TP noté de
Programmation impérative et fonctionnelle en OCaml

Durée: 2h
Tous documents autorisés.
Le barème est donné à titre indicatif.

20 novembre 2023

Consignes :
— Les données **mutables** (e.g. référence, tableau...) et les boucles **for** et **while** sont **interdites**.
— Déposer votre fichier de code source avant 10:00 sur e-campus.

Transformation de formules logiques

Le problème de satisfaisabilité d'une formule logique (SAT) est le premier à avoir été démontré NP-complet par Stephen Cook en 1971 :

> Soit une formule logique en **forme normale conjonctive** (CNF) sur un ensemble $U = \{u_1, \dots, u_n\}$ de n variables logiques, existe-t-il une affectation des variables, i.e. une fonction $U \mapsto \{\text{false}, \text{true}\}$, telle que la formule soit vraie ?

Une formule logique est en CNF ssi elle est une **conjonction** ($\wedge$) de m *clauses* c_j, une clause étant une **disjonction** ($\vee$) de *littéraux* l_j^r, et un littéral étant une variable logique $u_i \in U$ ou sa négation $\neg u_i$:

$$\bigwedge_{j=1}^{m} c_j$$
$$\text{avec } \forall j \in [1..m],\ c_j = l_j^1 \vee \cdots \vee l_j^{k_j}$$
$$\text{et } \forall r \in [1..k_j],\ \exists u_i \in U \text{ t.q. } l_j^r = u_i \text{ ou } l_j^r = \neg u_i$$

Par exemple, la formule suivante sur $U = \{u_1, u_2, u_3, u_4\}$ est en CNF :

$$(u_1 \vee \neg u_2 \vee u_3) \wedge (u_2 \vee \neg u_4) \wedge (\neg u_1) \tag{1}$$

avec 3 clauses $c_1 = u_1 \vee \neg u_2 \vee u_3$, $c_2 = u_2 \vee \neg u_4$ et $c_3 = \neg u_1$. Cette seconde formule n'est pas en CNF :

$$\neg(u_1 \vee (u_2 \wedge \neg u_3)) \vee (u_1 \wedge \neg(u_2 \wedge \neg u_4)) \tag{2}$$

en effet, il ne peut y avoir de négation ($\neg$) que devant une variable et il ne peut y avoir que des disjonctions ($\vee$) au sein d'une même clause.

Il existe des solveurs SAT très efficaces, mais il faut leur fournir la formule à résoudre en CNF. Nous allons écrire un programme fonctionnel en OCaml pour transformer une formule logique quelconque en formule CNF.

1. [2pt] Définir le type `t` des formules logiques quelconques qui peuvent être :
   — une variable logique identifiée par une chaîne de caractères ;
   — la négation d'une formule logique ;
   — la disjonction de deux formules logiques ;
   — la conjonction de deux formules logiques.
2. [1pt] Représenter les formules (1) et (2) avec le type `t`.
3. [2pt] Pour représenter une affectation des variables logiques, définir un type enregistrement `assignment` contenant un champ de type `string` correspondant à l'identifiant d'une variable et un champ de type `bool` correspondant à sa valeur. L'affectation de plusieurs variables sera représentée par une liste de type `assignment list`.

Écrire la fonction `get: string -> assignment list -> bool` telle que `get id env` renvoie la valeur de la variable d'identifiant `id` dans l'affectation `env` ou lève l'exception `Not_found` si elle ne s'y trouve pas.

4. [2pt] Écrire une fonction `eval: assignment list -> t -> bool` telle que `eval env f` renvoie la valeur booléenne de la formule logique `f` avec l'affectation `env`. Avec l'affectation $u_1 \leftarrow$ `false`, $u_2 \leftarrow$ `false`, $u_3 \leftarrow$ `false`, $u_4 \leftarrow$ `true`, cette fonction doit renvoyer `false` pour la formule (1) et `true` pour la formule (2).
5. [3pt] Pour transformer une formule quelconque en CNF, on va commencer par faire « descendre » les négations ($\neg$) récursivement jusqu'aux variables pour obtenir une formule en **forme normale négative** (NNF) où les négations ne peuvent apparaître que devant une variable. Écrire une fonction `nnf: t -> t` qui prend une formule logique quelconque en paramètre et la transfome en formule en NNF en appliquant les transformations suivantes (*double négation* (3) et *lois de de Morgan* (4) et (5)) :

