Rapport IPF

Gautier Poursin

4 mai 2020

Introduction

Dans ce projet, l'objectif est d'étudier des formules logiques. C'est à dire que nous allons construire l'arbre de vérité des formules, vérifier si une formule est juste en fonction des valeurs données aux variables.

Tour d'abord, nous nous intéresserons aux arbres de décision avec des fonctions qui permettent de récupérer les variables d'une formule, d'évaluer une formule ou même de construire son arbre de décision.

Ensuite, nous construirons les Binary Decision Diagram, en essayant de les simplifier.

Enfin, nous ferons la vérification des formules données.

Vous retrouverez un fichier **test.ml** qui contient des exemples de test effectués ainsi qu'un fichier **projel.mli** qui contient toutes les définitions de fonctions. Il y a 2 PDF présents pour illustrer les graphiques obtenus avec les questions bonus et les formules de l'énoncé. Ils sont accompagnés par les fichiers .dot. **J'ai défini la fonction buildBDD seulement pour permettre à mon code de compiler. Le code et la bdd renvoyés sont faux. Elle est seulement définie pour compiler les fonctions areEquivalent et isTautology. J'ai testé ces fonctions en utilisant directement la BDD de l'énoncé.**

Arbres de décision

Dans cette partie, nous allons étudier les arbres de décisions. La première fonction est **getVars**, qui prend en argument une **tformula** et qui renvoie une **string list**. Pour pouvoir réaliser cette fonction, j'ai tout d'abord crée une fonction **supprimedoublons**, qui prend en argument une **list** et renvoie la liste sans doublons. **val supprimedoublons: 'a list-> 'a list = <fun>**. Cette fonction fait appel à la fonction **existe : 'a->'a list -> 'a list=<fun>**. Elle permet de supprimer les doublons en cas de liste de taille impaire et donc on peut comparer le dernier élément.

```
# let rec supprimedoublons t= match t with
 |[]->[]
 |h::[]->[h]
 |h::n::t1 -> if (h=n) then [h]@(supprimedoublons t1)
             else [h;n]@(supprimedoublons t1);;
val supprimedoublons : 'a list -> 'a list = <fun>
```

Figure 1: La fonction pour supprimer les doublons d'une liste

Ensuite, j'ai pu créer la fonction récursive **getVars : tformula ->string list = <fun>**. Cette fonction consiste à regarder quelle **tformula** on a et puis en fonction de la **tformula**, on effectue différentes actions, d'où le **match f with**. Si on a pas de variable dans la **tformula**, on renvoie une liste vide. Si on a qu'une variable, on renvoie une liste comportant cette seule variable. Sinon, on réutilise la fonction **getVars** en triant puis en supprimant les doublons de la liste. Il est nécessaire de comparer et trier la liste, pour pouvoir facilement supprimer les doublons, à l'aide des fonctions **List.sort** et **compare**.

```
# let rec getVars (f:tformula)= match f with
  |Value true -> []
  |Value false->[]
  |Var (v) -> [v]
  |Not (t) -> supprimedoublons(List.sort compare (getVars t))
  |And (t1,t2)->supprimedoublons(List.sort compare (getVars t1  @ getVars t2))
  |Or(t1,t2)->supprimedoublons( List.sort compare (getVars t1 @getVars t2))
  |Implies(t1,t2)->supprimedoublons(List.sort compare (getVars t1 @getVars t2))
  |Equivalent(t1,t2)->supprimedoublons(List.sort compare (getVars t1 @ getVars t2));;

val getVars : tformula -> string list = <fun>
```

