Travaux pratiques 4

Résolution de clauses

Option informatique MP1, MP2 & MP3

Dans ce T.P. nous allons étudier un moyen de résoudre le problème de la satisfiabilité des formes normales (conjonctives) en transformant l'ensemble des clauses.

I Fonctions de base

Dans cette partie, comme dans la suite, les liste ne contiennent pas de doublons. On pourra supposer, sans avoir besoin de le vérifier, que les listes passées en paramètre ne contiennent pas d'élément en double et on devra s'assurer que les listes renvoyées vérifient la même propriété.

Question 1
Appartenance Écrire une fonction `mem : 'a -> 'a list -> bool` telle que `mem x liste` dit si `x` est un éléments de `liste`.

Question 2
Retrait Écrire une fonction `except : 'a -> 'a list -> 'a list` telle que `excepr t x liste` retourne une liste constituée des éléments de `liste` autres que `x`.

Question 3
Union Écrire une fonction `union : 'a list -> 'a list -> 'a list` telle que `union l1 l2` renvoie la liste **sans doublon** des éléments appartenant à `l1` ou à `l2`.

Question 4
Intersection Écrire une fonction `inter : 'a list -> 'a list -> 'a list` telle que `inter l1 l2` renvoie la liste des éléments appartenant à `l1` et à `l2`.

Question 5
Différence Écrire une fonction `moins : 'a list -> 'a list -> 'a list` telle que `delta l1 l2` renvoie la liste des éléments appartenant à `l1` mais pas à `l2`.

II Clauses

Définition 1 : clauses particulières

Une clause c est dite **triviale** s'il existe une variable booléenne x telle que x et $\neg x$ apparaissent dans c.
Une clause constituée d'aucun littéral est une contradiction.

Remarques

1. Une clause triviale est une tautologie et on peut la retirer d'une forme normale : on obtient une forme normale équivalente.
2. Une forme normale contenant une clause vide est une contradiction.

Implantation

- On représente une variable booléenne par une chaîne de caractères, `string`,
- Afin de pouvoir gérer les littéraux opposés (x et $\neg x$) on représente une clause par un couple de listes de chaînes de caractères ; la première liste rassemble les littéraux positifs (des variables), la seconde les littéraux négatifs (les négations de variables).
  Par exemple $x_1 \vee \neg x_5 \vee x_2$ pourra être représentée par `(["x1"; "x2"], ["x5"])`

```
type clause = (string list) * (string list);;
```

- Une forme normale sera représentée par une liste de clauses.

```
type fnc = clause list;;
```

Exemple. On considère les affirmations suivantes :

1. Quand ils ont un devoir, les élèves apprennent leur cours.
2. S'ils ont appris leur cours, ils n'ont pas de mauvaises notes.
3. S'ils n'ont pas de devoir, ils n'ont pas non plus de mauvaise notes.

On représente la variable booléenne "Les élèves ont un devoir" par `d`, la variable "Les élèves apprennent le cours" par `c` et "Les élèves ont une mauvaise note" par `m`, les affirmations s'écrivent `d` implique `c`, `c` implique non `n` et non `d` implique non `m`. On aboutit à la forme normale $(\neg d \vee c) \wedge (\neg c \vee \neg m) \wedge (d \vee \neg m)$ qui sera représenté par

```
let f0 = [(["c"], [ "d"]); ([ ], ["c"; "m"]); (["d"], ["m"])];;
```

Question 6
Écrire une fonction `triviale : clause -> bool` telle que `triviale c` teste si la clause c est triviale.

Remarque : on peut forcer le type `clause` de la variable en écrivant

```
let triviale (c : clause) =
```

Question 7
Écrire une fonction `no_triviale : fnc -> fnc` telle que `no_triviale f` renvoie une forme normale équivalente à f ne contenant pas de clause triviale.

Remarque : on peut aussi forcer le type du résultat en écrivant

```
let rec no_triviale (e : fnc) : fnc =
```

III Implication entre clauses

Définition 2 : Implication

La clause c **implique** la clause c' si tout littéral apparaissant dans c apparaît aussi dans c'. Dans un forme normale, une clause c est dite **minimale** si elle ne contient pas de clause c' distincte de c telle que c' implique c.