$$\mathrm{nnf}(\neg\neg p) \mapsto \mathrm{nnf}(p) \tag{3}$$
$$\mathrm{nnf}(\neg(p \vee q)) \mapsto \mathrm{nnf}(\neg p) \wedge \mathrm{nnf}(\neg q) \tag{4}$$
$$\mathrm{nnf}(\neg(p \wedge q)) \mapsto \mathrm{nnf}(\neg p) \vee \mathrm{nnf}(\neg q) \tag{5}$$

avec p et q deux formules logiques. Sur la formule (2), cette transformation doit renvoyer la formule en NNF :

$$(\neg u_1 \wedge (\neg u_2 \vee u_3)) \vee (u_1 \wedge (\neg u_2 \vee u_4)) \tag{6}$$

Indication : ne pas oublier d'appliquer également récursivement la transformation sur les sous-formules de tous les cas récursifs non couverts par les règles (3) *à* (5)*, ainsi que de traiter les cas d'arrêt de la récursion.*

6. [4pt] À partir d'une formule en NNF, il faut ensuite faire descendre récursivement les disjonctions ($\vee$) jusqu'au littéraux (variable ou négation de variable) par l'application de la règle de distributivité :

$$\mathrm{cnf}(p \vee q) \mapsto \bigwedge_{i\in[1..k],j\in[1..l]} p_i \vee q_j \tag{7}$$
$$\text{avec } \mathrm{cnf}(p) = p_1 \wedge \cdots \wedge p_k \text{ et } \mathrm{cnf}(q) = q_1 \wedge \cdots \wedge q_l \tag{8}$$

avec p et q des formules logiques en NNF, et $\mathrm{cnf}(p)$ et $\mathrm{cnf}(q)$ des formules en CNF. Les formules p_i et q_j sont donc des clauses (des disjonctions), par conséquent les formules $p_i \vee q_j$ également et la règle produit bien une formule en CNF.

Pour pouvoir appliquer cette règle, on va simplifier (« aplatir ») la représentation des formules en CNF en représentant :

— un littéral par une couple de type `(string * bool)` : e.g. u_1 sera représenté par `("u1",true)` et $\neg u_2$ par `("u2",false)` ;
— une clause (disjonction de littéraux $l_1 \vee \cdots \vee l_k$) par une liste de littéraux de type `(string * bool) list` : e.g. $u_1 \vee \neg u_2$ devient `[("u1",true);("u2",false)]`.
— une formule en CNF par une liste de clauses de type `(string * bool) list list`.

La formule (1) sera donc représentée par :

```
[[("u1",true);("u2",false);("u3",true)];[("u2",true);("u4",false)];[("u1",false)]]
```

On va écrire une fonction `n2cnf: t -> (string * bool) list list` qui prend en paramètre une formule en NNF et renvoie une formule en CNF sous la représentation simplifiée, en distribuant les disjonctions ($\vee$) avec la règle (7), ainsi qu'en transformant récursivement les conjonctions ($\wedge$) en CNF selon la règle :

$$\mathrm{n2cnf}(p \wedge q) \mapsto \mathrm{n2cnf}(p) \wedge \mathrm{n2cnf}(q) \tag{9}$$

avec p et q des formules en NNF, et $\mathrm{n2cnf}(p)$ et $\mathrm{n2cnf}(q)$ des formules en CNF, donc la règle produit bien une formule en CNF. Sur la formule (6) en NNF, cette transformation doit renvoyer la formule en CNF :

$$(\neg u_1 \vee u_1) \wedge (\neg u_1 \vee \neg u_2 \vee u_4) \wedge (\neg u_2 \vee u_3 \vee u_1) \wedge (\neg u_2 \vee u_3 \vee \neg u_2 \vee u_4) \tag{10}$$

Indications :

— *Ne pas oublier les cas d'arrêts pour les littéraux positifs et négatifs qui doivent renvoyer une formule en CNF d'une seule clause contenant un seul littéral, i.e. une liste de liste d'un seul élément.*
— *Pour construire la conjonction de deux formules en CNF, il suffit de concaténer les listes qui les représentent.*
— *Idem pour construire la disjonction de deux clauses ($p_i \vee q_j$).*
— *Pour appliquer la règle* (7)*, utiliser la fonction ci-dessous*
`prod: ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`
telle que `prod h p q` *renvoie la liste des applications de la fonction* `h` *au produit cartésien des éléments des listes* `p` *et* `q`. E.g. *avec* `p = [p1; p2]` *et* `q = [q1; q2]`, `prod h p1 p2` *renvoie* `[h p1 q1; h p1 q2; h p2 q1; h p2 q2]`.

```
let prod = fun h p q ->
  List.fold_right
    (fun pi pr -> List.fold_right (fun qj qr -> h pi qj :: qr) q pr)
    p []
```