Figure 2: Fonction permettant d'obtenir les variables d'une formule

Après avoir créer ces fonctions, il me restait à construire l'arbre de décision d'une formule, à l'aide de la fonction **evalFormula: env -> tformula -> bool=<fun>**. J'ai tout d'abor créer 2 fonctions préalables **search:env ->string -> string*bool=<fun>** et **variable: env->string -> bool=<fun>** . Ces fonctions me permettent d'abord de chercher une variable dans un **env** puis ensuite d'avoir la valeur que l'on donne à cette variable. Je peux maintenant construire ma fonction **evalFormula**, qui sera récursive. Il suffit de matcher la **tformula** en argument avec les différents valeurs qu'elle peut prendre. Si c'est un booléen, on renvoie le booléen. Si c'est une variable, on renvoie la valeur associée à la variable dans l'**env** avec les fonctions définies au préalable. Enfin, pour toutes les autres valeurs, j'ai utilisé une table de vérité pour savoir ce que renvoyais un **And**, **Or**... J'ai donc réappelé la fonction **evalFormula** pour les **tformula** présentes dans les **And,Or...** avec le même **env** initial puis j'ai matché les **evalFormula** avec des couples de booléen puis je renvoie ce que cela

vaut en fonction du couple.

```
let rec evalFormula (t:env) (f:tformula)= match f with
 |Value v->v
 |Var v-> variable t v
 |Not v-> if ((evalFormula t v)=true) then false else true
 |And(v1,v2) -> (match (evalFormula t v1, evalFormula t v2) with
                 |(true,true)->true
                 |(_,_)->false)
 |Or(v1,v2) -> (match (evalFormula t v1, evalFormula t v2) with
                 |(false,false)->false
                 |(_,_)->true)
 |Implies (v1,v2)->(match (evalFormula t v1, evalFormula t v2) with
                 |(true, false)->false
                 |(_,_)->true)
 |Equivalent (v1,v2)->(match (evalFormula t v1, evalFormula t v2) with
                 |(true, true)->true
                 |(false, false)->true
                 |_,_->false);;
          type env = (string * bool) list
val search : env -> string -> string * bool = <fun>
#   val variable : env -> string -> bool = <fun>
#                                    val evalFormula : env -> tformula -> bool =
<fun>
```

Figure 3: Fonction permettant d'évaluer une formule

Enfin, je peux construire ma fonction **buildTree:tformula-> decTree=<fun>**. Pour cela, j'ai utilisé une fonction auxiliaire **buildBddAux: tformula -> env -> string list -> decTree= <fun>** . Pour cette fonction auxiliaire, je vais matcher ma **string list**. Si c'est une liste vide, je renvoie un **DecLeaf**, qui fait appelle à la fonction **evalFormula**.Sinon, on construit un **DecRoot**, en faisant appel à **buildTreeAux** 2 fois: pour le sous arbre gauche, on concatène l'**env** avec **false** et avec **true** pour le sous arbre droit. Quand au noeud, ce sera la variable extraite de la **string list** et on utilisera la nouvelle **string list** dans **buildTreeAux**.