Remarques :

1. L'implication correspond à la notion usuelle : c implique c' si et seulement si $c \Rightarrow c'$ est une tautologie.
2. La clause vide implique toutes les autres.
3. On peut retirer les clauses non minimales d'une forme normale ; on obtient alors une forme normale équivalente.

Question 8
Écrire une fonction `implique : cause -> cause -> bool` telle que `implique c1 c2` indique si c_1 implique c_2

Question 9
Écrire une fonction `ajoute : clause -> fnc -> fnc` telle que `ajoute c f` appliquée à une clause c et une forme normale f renvoie une forme normale f' équivalente à $f \wedge c$ en retirant de $f \wedge c$ toutes les clauses c' telles que c implique c'. De la sorte, si l'on part d'une forme normale ne contenant que des clauses minimales, f' a aussi cette propriété. On n'ajoutera donc pas c si c est impliquée par une clause de f.

Question 10
En déduire une fonction `minimise : fnc -> fnc` telle que `minimise f` retourne une forme normale f' équivalente à f et ne contenant que des clauses minimales.

IV Consensus de deux clauses

Si c et c' sont deux clauses telles qu'il existe une variable x apparaissant positivement dans une clause et négativement dans l'autre : $c = x \vee c_1$ et $c' = \neg x \vee c'_1$ (c_1 et c'_1 ne contenant ni x ni $\overline{x}$), on appelle **consensus** de c et de c' la clause $c'' = c_1 \vee c'_1$. c'' a la propriété d'être vraie chaque fois que c et c' le sont, et c'est la plus petite clause pour l'ordre d'implication parmi les clauses indépendantes de x ayant cette propriété (c'est-à-dire si d est une clause indépendante de x, on a $c \wedge c' \Rightarrow d$ si et seulement si $c'' \Rightarrow d$). S'il existe une autre variable q apparaissant positivement dans l'une des clauses c ou c' et négativement dans l'autre, alors $c_1 \vee c'_1$ contient $q \vee \overline{q}$ donc est la clause triviale.
Il y a ainsi trois cas possibles pour deux clauses c et c' :

- ou bien elle n'ont pas de consensus (aucune variable n'apparaît positivement dans une clause et négativement dans l'autre),
- ou bien elles ont un unique consensus,
- ou bien elles ont plusieurs consensus mais alors ils sont tous triviaux.

Exemple. Le consensus de $(\neg d \vee c)$ et de $(d \vee \neg m)$ est $(c \vee \neg m)$: si les élèves ont une mauvaise note, ils ont appris leurs cours.

Question 11
Écrire une fonction `ajoute_cons : clause -> clause -> cnf -> cnf` telle que `ajoute_cons c1 c2 f` ajoute le consensus éventuel, s'il est non trivial, entre c_1 et c_2 à la forme normale f. On utilisera la fonction `ajoute` pour éliminer les clauses non minimales.

Question 12
Écrire une fonction `consensus -> cnf -> cnf` telle que `consensus f` ajoute à la forme normale f tous les consensus entre ses clauses ; ici encore on s'assurera d'éliminer les clauses non minimales.

Exemple. La fonction ci-dessus appliquée à la forme normale f_0 donne

```
[(["c"], ["d"]); ([], ["c"; "m"]); (["d"], ["m"]);
 ([], ["m"; "d"]); (["c"], ["m"])]
```

V Recherche de contradictions

Théorème 1

1. La répétition de l'ajout de consensus avec élimination des non minimaux atteint un point fixe après un nombre fini d'étapes.
2. La forme normale est une contradiction si et seulement si l'étape finale ne contient qu'une clause vide.

Exemple. Si on ajoute à f_0 la clause m les différent appels de `consensus` donnent
$(\neg d \vee c) \wedge (\neg c \vee \neg m) \wedge (d \vee \neg m) \wedge m$
$(\neg d \vee c) \wedge m \wedge (\neg m \vee \neg d) \wedge (c \vee \neg m) \wedge \neg c \wedge d$
$m \wedge \neg c \wedge d \wedge \neg d \wedge c \wedge \neg d$ puis une clause vide.

