Projet IPF - Formules logiques et BDD

1 Travail à rendre

Le projet est à réaliser en OCaml *individuellement.* Il sera accompagné d'un rapport qui doit contenir *impérativement* la description des choix faits, des types et des fonctions. Pour chaque fonction, on donnera impérativement l'interface complète. Le rapport fournira également des cas de tests accompagnés des résultats retournés. Pour cela, vous pouvez utiliser des assertions, comme vu en TP.

Vous devez rendre **votre rapport** (en pdf) et **vos fichiers de code** rassemblés dans une archive identifiée comme `votre_prénom_votre_nom.tgz`. Une archive qui ne contient pas les fichiers demandés ne sera pas excusable. Votre archive doit être déposée sur http://exam.ensiie.fr, dans le dépôt `ipf_projet_2020`, **avant le 4 Mai 2020, à 23 :59 CET**.

Vous devrez **IMPÉRATIVEMENT** respecter les noms des types, constructeurs et fonction donnés dans le sujet. Vous êtes libre de définir tout type ou fonction qui vous semblera utile.

2 Un peu de logique

Dans ce projet, on s'intéresse aux formules de la *logique propositionnelle* que l'on appelle aussi propositions. Une proposition est une affirmation qui peut être vraie ou fausse. On associe à toute proposition sa valeur de vérité (unique), notée V (pour vrai) ou F (pour faux). Elle ne peut prendre qu'une seule valeur, indépendamment de tout paramètre : ou bien une proposition est vraie et elle le sera en toutes circonstances ou bien elle est fausse et elle sera fausse en toutes circonstances. En cela une proposition est différente d'expressions telles que $p(e)$ qui affirme que l'élève e aura une note supérieure à 10, expression qui peut avoir pour valeur de vérité V ou F, selon l'élève.

Les propositions sont formées avec des variables logiques appelées variables propositionnelles et des connecteurs logiques. Les variables ont une valeur de vérité (vrai ou faux) et sont notées dans la suite $P, Q, P_1, P_2, \ldots, P_n$. Les connecteurs sont $\neg$ la négation, $\vee$ la disjonction (le ou), $\wedge$ la conjonction (le "et"), l'implication $\Rightarrow$ (le "si ... alors ...") et l'équivalence $\Leftrightarrow$ (le "si et seulement si"). On ajoute à ceci, deux constantes $\top$ et $\bot$.[1] Une table de vérité donne la valeur d'une formule en envisageant tous les cas possibles pour les variables propositionnelles. Ainsi si une formule compte n variables, il faudra examiner 2^n cas. Les tables de vérité élémentaires rappelées ci-dessous (Figure 1) décrivent le comportement de chaque connecteur :

P	$\neg P$
V	F
F	V

P_1	P_2	$P_1 \wedge P_2$	$P_1 \vee P_2$	$P_1 \Rightarrow P_2$	$P_1 \Leftrightarrow P_2$
V	V	V	V	V	V
V	F	F	V	F	F
F	V	F	V	V	F
F	F	F	F	V	V

FIGURE 1 – Tables de vérité élémentaires.

1. dans la suite du sujet, les connecteurs ont une priorité. Ils sont donnés ici dans l'ordre des priorités décroissantes : $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$ et $\Rightarrow$. Par exemple, $\neg P_1 \wedge P_2 \vee P_3 \Rightarrow P_4$ doit être comprise comme $(((\neg P_1) \wedge P_2) \vee P_3) \Rightarrow P_4$. Si on veut contrecarrer les priorités on utilise des parenthèses.

P	Q	$P \Rightarrow Q$	$\neg P \vee Q$	$(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$
V	V	V	V	V
V	F	F	F	V
F	V	V	V	V
F	F	V	V	V

FIGURE 2 – Table de vérité de $(P \Rightarrow Q) \Leftrightarrow \neg P \vee Q$

Par exemple, la table de vérité de $(P \Rightarrow Q) \Leftrightarrow \neg P \vee Q$ est donnée dans la Figure 2. Chaque ligne de la table correspond à une assignation : ainsi la première ligne correspond au cas ou P et Q ont la valeur de vérité V.

On dit que deux formules sont *équivalentes* si elles ont même table de vérité ou même valeur de vérité pour toute assignation possible.