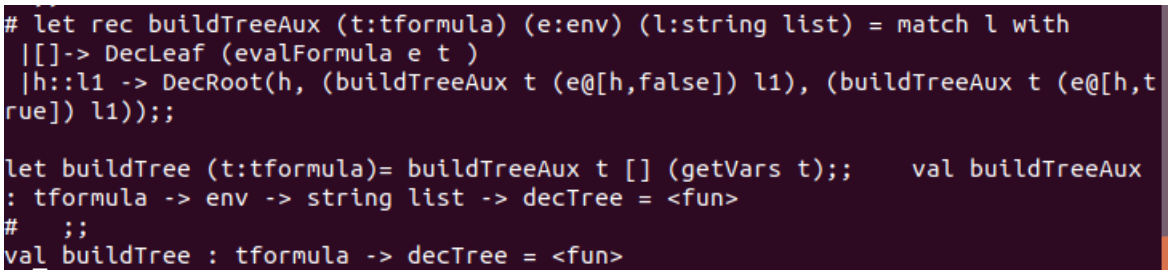

Figure 4: Fonction permettant de construire l'arbre de décision une formule

Construction de la BDD

Je n'ai pas réussi à coder cette partie, mais j'aurais peut être une idée d'algorithme à mettre en place. La première étape de cette algorithme consisterait à récupérer tous les sous arbres d'un arbre. Une fois ces sous arbres récupérer, il faut regarder lesquels sont identiques et on ne conserve en mémoire que l'arbre de base ainsi que les différents sous arbres possibles. Inutile de garder 2 sous arbres identiques. Ensuite, on va numéroter ces sous arbres. Cette numérotation ne change par au cours de l'algorithme.Ensuite, on utilise une fonction récursive pour parcourir notre arbre de base. Lorsqu'on est tout en haut de l'arbre, on note récupère le numéro associé à l'arbre: il s'agit de l'entier dans la bdd. On commence à former la liste de la bdd. Il suffit de reconnaitre les sous arbres associés au noeud dans lequel nous nous situons pour ensuite ajouter leur numéro. SI jamais le numéro du sous arbres à déja été ajouté dans la liste en tant que numéro de noeud, il faut passer ce sous arbre et regarder ses sous arbres directement.

BDD simplifié

Dans cette partie, l'objectif est de simplifier la Binary Decision Diagram en ignorer les noeuds qui ont la forme **bddNode (a,b,n,n)**.

Tout d'abord, j'ai créer une fonction **sup : bddNode list -> bddNode list=<fun>** . Cette fonction permettra de supprimer les BddNode d'une liste de la forme **BddNode (n,s,p,p)**. J'ai créer une fonction **remplace: int -> int ->bddNode list ->bddNode list =<fun>**. Cette fonction permet de remplacer le premier entier par le deuxième dans une liste de BddNode. Ensuite, j'ai créer 2 fonctions **double : int*bddNode list -> int list=<fun>** qui permet de récupérer la liste de tous les noeuds des BddNode de la forme **BddNode(n,s,p,p)** ainsi que la fonction **recup : int-> 'a*bddNode list -> int=<fun>** qui permet de récupérer l'entier p d'une **BddNode (n,s,p,p)**, avec la valeur du noeud en argument.

Enfin, pour construire ma fonction **simlifyBDD : bdd ->bdd=<fun>**, j'ai assemblé mes fonctions précédement créer. Tout d'abord, je récupére la liste des noeuds des **BddNode (n,s,p,p)**. Ensuite, je récupère la longueur de cette liste avec la fonction **List.length()**. J'implémente la fonction récursive **remplacement: bdd -> int-> bdd =<fun>** qui va remplacer dans ma bdd les entiers n par l'entier p. Elle va le faire autant de fois qu'il y a de **BddNode(n,s,p,p)**. Cette fonction fait appel à la fonction **remplace**, **recup** et **List.nth**. Ensuite, il suffit de supprimer les **BddNode (a,b,c,c)** avec la fonction **sup** puis on obtient le résultat voulu. Un exemple détaillé est en fin de sujet. **Pour utiliser cette fonction, il faut au préalable être sur d'avoir dans sa liste une bdd de la forme BddNode (n,s,p,p). Sinon, le compilateur renvoie une erreur puisqu'il cherchera dans une liste l'élément à l'index -1.**

```
#
let simplifyBDD b=
let d=double b in
let i=longueur d-1 in
let rec remplacement b i= match i with
|0->(fst(b), remplace (List.nth d 0) (recup (List.nth d 0) b)   (snd(b)))
|a-> remplacement ((fst(b), (remplace (List.nth d a) (recup (List.nth   d a) b)   (snd(b))))) (a-1) in
let sup=sup (snd(remplacement b i))
in (fst(b), sup);;
val simplifyBDD : int * bddNode list -> int * bddNode list = <fun>
#
```