Question 13
Écrire une fonction `contradiction f : fnc -> bool` qui teste si une forme normale est contradictoire.

Question 14
Le club écossais Les règles d'admission dans un club écossais sont les suivantes :

1. Tout membre non écossais porte des chaussettes rouges.
2. Tout membre qui porte des chaussettes rouges porte un kilt.
3. Les membres mariés ne sortent pas le dimanche.
4. Un membre sort le dimanche si et seulement s'il est écossais.
5. Tout membre qui porte un kilt est écossais et marié.
6. Tout membre écossais porte un kilt.

Montrer que les règles de ce club sont si contraignantes que personne ne peut en être membre.

Question 15 - En plus
Écrire une fonction qui donne les étapes qui aboutissent à une contradiction.

Solutions

Solution de la question 1

```
let rec mem x liste =
   match liste with
   |[] -> false
   |t::q -> (t = x) || (mem x q);;
```

Solution de la question 2

```
let rec except x liste =
   match liste with
   |[] -> []
   |t::q when t = x -> q
   |t::q -> t :: (except x q);;
```

Solution de la question 3

```
let rec union l1 l2 =
   match l1 with
   |[] -> l2
   |t::q when mem t l2 -> union q l2
   |t::q -> t :: (union q l2);;
```

Solution de la question 4

```
let rec union l1 l2 =
   match l1 with
   |[] -> []
   |t::q when mem t l2 -> t :: (inter q l2)
   |t::q -> inter q l2;;
```

Solution de la question 5

```
let rec moins l1 l2 =
   match l1 with
   |[] -> []
   |t::q when mem t l2 -> (moins q l2)
   |t::q -> t :: (moins q l2);;
```

Solution de la question 6

```
let triviale (c : clause) =
   let pos, neg = c in
   inter pos neg <> [];;
```

On peut aussi décomposer le couple dans la variable

```
let triviale ((pos, neg) : clause) =
   inter pos neg <> [];;
```

Solution de la question 7

```
let rec no_triviale (e : fnc) : fnc =
   match e with
   |[] -> []
   |c::ee when triviale c -> no_triviale ee
   |c::ee -> c :: (no_triviale ee);;
```

Solution de la question 8

```
let implique c1 c2 =
   let pos1, neg1 = c1 in
   let pos2, neg2 = c2 in
   moins pos1 pos2 = []  && moins neg1 neg2 = [];;
```

On peut définir cette fonction de manière infixe en donnant son nom entre parenthèses :

```
let (-->) c1 c2 =
```

On l'utilisera alors sous la forme `c1 --> c2`. Il y a de fortes contraintes[1] pour les noms possibles qui ne peuvent contenir que les caractères `$ & * + - / = > @ ^ % < . : = ?` $\sim$.
Les 5 derniers ne peuvent pas apparaître en première position.

Solution de la question 9

```
let rec ajoute (c : clause) (f : fnc) : fnc -
   match f with
    |[] -> [c]
    |c1::f1 when implique c1 c -> f
    |c1::f1 when implique c c1 -> ajoute c f1
    |c1::f1 -> c1 :: (ajoute c f1);;
```

Solution de la question 10

```
let minimise (f : fnc) : fnc =
   let rec aux reste fait =
      match reste with
      |[] -> fait
      |c::f1 -> aux f1 (ajoute c reste) in
   aux f [];;
```

Solution de la question 11

```
let ajoute_cons (p1, n1 : clause) (p2, n2 : clause) (f : cnf) :
   cnf =
   match (inter p1 n2),  (inter n1 p2) with
   |[x], [] -> let cs = (union (except x p1) p2, union n1 (except
      x n2)) in
               ajoute cs f
   |[], [x] -> let cs = (except p1 (moins x p2), union (except x
      n1) n2) in
               ajoute cs f
   |_ -> f;;
```

