TP 8. Logique : syntaxe et sémantique

I Syntaxe des formules de la logique propositionnelle en OCaml

Dans tout le sujet, l'ensemble des variables propositionnelles $\mathcal{V}$ est $\{x_0, x_1, \cdots\} = \{x_i : i \in \mathbb{N}\}$.

I.1 Représentation

on définit le type suivant pour représenter les formules en OCaml :

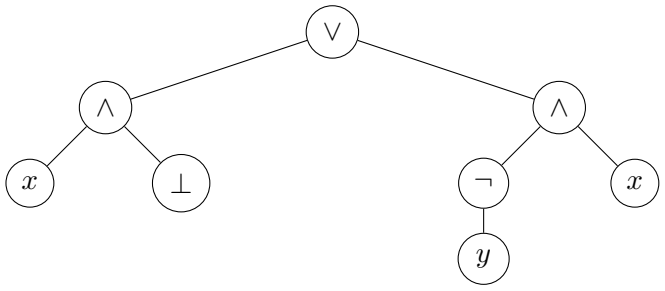
```
type formule =
  | Vrai | Faux                 (* constantes True et Bottom *)
  | Var of int                  (* représente la variable x_i de numéro i >= 1 *)
  | Non of formule              (* Non(P) représente  P *)
  | Et of formule * formule     (* Et(P,Q) représente P ^ Q *)
  | Ou of formule * formule     (* Ou(P,Q) représente P v Q *)
;;
```

Les lignes correspondantes sont déjà écrites dans le fichier tp08.ml qui accompagne cet énoncé.

Question 1. Définir au moins quatre constantes `x0 = Var(0)` $\cdots$, `x3 = Var(3)` , pour facilement écrire des formules dépendant de ces quelques variables.

Imaginons que l'on veuille représenter en OCaml la formule P représentée dans la figure ci-dessous (sous forme de son arbre syntaxique).

Question 2. Quelles variables parmi `x0` , $\cdots$ `x3` allez-vous choisir pour représenter les variables x et y ? Définissez les variables correspondantes, puis la formule `formule_P` (de type formule) la représentant.



I.2 Premières fonctions : hauteur et taille d'une formule

Question 3. Écrire deux fonctions `hauteur` de type formule -> int et taille de type formule -> int calculant la hauteur et la taille d'une formule (attention aux définitions !).

Question 4. Tester-les sur la formule `formule_P` .

Question 5. Quelles sont les complexités temporelles de ces deux fonctions, en fonction des valeurs de $h(P)$ et/ou $t(P)$?

I.3 Ensemble des variables d'une formule

On définit pour une formule P , l'ensemble (fini) $\mathcal{V}(P)$ comme l'ensemble des variables $x \in \mathcal{V}$ qui apparaissent dans P .

On va représenter un tel ensemble par l'idée la plus simple : une `int list` donnant les indices des x_i apparaissant dans P , dans un ordre quelconque.

Question 6. Pour la formule P de l'exemple, donner $\mathcal{V}(P)$ ainsi que sa représentation en OCaml .

Question 7. Sans chercher à renvoyer une `int list` qui n'aurait pas de doublons, écrire une fonction `variables_dans` de type formule -> int list qui renvoie tous les indices des variables x_i apparaissant dans une formule donnée. Tester-la sur `formule_P` . On peut obtenir par exemple $[0; 1; 0]$ (l'ordre n'importe pas).

I.4 Affichage préfixe (quasi sérialisation)

On cherche à afficher une formule sous la forme suivante, qui permet de la redéfinir si on le copie-colle dans une console ou un fichier OCaml :

```
# affiche_prefixe formule_P;;
Ou (Et (Var 0, Faux), Et (Non (Var 1), Var 0))
- : unit = ()
```

Question 8. Écrire une fonction `repr_prefixe` de type formule -> string qui construit une chaîne de caractères représentant en notation préfixe la formule donnée en argument. On rappelle que l'on peut concaténer deux chaînes avec `chaine1 ^ chaine2` (accent circonflexe).

Question 9. En déduire une fonction `affiche_prefixe` de type formule -> unit qui affiche la formule sur une ligne à part. Tester-la sur `formule_P` .

I.5 Affichage infixe

On cherche à afficher une formule sous la forme infixe, qui correspond à la notation "naturelle" que l'on utilise généralement.

Question 10. Donner (au brouillon) la notation infixe de la formule P de l'exemple.

Question 11. Écrire une fonction (récursive) `repr_infixe` de type formule -> string qui construit une chaîne de caractères représentant en notation infixe la formule donnée en argument.

Question 12. En déduire une fonction `affiche_infixe` de type `formule -> unit` qui affiche la formule sur une ligne à part. Tester-la sur `formule_P` .