Figure 5: Fonction permettant de simplifier la BDD

Verification des formules

Pour cette partie, l'objectif est de verifier la validité d'une formule. Je vais dans un premier temps vérifier si une formule est une tautologie à l'aide de la fonction **isTautology : tformula -> bool=<fun>**. Pour cette fonction, je vais utiliser la fonction **buildBDD** et **simplifyBDD**. Il suffit de matcher **simplifyBDD(buildBDD f)**, où f est la **tformula** en argument. Si cela match avec la formule donnée par l'énoncé, alors c'est une tautologie et on renvoie **true**. Sinon, on renvoie false.

```
let isTautology (f:tformula)= let m= simplifyBDD(buildBDD f) in match m with
|(_,[BddLeaf(_,true)])->true
|_->false;;

  val isTautology : tformula -> bool = <fun>
```

Figure 6: Fonction permettant de regarder si une formule est une tautologie

Désormais, je souhaite vérifier si 2 formules sont équivalentes au numéro des noeuds près. Pour cela, je vais utiliser une fonction **compare : 'a list -> 'a list -> bool=<fun>** . Avec cette fonction, je vais pouvoir construire

ma fonction **areEquivalent : tformula -> tformula -> bool=<fun>**. Je vais construire les BDD associées aux tformula puis les simplifier et comparer le second élément des couples obtenus après simplification, qui sont bien les listes des noeuds. Je vérifie seulement le second élément du couple et non le premier car si les premiers sont différents, cela implique nécessairement une différence au niveau des listes.

```
let rec compare t1 t2= match t1,t2 with
  |[],[]-> true
  |_,[]->false
  |[],_->false
  |h::t3,n::t4-> if (h=n) then (compare t3 t4)
                 else false;;
            val compare : 'a list -> 'a list -> bool = <fun>
```

Figure 7: Fonction permettant de comparer 2 listes

```
#   let areEquivalent (t:tformula) (f:tformula)= comparer (snd(simplifyBDD(build
BDD t))) (snd(simplifyBDD(buildBDD f)));;
val areEquivalent : tformula -> tformula -> bool = <fun>
```

Figure 8: Fonction permettant de vérifier si 2 formules sont équivalentes

Bonus

Pour réaliser la fonction **dotBDD : string -> bdd -> unit**, je dois tout d'abord ouvrir le fichier rentré en argument. Ensuite, j'écris le début du fichier qui est *digraph G {*. Ensuite, je ne travaille qu'avec la liste de la bdd avec la fonction **snd**. Je fais appel à une fonction récursive **ecrire : list -> bool**. Si ma liste est vide, je renvoie true. Sinon, j'écris dans mon fichier les lignes de codes correspondantes si j'obtiens une BddNode ou BddLeaf puis je réappelle ma fonction ecrire. Enfin, je ferme mon fichier en placant } à la fin.

```
let dotBDD (nom:string) (b:bdd)=
let file=open_out nom in
let init=output_string file "digraph G {\n" in
let list=snd(b) in
let rec ecrire l2= match l2 with
|[]-> true
|h::l-> match h with |BddLeaf (a,b) -> (output_string file (String.concat "" [(s
tring_of_int a);"[style=bold, label=";(string_of_bool b);"]";";\n"])); (ecrire l
);
                     |BddNode (a,b,c,d)-> (output_string file (String.concat ""
[(string_of_int a);" [label=";b;"];\n";(string_of_int a);"->";(string_of_int c);
" [color=red, style=dashed];\n";(string_of_int a);"->";(string_of_int d);";\n"])
); (ecrire l)
in let ecriture=output_string file (string_of_bool (ecrire list)) in
let ende=(output_string file "}") in close_out(file);;
```