1. voir https ://ocaml.org/manual/lex.html

Solution de la question 12

```
let consensus f =
   let rec aux en_cours reste fait =
      match en_cours, reste with
      |_, [] -> fait
      |[], c::ee -> aux ee ee fait
      |c1::e1, c2::e2 -> aux e1 en_cours (ajoute_cons c1 c2 fait)
   in aux e e e;;
```

Solution de la question 13
Les clauses qui restent sont inchangées donc on peut tester l'égalité de deux formes normales par une double différence (les clauses peuvent changer de place.

```
let rec contradiction f =
   let f1 = consensus f in
   if moins f f1 = [] && moins f1 f = []
   then f = [([], [])]
   else contradiction f1;;
```

Solution de la question 14
`e` pour membre écossais, `r` pour membre portant des chaussettes rouges, `k` pour membre portant un kilt, `m` pour membre marié, `d` pour membre sortant le dimanche. On doit avoir $(\neg e \Rightarrow r) \wedge (r \Rightarrow k) \wedge (m \Rightarrow \neg d) \wedge (d \Leftrightarrow e) \wedge (k \Rightarrow e \wedge m) \wedge (e \Rightarrow k)$. D'où la `fnc`

```
[["e"; "r"], []; ["k"], ["r"]; [], ["m"; "d"]; ["e"], ["d"];
 ["d"], ["e"]; ["e"], ["k"]; ["m"], ["k"]; ["k"], ["e"]]
```

La fonction `contradiction` donne `true`.

Solution de la question 15 - En plus
On a joute aux clauses leur origine : hypothèse ou consensus.

```
type clause = (string list) * (string list) * raison
     and raison  = Hyp | Cons of clause * clause;;
```

On transforme les fonctions `implique` et `ajoute_cons`

```
let implique c1 c2 =
   let pos1, neg1, _ = c1 in
   let pos2, neg2,_ = c2 in
   moins pos1 pos2 = []  && moins neg1 neg2 = [];;
```

```
let ajoute_cons (c1 : clause) (c2 : clause) (f : fnc) : fnc =
   let p1, n1, r1 = c1 in
   let p2, n2, r2 = c2 in
   match (inter p1 n2),  (inter n1 p2) with
   |[x], [] -> let cs = (union (except x p1) p2,
                         union n1 (except x n2),
                         Cons (c1, c2)) in
                        ajoute cs f
   |[], [x] -> let cs = (union (except x p2) p1,
                         union n2 (except x n1),
                         Cons (c1, c2)) in
                        ajoute cs f
   |_ -> f;;
```

On écrit les clause sous forme d'implications

```
let rec afficher p n =
   match p, n with
   |[], [] -> print_string "faux"
   |p, [] -> print_string (String.concat " | " p)
   |[], n -> print_string "non (";
             print_string (String.concat " & " n);
             print_string ")"
             |p, n -> print_string "(";
             print_string (String.concat " & " n);
             print_string ") => (";
             print_string (String.concat " | " p);
             print_string ")";;
```

On définit un raisonnement

```
let rec rais r =
   match r with
   |(p, n, Hyp) -> ()
   |(p, n, Cons ((p1, n1, r1), (p2, n2, r2)))
      -> rais (p1, n1, r1);
         rais (p2, n2, r2);
         print_string "De ";
         afficher p1 n1;
         print_string " et ";
         afficher p2 n2;
         print_string " on déduit ";
         afficher p n;
print_newline();;
```

On modifie `contradiction`

```
let rec contradiction f =
   let f1 = consensus f in
   if moins f f1 = [] && moins f1 f = []
   then match f with
      |[([], [], r)] -> rais ([], [], r)
      |_ ->  print_string "satisfiable"
   else contradiction f1;;
```

```
De (e) => (k) et (k) => (m) on déduit (e) => (m)
De (e) => (d) et non (m & d) on déduit non (e & m)
De (e) => (m) et non (e & m) on déduit non (e)
De (k) => (e) et (r) => (k) on déduit (r) => (e)
De (r) => (e) et e | r on déduit e
De non (e) et e on déduit faux
```