I.6 Substitutions $P\,[Q/x]$

La substitution $P\,[Q/x]$ est définie par induction structurelle sur la formule P. L'idée est simple : on propage la substitution à l'intérieur de l'arbre syntaxique sans modifier les connecteurs unaires et binaires (e.g. $(P_1 \wedge P_2)\,[Q/x] = P_1\,[Q/x] \wedge P_2\,[Q/x]$) et aux feuilles, et on ne touche ni les constantes ni les variables $y \neq x$, mais on remplace les occurrences de la variable x par la formule Q.

Question 13. Soit $Q = x_2 \wedge (x_3 \vee \neg x_2)$. Donner au brouillon la formule $P\,[Q/x_0]$.

Question 14. Écrire une fonction `substitution` de type formule -> int -> formule -> formule telle que `substitution formule_P i formule_Q` donne la formule $P\,[Q/x_i]$.

Question 15. Vérifier le résultat donné en Q13.

II Sémantique des formules en OCaml

Question 16. Écrire les trois fonctions suivantes : `non : bool -> bool` , `et : bool -> bool -> bool` , `ou : bool -> bool -> bool` correspondant aux trois opérateurs logiques habituels.

II.1 Valuations v : donner une valeur de vérité aux variables

Comme on a fixé l'ensemble $\mathcal{V}$ des variables à $\{x_i : i \in \mathbb{N}\}$, on peut représenter une valuation $v : \mathcal{V} \to \mathcal{B}$ par une fonction de type int -> bool , ou un tableau bool array , indicé de 0 à $n-1$ où l'indice i (pour $0 \leqslant i \leqslant n-1$) est utilisé pour stocker la valeur booléenne de $v(x_i) \in \mathcal{B} = \{\mathbf{false}, \mathbf{true}\}$, $n-1$ étant l'indice de la plus grande variable utilisée dans la formule sur laquelle on veut faire agir la valuation.

Question 17. Donner au brouillon une valuation v^+ telle que $[\![P]\!]_{v^+} = \mathbf{true}$. De même pour v^- telle que $[\![P]\!]_{v^-} = \mathbf{false}$.

Question 18. Définir des fonctions `v_plus` et `v_moins` de type int -> bool , représentant en OCaml ces deux valuations.

Question 19. Même question avec la représentation sous forme de tableau : définir deux variables de type bool array `v_plus_tab` et `v_moins_tab` représentant ces deux valuations.

II.2 Interprétation d'une formule

Pour une valuation $v : \mathcal{V} \to \mathcal{B}$ fixée, on définit la fonction d'interprétation, qui calcule la valeur de vérité d'une formule P, par induction structurelle sur P (cf. cours) .

Pour la suite afin de faciliter le travail on utilise la description des valuations par l'utilisation de tableaux de booléens.

Question 20. Écrire une fonction `evaluation` de type bool array -> formule -> bool telle que `evaluation v formule_P` calcule $[\![P]\!]_v$.

Question 21. Quelle est sa complexité temporelle, en fonction de $h(P)$ ou de $t(P)$?

II.3 Affichage d'une table de vérité

On souhaite pouvoir afficher une table de vérité d'une formule, sous la forme suivante (par exemple) :

```
| x1 | x2 | x1 v x2
| false | false | false
| false | true | true
| true | false | true
| true | true | true
```

Question 22. On veut écrire une fonction `toutes_les_valuations` de type int -> bool list list qui donne (sous forme de bool list) toutes les valuations possibles de n variables.

Pour faciliter le travail, on commence par générer une liste de listes de booléens. On fera la conversion en tableau dans la question suivante.

Voici ce que devrait donner la fonction pour $n = 0, 1, 2$ et 3 :

```
toutes_les_valuations 0 = []
toutes_les_valuations 1 = [[false]; [true]]
toutes_les_valuations 2 = [
[false; false]; [false; true];
[true; false]; [true; true]
]
toutes_les_valuations 3 = [
[false; false; false]; [false; false; true];
```

```
[false; true; false]; [false; true; true];
[true; false; false]; [true; false; true];
[true; true; false]; [true; true; true]
]
```

Je vous laisse trouver un algorithme pour cette fonction. D'un point de vue technique ce sera une très bonne occasion d'utiliser la fonction `List.map` !

Cet algorithme est évidemment très (trop !) coûteux quand n devient grand, car il y a 2^n valuations à renvoyer !

Question 23. En déduire une fonction `affiche_table_verite` de type formule -> unit qui affiche la table de vérité au format décrit ci-dessus. On pourra utiliser `List.iter` pour itérer sur la liste des valuations donnée par `toutes_les_valuations` . Vous réutiliserez bien sûr des fonctions précédentes mais il y a des "pièges"...

Rappel : en utilisant `Array.of_list` de type 'a list -> 'a array , on peut convertir un tableau en une liste (dans le même ordre).

Question 24. Tester-la pour `formule_P` de l'exemple, on devrait obtenir cela

```
| x0 | x1 | ((x0 ^ Faux) v (n x1 ^ x0))
| false | false | false
| false | true | false
| true | false | true
| true | true | false
```

II.4 Nature d'une formule : tautologique, satisfiable ou insatisfiable

Question 25. Proposer un type énumération `nature_formule` en OCaml pour représenter la nature d'une formule, qui peut être Tautologique, Satisfiable (signifiant qu'elle est satisfiable mais pas tautologique), ou Insatisfiable.