Figure 9: Fonction permettant de tracer la description d'une BDD

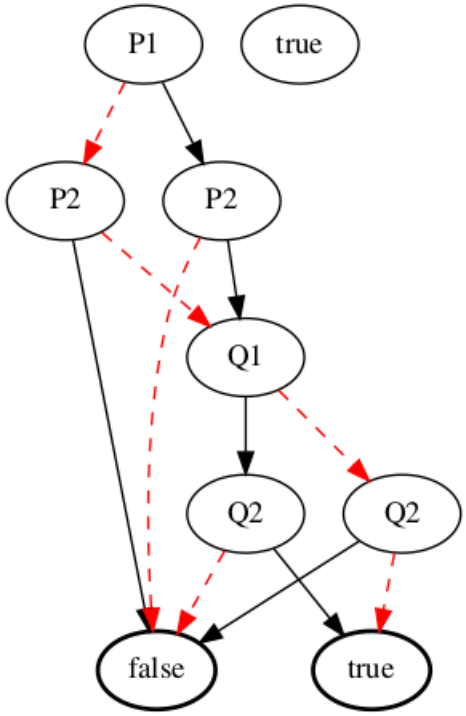

Figure 10: Graphique obtenu avec la BDD simplifié donné en exemple

Ce graphique est le miroir de l'exemple sur le projet. Cependant, je n'arrivais

pas a supprimer l'elipse "true" situé juste au dessus. Cela est du au fait que je termine ma fonction récursive par true pour toujours renvoyer un booléen et ne par avoir une erreur de typage. Il faudrait dans le fichier dot générer remplacer ce "true" par " ". Pour remedier a cela, j'ai légèrement modifié ma fonction récursive écrire. Au lieu de renvoyer un booléen, elle renvoie désormais un **unit** et ma variable **ecriture** prend la valeur **ecrire list**, avec list qui est en argument de ma fonction **dotBDD**. Voici la nouvelle écriture du code et le graphique associé.

```
let dotBDD (nom:string) (b:bdd)=
let file=open_out nom in
let init=output_string file "digraph G {\n" in
let list=snd(b) in
let rec ecrire l2= match l2 with
|[]->(output_string file "}")
|h::l-> match h with |BddLeaf (a,false) -> (output_string file (String.concat "" [(string_of_int a);"[style=bold, label=";(string_of_bool fa
lse);"]";";\n"])); (ecrire l);
|BddLeaf (a,true) -> (output_string file (String.concat "" [(string_of_int a);"[style=bold, label=";(string_of_bool true);"]";";\n"])); ecri
re l;
                    |BddNode (a,b,c,d)-> (output_string file (String.concat "" [(string_of_int a);" [label=";b;"];\n";(string_of_int a);"->
";(string_of_int c);" [color=red, style=dashed];\n";(string_of_int a);"->";(string_of_int d);";\n"])); (ecrire l)
in let ecriture=(ecrire list) in
init;ecriture; close_out(file);;
```

