alors
                                                                si d1 < d2 alors
d1::(unionPartiel l1 (d2::l2))
                                                                sinon si (d1 == d2) alors
(unionPartiel l1 (d2::l2))
                                                                sinon d2::(unionPartiel
(d1::l1) (l2))
```

determinExp : Permet de determiner quelles sont les variables utilisés dans une expression booléenne.

C'est une fonction récursive.

Entrée : Expression booléenne eb

Sortie : Tableau d'entier représentant les variables

fonction determinExp eb

determinEq : Permet de determiner quelles sont les variables utilisés dans une expression booléenne.

C'est une fonction récursive.

Entrée : Une equation booléenne eq

Sortie : Tableau d'entier représentant les variables

fonction determinEq eq

si eq == (a = b) alors unionPartiel (determinExp a) (determinExp b)

determin : Permet de determiner quelles sont les variables utilisés dans un systeme d'equations booléennes.

C'est une fonction récursive.

Entrée : Systeme d'equations booléennes

Sortie : Tableau d'entier représentant les variables

fonction determin s

si s == [a] alors determinEq a

si s == a::l alors unionPartiel (determinEq a) (determin l)

genAux : permet d'ajouter une variable dans un environnement sous forme de liste

C'est une fonction récursive.

Entrée : une environnement et une liste

Sortie : Tableau d'entier représentant les variables

fonction genAux e (a::l)

si l == liste_vide alors (e::a)

sinon (e::a) :: (genAux e l)

gen : permet des générer tous les environnements possibles

C'est une fonction récursive.

Entrée : une liste de variable

Sortie : l'environnement générer

Fonction gen (a::l)

si l == liste_vide alors [[a, true];[a, false]]

sinon (genAux (a, true) (gen l)) @ (genAux (a, false) (gen l))

existance : verifie l'existance d'une variable dans l'environnement

C'est une fonction récursive.

Entrée : une liste et un booléen

Sortie : un booléen

fonction existance a n

si a == liste_vide alors false

si (i, b)::[] == si i = n alors true

sinon false

sinon(I, b)::l == si i = n alors true

sinon existance l n

found : trouve la valeur associée a la variable d'environnement

Entrée : une liste et un booléen

Sortie : un booléen

Fonction a n

si (i, b)::[] alors b

sinon (I, b)::l alors si (i = n) alors b

sinon found l n

eval : permet d'évaluer chacun des deux membres, gauche et droit, et vérifie l'égalité de ces deux valeurs

C'est une fonction récursive.

Entrée : une equation booléenne et un environnement

Sortie : un booléen

Fonction eval eb e

si TRUE alors true

si FALSE alors false

si V i alors si (existance e i) alors (found e i) sinon false

si AND(a,b) alors (eval a e) et (eval b e)

si OR(a,b) alors (eval a e) ou (eval b e)

si XOR(a,b) alors si ((eval a e) == (eval b e)) alors false sinon true

si NOT a alors not (eval a e)

4. les fonctions supplémentaires

evalEq : fonction bonus qui dit si une equation est valide pour un environnement donné ; si on a "a = b" alors eval a doit être équivalent à eval b

Entrée : une equation et un environnement

Sortie : un booléen

Fonction (EQ(ab)) e

si (eval a e == eval b e) alors true sinon false

evalSyst : fonction bonus qui dit si un systeme est valide pour un environnement donné ; chaque equation composant le systeme doit être valide

Entrée : une liste et un environnement

Sortie : un booléen

Fonction evalSyst (a::l) e

si l = liste_vide alors (evalEq a e)

sinon (evalEq a e) et (evalSyst l e)

getEnvValide : fonction bonus qui filtre les environnements valides ; un environnement est valide si il est valide pour chaque equation du systeme

Entrée : un système et un liste d'environnement

Sortie : une liste d'environnement

fonction getEnvValide syst (e::l)

si l == liste_vide alors si (evalSyst syst e) alors [e] sinon []

sinon si (evalSyst syst e) alors ([e]@(getEnvValide syst l)) sinon (getEnvValide syst l)

solutions : fonction bonus qui prend un systeme et retourne les environnements valides

Entrée : un système

Sortie : un environnement valide

fonction solutions syst

si (syst == []) alors [] sinon getEnvValide syst (gen (determin syst))

5. A propos des tests

Les tests ne sont pas nombreux car ceux présent suffisent à prouver que l'implémentation que nous souhaitions faire pour notre algorithme était correcte. Ainsi certaines fonctions (gen, genAux, evalSyst et getEnvValide) sont incapables de fournir un résultat lorsque ils ont en paramètre une liste vide et cela nous paraît plutôt bien de gagner (un peu) en efficacité pour perdre les cas assez absurdes.