Une formule est une *tautologie* si sa valeur de vérité est V pour chaque assignation possible. La formule considérée à la Figure 2 est une tautologie.

Une formule est dite *satisfiable* si sa valeur de vérité est V pour au moins une assignation, pour chaque assignation possible.

En général, il est difficile de déterminer si une proposition est satisfiable. Il s'agit d'un problème dit NP-complet [2]. On s'intéresse dans la suite à des représentations plus ou moins compactes des formules. Ces représentations dépendent de l'ordre dans lequel on va considérer les variables des formules.

3 Arbres de décision

Soit f une proposition et $P_1, P_2, \ldots, P_n$ la liste de ses variables (donnée dans un certain ordre). L'arbre de décision pour f selon l'ordre des variables $P_1, P_2, \ldots, P_n$ est un arbre binaire étiqueté par P_1, dont le sous arbre gauche est l'arbre de décision pour la formule obtenue en remplaçant P_1 par *faux* dans f et dont le sous arbre droit est l'arbre de décision pour la formule obtenue en remplaçant P_1 par *vrai* dans f. Les feuilles sont les valeurs de vérité obtenues.

L'arbre de décision est une représentation arborescence de la table de vérité. Par exemple, l'arbre de décision obtenue pour la formule $(Q_1 \Leftrightarrow Q_2) \wedge (P_1 \Leftrightarrow P_2)$ pour l'ordre P_1, Q_1, P_2, Q_2 est le suivant :

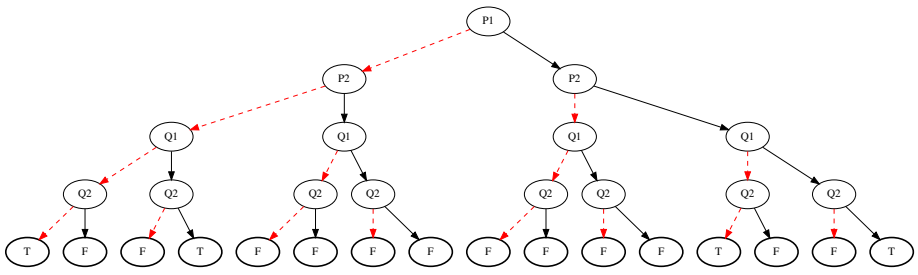


FIGURE 3 – Arbre de décision.

On se donne, pour le moment les deux définitions de types suivante correspondant respectivement aux formules et aux arbres de décision.

```
type tformula =
  | Value of bool                          (* ⊥ ou ⊤  *)
  | Var of string                          (* Variable *)
  | Not of tformula                        (* Negation *)
  | And of tformula * tformula             (* Conjonction *)
  | Or of tformula * tformula              (* Disjonction *)
  | Implies of tformula * tformula         (* Implication *)
```

2. Un problème NP-complet ne peut être résolu que par un algorithme qui s'exécute en un temps qui croît de manière exponentiel avec la taille de problème

```
  | Equivalent of tformula * tformula                    (* Equivalence *)

type decTree =
  | DecLeaf of bool
  | DecRoot of string * decTree * decTree
```

`Valeur true` représentera $\top$ et `Valeur false` représentera $\bot$

La formule $\texttt{ex}_1$ définie par $(Q_1 \Leftrightarrow Q_2) \wedge (P_1 \Leftrightarrow P_2)$ pourra, par exemple, être définie à l'aide du programme suivant à l'aide de ces types :

```
let p1 = Var "P1"
let p2 = Var "P2"
let q1 = Var "Q1"
let q2 = Var "Q2"
let f1 = Equivalent(q1,q2)
let f2 = Equivalent(p1,p2)
let ex1 = And(f1,f2)
```

Question 1 : Écrire une fonction `getVars : tformula -> string list` qui sur la donnée d'une formule `f` retourne la liste des variables apparaissant dans `f`. Chaque variable présente dans `f` ne devra apparaître qu'une fois et la liste devra être triée par ordre croissant de l'ordre naturel sur les chaînes de caractère (pensez à utiliser la fonction `compare` et la fonction `List.sort`). Ainsi `getVars ex1` devra retourner `["P1";"P2";"Q1";"Q2"]`

Il est possible, connaissant la valeur des variables d'une formule, d'évaluer la valeur globales de cette formule. Nous définissons pour cela le type des environnements comme étant **type** `env = (string*bool) list`

Question 2 : Définir une fonction `evalFormula : env -> tformula -> bool` qui évalue une formule `f` dans un environnement `env`. On pourra supposer que toutes les variables présentes dans `f` ont une valeur définie dans `env`.