Figure 11: Nouvelle fonction permettant de tracer la description d'une BDD

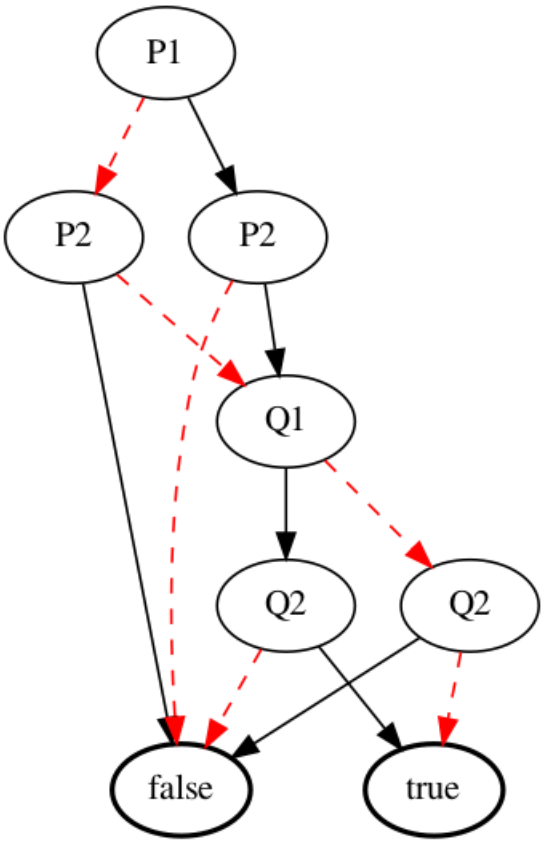

Figure 12: Graphique obtenu avec la BDD simplifié donné en exemple

Ensuite, pour tracer la fonction **dotTree: string ->decTree-> unit**, je me suis fortement inspiré de la fonction précédente. Cependant, il fallait prendre en compte la numérotation des noeuds. C'est pour cela que ma fonction récursive prend en argument un entier. Cette entier est incrémenté au fur et a mesure des appels à la fonction et pour éviter d'avoir 2 entiers identiques, j'ai decidé de travailler avec les paires et les impairs. Le sous arbre gauche sera toujours pair tandis que le sous arbre droit sera impair. Ainsi, mon fichier dot sera bien organisé pour éviter d'avoir un arbre avec des "lianes entre tous les noeuds.

```
# let dotTree (nom:string) (t:decTree)=
let file=open_out nom in
let init=output_string file "digraph G {\n" in
let rec ecrire i t= match t with
|DecLeaf(a) -> (output_string file (String.concat "" [(string_of_int i);"[style=
bold, label=";(string_of_bool a);"]";";\n"]));
|DecRoot (a,sag,sad) -> (output_string file (String.concat "" [(string_of_int i)
;"[label=";a;"]";";\n"]));
(output_string file (String.concat "" [(string_of_int (i));"->";(string_of_int (
2*i));"[color=red, style=dashed]";";\n  "]));
(output_string file (String.concat "" [(string_of_int (i));"->";(string_of_int (
2*i+1));";\n"])); ecrire (2*i) sag; ecrire (2*i+1) sad;
in let ecriture=ecrire 1 t in
let ende=(output_string file "}") in init;ecriture;ende; close_out(file);;
                val dotTree : string -> decTree -> unit = <fun>
```

Figure 13: Fonction dotTree

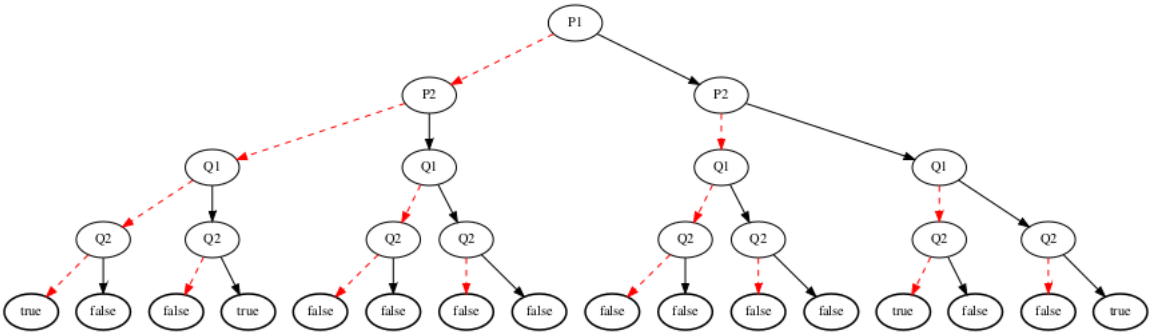

Figure 14: Graphique obtenu avec l'arbre du projet

Résumé des interfaces des fonctions

existe : 'a->'a list -> 'a list=<fun>
getVars : tformula ->string list = <fun>
val supprimedoublons: 'a list-> 'a list = <fun>
evalFormula: env -> tformula -> bool=<fun>
search:env ->string -> string*bool=<fun>
variable: env->string -> bool=<fun>
buildTree:tformula-> decTree=<fun>
buildBddAux: tformula -> env -> string list -> decTree= <fun>
sup : bddNode list -> bddNode list=<fun>
remplace: int -> int ->bddNode list ->bddNode list =<fun>
double : int*bddNode list -> int list=<fun>

recup : int-> 'a*bddNode list -> int=<fun>
simlifyBDD : bdd ->bdd=<fun>
remplacement: bdd -> int-> bdd =<fun>
isTautology : tformula -> bool=<fun>
compare : 'a list -> 'a list -> bool=<fun>
areEquivalent : tformula -> tformula -> bool=<fun>
dotBDD : string -> bdd -> unit
ecrire : list -> bool
En complément de cette partie,veuillez trouver joint au projet un fichier **projet.mli** qui regroupe toutes les définitions de fonctions.