II.5 Résolution naïve (exploration exhaustive) pour le problème SAT

Question 26. En utilisant un code assez similaire à celui de la fonction `affiche_table_verite` , écrire une fonction `valuations_sat` de type formule -> bool list list qui calcule toutes les valuations qui évaluent P à true.

Question 27. Définir une fonction `deux_puissance` de type int -> int qui calcule 2^n. Note : on peut aussi astucieusement utiliser `1 lsl n` qui calcule la même chose ! `m lsl n` décale les bits de l'écriture binaire de l'entier m de n crans vers la droite, ce qui revient à multiplier par 2^n (modulo le plus gros entier représentable)

Question 28. En déduire une fonction `calcule_nature` de type formule -> nature_formule qui calcule le nombre de valuations v satisfaisant P et en déduit sa nature.

II.6 Satisfiabilité et validité par l'algorithme de Quine

Dans cette partie on s'intéresse à nouveau au problème de la satisfiabilité et de la validité d'une formule, mais on essaye d'éviter l'énumération des valuations en procédant par substitution et simplifications. Plus précisément, pour la satisfiabilité d'une formule A par exemple, il s'agit de substituer à une variable x apparaissant dans la formule, la formule T, de tester (récursivement) si la formule obtenue est satisfiable, auquel cas une valuation satisfaisante prolongée par $x \mapsto V$ satisfait A, et dans le cas contraire de tester (récursivement) si la formule obtenue en substituant à x la formule $\bot$ est satisfiable, auquel cas une valuation satisfaisante prolongée par $x \mapsto F$ satisfait A, et enfin dans le cas contraire on conclut que la formule n'est pas satisfiable.

Dans cette partie pour faciliter le travail les valuations seront représentées par une liste de couples de type string * bool , qui associent à chaque variable sa valeur de vérité. On considère donc la définition de type suivante.

`type valuation = (string * bool) list`

On suppose qu'une valuation contient uniquement des couples dont les premières composantes sont 2 à 2 disjointes. Il faudra veiller à maintenir cette propriété. L'ordre des associations dans la liste est quelconque et non significatif.

On réutilisera si nécessaire et quand cela est possible les fonctions déjà définies auparavant. Si nécessire on adapatera des fonctions déjà existantes, et enfin On écrira toute fonction nécessaire à la résolution (sans faire une usine à gaz SVP...)

Question 28. Définir une fonction simpl qui simplifie une formule en appliquant là où elle le peut les règles de simplifications mentionnées en cours (à prendre parmi les équivalences classiques présentées en cours). Par exemple

```
# simpl (Et (Ou (Vrai, Var 1), Ou (Vrai, Var 2)));;
- : formule = Et (Vrai, Vrai)
# simpl (simpl (Et (Ou (Vrai, Var 1), Ou (Vrai, Var 2))));;
- : formule = Vrai
# simpl (simpl (simpl (Et (Ou (Vrai, Var 1), Ou (Vrai, Var 2)))));;
- : formule = Vrai
```

Question 29. Définir une fonction `simpl_complet` qui itère la fonction `simpl` jusqu'à ce que celle-ci n'ait plus d'effet. Par exemple, `simpl_complet (Et (Ou (Vrai, Var 1), Ou (Vrai, Var 2)))` calcule `Vrai` .

Il existe un type prédéfini 'a option qui propose deux constructeurs. Le premier `None` est un constructeur constant qui représente l'absence de valeur, et le second `Some of 'a` qui représente une valeur de type 'a . Ce type permet donc à des fonctions qui doivent calculer une valeur de ne rien renvoyer en cas d'échec (la fonction renvoie `None`) ou une valeur en cas de réussite par l'intermédiaire de `Some (quelquechose)` ! Pratique ! On va s'en servir dans la fonction suivante.

Question 30. Définir une fonction `est_sat_quine` de type formule -> valuation option qui calcule une valuation satisfaisant la formule prise en argument s'il en existe.

Question 31. Définir une fonction `est_non_sat_quine` de type formule -> valuation option qui calcule quant à elle un environnement ne satisfaisant pas la formule prise en argument s'il en existe.

II.7 Conséquence logique et équivalence logique

On rappelle que $P \vDash Q$, qui se lit "Q est conséquence logique de P", signifie que toute valuation v qui satisfait P satisfait aussi Q. Autrement dit, $P \to Q$ (l'implication logique) est une tautologie (ce que l'on peut noter $\vDash P \to Q$).

Question 29. Écrire une fonction `est_consequence_logique` de type formule -> formule -> bool qui calcule si $P \vDash Q$ ou non.

Question 30. En déduire une fonction `sont_equivalents` de type formule -> formule -> bool qui calcule si $P \equiv Q$ (on rappelle que cela signifie $P \vDash Q$ et $Q \vDash P$).