6. Le code version caml

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

type eq = EQ of eb * eb;;
```

```
(* cette fonction filtre les doublons d'une liste déjà triée *)

let rec filtrer l = match l with
        | [] -> []
        | a::[] -> [a]
        | a::l -> if (a==(List.hd l))
                        then (filtrer (l))
                        else ([a]@(filtrer l));;

(* union partiel car ce n'est pas vraiment union, mais plutôt une concaténation de deux
listes triées auquel on n'ajoute pas de doublon. La liste en sortie est triée *)

let rec unionPartiel a b = match a with
        | [] ->  filtrer b;
        | (d1::l1) -> match b with
                | [] -> filtrer (d1::l1)
                | (d2::l2) -> if (d1 < d2) then (d1)::(unionPartiel l1 (d2::l2))
                        else if (d1 == d2) then (unionPartiel l1 (d2::l2))
                        else d2::(unionPartiel (d1::l1) (l2));;

(* Determiner les variables d'une expression *)

let rec determinExp eb = match eb with
    | V i -> [(i)]
        | TRUE -> []
    | FALSE -> []
    | AND(a,b) -> unionPartiel (determinExp a) (determinExp b)
    | OR(a,b) -> unionPartiel (determinExp a) (determinExp b)
    | XOR(a,b) -> unionPartiel (determinExp a) (determinExp b)
    | NOT e -> determinExp e;;

(* Determiner les variables d'une equation *)
```

```
let rec determinEq eq = match eq with
        | EQ(eb1,eb2) -> unionPartiel (determinExp eb1) (determinExp eb2);;

(* Première fonction demandée dans le sujet *)

let rec determin (a::l) = match l with
        | [] -> determinEq a
        | l -> unionPartiel (determinEq a) (determin l);;

(* cette fonction ajoute une variable dans un environnement (qui est sous forme de
liste )*)

let rec genAux e (a::l) = match l with
        | [] -> [e::a]
        | l -> (e :: a) :: (genAux e l);;

(* deuxième fonction demandée dans le sujet*)

let rec gen (a::l) = match l with
        | [] -> [[a, true];[a, false]]
        | l -> (genAux (a, true) (gen l)) @ (genAux (a, false) (gen l));;

(* verifie l'existance d'une variable dans l'environnement *)

let rec existance a n = match a with
    | [] -> false
    | (i, b)::[] -> if i = n then true
                    else false
    | (i, b)::l -> if i = n then true
                    else existance l n;;

(* trouve la valeur associée a la variable dans l'environnement *)

let rec found a n = match a with
```

```
    |(i, b)::[] -> b
    |(i, b)::l -> if(i = n) then b
                  else found l n;;

(* troisième fonction demandée*)

let rec eval eb e = match eb with
        | TRUE -> true
        | FALSE -> false
        | V i -> if (existance e i) then (found e i)
                        else false;
        | AND(a,b) -> (eval a e)&&(eval b e)
        | OR(a,b) -> (eval a e)||(eval b e)
        | XOR(a,b) -> if((eval a e) == (eval b e)) then false else true
        | NOT a -> not (eval a e);;

(* ____________________________________________________ *)

(* fonction bonus qui dit si une equation est valide pour un environnement donné ; si
on a "a = b" alors eval a doit être équivalent à eval b *)

let evalEq (EQ(a,b)) e = if(eval a e == eval b e) then true else false;;

(* fonction bonus qui dit si un systeme est valide pour un environnement donné ;
chaque equation composant le systeme doit être valide *)

let rec evalSyst (a::l) e = match l with
        | [] -> (evalEq a e)
        | l -> (evalEq a e) && (evalSyst l e);;

(* fonction bonus qui filtre les environnements valides ; un environnement est valide si
il est valide pour chaque equation du systeme *)

let rec getEnvValide syst (e::l) = match l with
```

```
        | [] -> if (evalSyst syst e) then [e] else []
        | _ -> if (evalSyst syst e) then ([e]@(getEnvValide syst l)) else (getEnvValide
syst l);;

(* fonction bonus qui prend un systeme et retourne les environnements valides *)

let solutions syst = if (syst == []) then [] else getEnvValide syst (gen (determin syst));;

(* _____________________________________________________ *)


let equation1 = EQ(OR(V 1, V 2), TRUE);;
let equation2 = EQ(XOR(V 1, V 3), V 2);;
let equation3 = EQ(NOT(AND(V 1, (AND(V 2, V 3)))), TRUE);;
let syst1 = (equation1)::(equation2)::(equation3)::[];;

let equation4 = EQ(XOR(V 1, V 2), FALSE);;

(* _____________________________________________________ *)

let test1 = determin syst1;;
let test2 = gen test1;;
let test3 = (List.nth test2 0);;
let test4 = evalEq equation1 test3;;
let test5 = evalSyst syst1 test3;;
let test6 = getEnvValide syst1 test2;;
let test7 = solutions syst1;;
(* ci-dessus l'exemple du sujet avec une petite erreur, en effet [V1 = TRUE; V2 =
FALSE; V3 = TRUE] est une solution *)

(* le seul test qu'on a pas avec l'exemple c'est celui qu'une equation avec FALSE
dedans... Le voici *)

let test8 = solutions (equation4::[]);;
```

```
(* _____ *)

(* les tests pour nous rassurer sur le bon fonctionnement de filtrer et unionPartiel *)

let test1 = filtrer [];;
let test2 = filtrer [1];;
let test3 = filtrer [1;2;3];;
let test4 = filtrer [1;1;2;3;3;3;4;4;4;6;6;6;9];;

let test5 = unionPartiel [] [];;
let test6 = unionPartiel [1] [];;
let test7 = unionPartiel [] [2];;
let test8 = unionPartiel [1;2;3] [1;2;3];;
let test9 = unionPartiel [1;3;5] [2;4];;
let test10 = unionPartiel [1;3;4;7;9;11] [2;2;3;4;5;5;6;6;6;7;7;11;13];;
```

7. Conclusion

L'algorithme demandé devait permettre de calculer l'ensemble des solutions d'un système d'équations booléennes.
C'est exactement ce que fait cet algorithme. Appliquez "solutions" sur votre système et laissez la magie opérer.