Exemple

On pose :
let p1= Var "P1";;
let p2= Var "P2";;
let q1= Var "Q1";;
let q2= Var "Q2";;
let r1= Var "R1";;
let r2= Var "R2";;
let i1=Or(r1,r2);;
let i2=Implies(r1,r2);;
let f1= Equivalent(q1,q2);;
let f2= Equivalent(p1,p2);;
let ex1= And(f1,f2);;
let ex2= (10, [BddNode (10, "P1", 8, 9); BddNode (9, "P2", 7, 5); BddNode (8, "P2", 5, 7); BddNode (7, "Q1", 6, 6); BddNode (6, "Q2", 2, 2); BddNode (5, "Q1", 3, 4); BddNode (4, "Q2", 2, 1); BddNode (3, "Q2", 1, 2); BddLeaf (2, false); BddLeaf (1,true)]);;
let ex3=And((And(f1,f2)),(Or(i1,i2)));;
let ex4=["P1";"P1";"Q2";"Q3";"Q3"];;
let ex5=["P1",true;"P2",true;"Q1",true;"Q2",true;"R1",false;"R2",true];;
let ex6=["P1",false;"P2",true;"Q1",true;"Q2",true;"R1",false;"R2",true];;
let ex7=And (q1,q2);;
let ex8= ["Q1",true;"Q2",false];;
let ex9=["Q1",true;"Q2",true];;
let ex10=[BddNode (5,"P1",4,3); BddNode (4,"P2",3,2); BddNode (3,"Q1",2,1); BddLeaf (2,false); BddLeaf(1,true)];;
let ex11=(5,[BddNode (5,"P1",4,3); BddNode (4,"P2",3,2); BddNode (3,"Q1",2,1); BddLeaf (2,false); BddLeaf(1,true)]);;

```
# getVars ex3;;
- : string list = ["P1"; "P2"; "Q1"; "Q2"; "R1"; "R2"]
```

Figure 15: Getvars ex3

```
# supprimedoublons ex4;;
- : string list = ["P1"; "Q2"; "Q3"]
```

Figure 16: supprime doublons ex4

```
# evalFormula ex5 ex3;;
- : bool = true
```

Figure 17: evalFormula ex5 ex3

```
# evalFormula ex6 ex3;;
- : bool = false
```

Figure 18: evalFormula ex6 ex3

```
# simplifyBDD ex2;;
- : int * bddNode list =
(10,
 [BddNode (10, "P1", 9, 8); BddNode (9, "P2", 4, 3); BddNode (8, "P2", 2, 7)
  BddNode (7, "Q1", 3, 2); BddNode (4, "Q2", 2, 1); BddNode (3, "Q2", 1, 2);
  BddLeaf (2, false); BddLeaf (1, true)])
#
```

Figure 19: simplifyBDD ex2

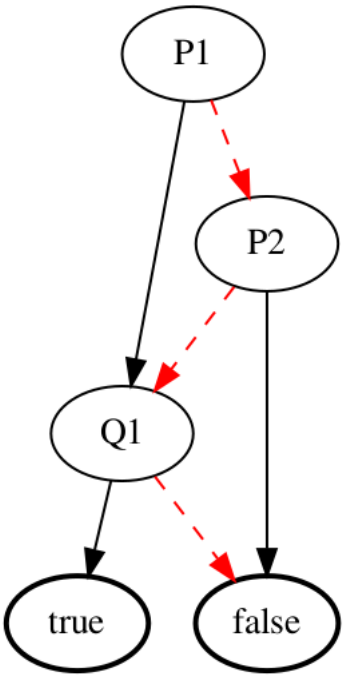

Figure 20: dotBDD "exemple.dot" ex11