À titre d'exemple, `evalFormula ["P1",`**false**`;"P2",`**false**`;"Q1",`**false**`;"Q2",`**false**`] ex1` devra retourner **true**.

Question 3 : Écrire une fonction `buildDecTree : tformula -> decTree` qui construit l'arbre de décision d'une formule. Les variables seront traitées dans l'ordre donné par la fonction `getVars`.

les résultat de `buildDecTree ex1` devra être

```
DecRoot ("P1",
 DecRoot ("P2",
  DecRoot ("Q1", DecRoot ("Q2", DecLeaf true, DecLeaf false),
   DecRoot ("Q2", DecLeaf false, DecLeaf true)),
  DecRoot ("Q1", DecRoot ("Q2", DecLeaf false, DecLeaf false),
   DecRoot ("Q2", DecLeaf false, DecLeaf false))),
 DecRoot ("P2",
  DecRoot ("Q1", DecRoot ("Q2", DecLeaf false, DecLeaf false),
   DecRoot ("Q2", DecLeaf false, DecLeaf false)),
  DecRoot ("Q1", DecRoot ("Q2", DecLeaf true, DecLeaf false),
   DecRoot ("Q2", DecLeaf false, DecLeaf true))))
```

L'étape suivante consiste à éliminer les redondances dans cet arbre de décision, le résultat sera un Binary Decision Diagram (BDD).

4 Binary Decision Diagram

Si l'étape précédente construisait un arbre, il s'agit ici de construire un *graphe acyclique*. On remarque que dans l'arbre précédent le sous arbre :

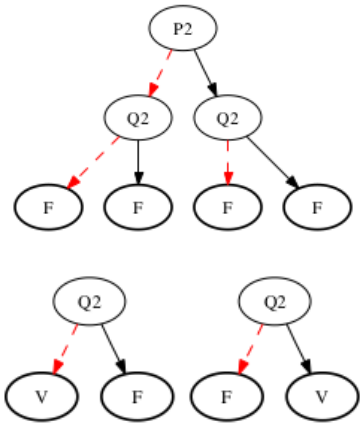
est répété deux fois. On va donc le partager. De même pour les sous-arbres

Remarquer aussi que le sous-arbre droit de $Q1$ côté droit est identique au sous-arbre gauche de $Q1$ côté gauche. La nouvelle représentation de la formule est donnée à la Figure 3.

On va représenter une telle structure par une liste de nœuds de type `bddNode` et le nœud racine.

On propose de numéroter chacun des nœuds du graphe. Il y a deux sortes de nœuds : les feuilles qui contiennent une valeur de vérité (de type `bool`), les autres nœuds contenant un nom de variable (de type `string`) et un successeur gauche et un successeur droit qui sont tous deux des numéros de nœuds.

On défini donc les deux types suivants :




```
type bddNode =
  | BddLeaf of int*bool
  | BddNode of int * string * int * int

type bdd = (int * (bddNode list)) (* un entier pour designer le noeud racine et
                                     la liste des noeuds *)
```

Le BDD de la Figure 4 est représenté en Ocaml par :

```
(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)])
```

(Toute autre numérotation des nœuds et ordre dans la liste sont possibles)

Question 4 : Écrire une fonction `buildBdd : tformula -> bdd` qui construit le BDD d'une formule. On ne construira pas la table de décision au préalable. *Idée de l'algorithme* : à chaque étape de construction il s'agit de vérifier dans le graphe déjà construit si le nœud que l'on désire construire est déjà présent ou non. S'il est présent, on le réutilise en récupérant son numéro, sinon on l'ajoute à la liste.