7. [5pt] Lors de la transformation en CNF, une variable peut apparaître plusieurs fois dans la même clause, avec ou sans négation, comme dans l'exemple de la question précédente. Il faut donc simplifier chaque clause en ne gardant qu'une seule occurence des littéraux identiques, puis en supprimant les clauses qui contiennent une variable et sa négation (elles sont trivialement vraies). Pour simplifier une clause :

   (a) on commence par trier (par ordre lexicographique sur les deux éléments du couple, i.e. avec l'ordre naturel de la fonction `compare`) la liste qui la représente pour que les *littéraux identiques* soient consécutifs, e.g. :
   `[("u1",false);("u1",false);("u1",true);("u2",false);("u4",true);("u4",true)]`

   (b) puis on parcourt (une seule fois) cette liste triée pour supprimer les doublons en ne gardant qu'une seule occurrence de chaque littéral, e.g.
   `[("u1",false);("u1",true);("u2",false);("u4",true)]`

   (c) enfin, on parcourt cette liste de littéraux uniques pour vérifier s'il y a deux littéraux consécutifs de signes opposés (comme dans l'exemple précédent).

   Écrire la fonction `unique: 'a list -> 'a list` qui parcourt une liste en supprimant tous les doublons après l'avoir triée (étapes 7a et 7b).

   Écrire ensuite une fonction `check_opposite: ('a * 'b) list -> bool` qui parcourt (une seule fois) la liste résultat de la fonction précédente et renvoie `true` si la même variable apparaît avec des signes opposés dans la clause, et `false` sinon (étape 7c).

   À l'aide des deux fonctions précédentes, écrire une fonction `simplify: ('a * 'b) list -> ('a * 'b) list` qui prend une clause CNF en paramètre et renvoie une liste vide si elle contient deux littéraux opposés, ou une clause équivalente sans occurrence multiple de variable sinon.

   Enfin, écrire une nouvelle version `scnf` de la fonction `cnf` en appliquant `simplify` à la concaténation des littéraux de p_i et q_j (en modifiant la fonction `h` prise en paramètre par `prod`) lorsque la règle de distributivité (7) doit être utilisée. Pour supprimer les clauses vides, on filtrera également le résultat de `prod` avec la fonction `List.filter`[1] (sans calculer la taille de la liste).

   Sur la formule (6) en NNF, la transformation en CNF doit maintenant produire :

$$(\neg u_1 \vee \neg u_2 \vee u_4) \wedge (u_1 \vee \neg u_2 \vee u_3) \wedge (\neg u_2 \vee u_3 \vee u_4) \tag{11}$$

8. [1pt] Écrire une fonction `cnf: t -> (string * bool) list list` qui transforme une formule logique quelconque en formule CNF simplifiée en combinant les étapes précédentes. On appliquera également la fonction `unique` de la question précédente au résultat final pour supprimer d'éventuelles clauses identiques.

1. `List.filter: ('a -> bool) -> 'a list -> 'a list`
`List.filter p l` renvoie la liste des éléments `x` de `l` tels que `p x` renvoie `true`.

9. [Bonus] Certaines formules sont également écrites à l'aide des connecteurs logiques d'implication ($\Rightarrow$), d'équivalence ($\Leftrightarrow$) et de disjonction exclusive ($\oplus$) qu'il faut éliminer récursivement en les remplaçant par $\vee$, $\wedge$ et $\neg$ selon les règles suivantes :

$$p \Rightarrow q \mapsto \neg p \vee q \tag{12}$$
$$p \Leftrightarrow q \mapsto (\neg p \vee q) \wedge (p \vee \neg q) \tag{13}$$
$$p \oplus q \mapsto (p \vee q) \wedge (\neg p \vee \neg q) \tag{14}$$

Enrichir le type `t` de ces nouveaux connecteurs logiques, puis écrire une fonction qui les supprime à l'aide des règles précédentes avant de l'appliquer préalablement à la transformation en NNF au sein de la fonction `cnf`.