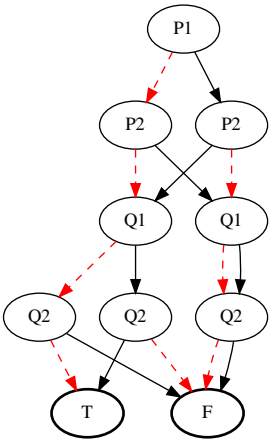


FIGURE 4 – Exemple BDD

5 BDD simplifié

On peut encore optimiser la structure précédente en éliminant les nœuds de la forme `bddNode (n, s, p, p)` : les successeurs gauche et droit sont alors les mêmes. L'idée est alors d'ignorer le nœud `n`. Tout nœud qui a pour successeur (gauche ou droit) le nœud de numéro `n` aura désormais pour successeur le nœud de numéro `p`. La représentation graphique est à la figure suivante :

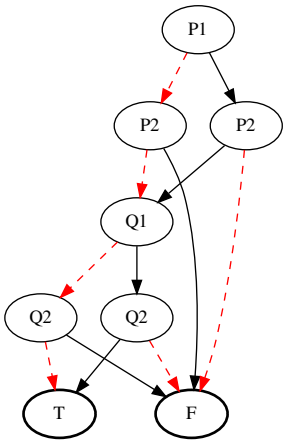


FIGURE 5 – Exemple BDD simplifié

Avec cette simplification, toute tautologie se réduit à un seul nœud : la feuille true (en OCaml `[BddLeaf (1, true)]`).

Question 5 : Écrire une fonction `simplifyBDD: bdd -> bdd` qui effectue la simplification.

`simplifyBDD (buildBdd ex1)` devra retourner (à l'ordre des noeuds et leur représentation prêts) :

```
(10,
[BddNode (10, "P1", 8, 9); BddNode (9, "P2", 2, 5); BddNode (8, "P2", 5, 2);
BddNode (5, "Q1", 3, 4); BddNode (4, "Q2", 2, 1); BddNode (3, "Q2", 1, 2);
BddLeaf (2, false); BddLeaf (1, true)])
```

6 Vérification de formules

Question 6 : En déduire une fonction `isTautology : tformula -> bool` qui prend une formule en paramètre et teste s'il s'agit d'une tautologie.

Question 7 : Écrire une fonction `areEquivalent : tformula -> tformula -> bool` qui teste si deux formules sont équivalentes. Pour cela on construira les deux BDDs simplifiés en se fixant un même ordre pour les variables. Dans ce cas elles sont équivalentes si les deux BDDs sont identiques (au numérotage des nœuds près).

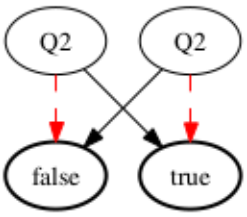
7 Bonus : Affichage graphique de BDD

On va utiliser un programme externe `dot` pour afficher les arbres. C'est un programme qui prend en entrée un fichier DOT décrivant un graphe et qui produit une sortie dans différents formats, comme postscript ou pdf.

Ces lignes dans un fichier de nom `essai.dot` :

```
digraph G {
4 [ label="Q2"];
4 -> 2 [color=red,style=dashed];
4 -> 1;
7 [ label="Q2"];
7 -> 1 [color=red,style=dashed];
7 -> 2;
2 [style = bold, label="false"];
1 [style = bold, label="true"];
}
```

. . . produisent la sortie suivante :



La commande pour produire le fichier `essai.pdf` est : `dot -Tpdf -o essai.pdf essai.dot`. Reste à utiliser un outil adapté pour visualiser le fichier pdf. Seule la production du fichier `.dot` se fait en OCaml. Dot peut se télécharger librement sur le site www.graphviz.org.

Pour la manipulation de fichiers, voir le document complémentaire ici : http://web4.ensiie.fr/~stefania.dumbrava/IPF/impression_fichiers.pdf.

Remarque : les dessins précédents ont été générés en utilisant dot.

Question 8 : Écrire une fonction `dotBDD : string -> bdd -> unit` qui, à partir d'un nom de fichier et d'un arbre de décision, produit un fichier qui contient la description du BDD au format DOT. Les feuilles seront dans des ellipses en gras. Les arcs indiquant un successeur gauche seront en pointillés ou en rouge. Les arcs indiquant un successeur droit seront en traits pleins. Pour faciliter la mise au point, vous pourrez faire apparaître les numéros des nœuds à côté des noms des variables et ainsi mieux vérifier la correspondance avec la structure OCaml calculée.
Question 9 : Écrire une fonction `dotDec: string -> decTree -> unit` qui, à partir d'un nom de fichier et d'un arbre de décision, produit un fichier qui contient la description de l'arbre de décision au format DOT.