TP n°4 - Correction

Logique propositionnelle en OCAML

Dans les exercices ci-dessous, les fonctions du module `List` sont librement utilisables (`List.map`, `List.exists`, `List.for_all`...)

1 Calcul propositionnel

On considère les formules de la logique propositionnelle construites avec des variables propositionnelles et les connecteurs $\neg$, $\wedge$ et $\vee$. Pour représenter ces formules en Ocaml, on se servira du type suivant :

```
type formule  = Var of string
              | Neg of formule
              | Et of formule * formule
              | Ou of formule * formule;;
```

Par exemple, la formule $p \wedge \neg(q \vee r)$ sera représentée par :

```
Et (Var "p", Neg(Ou(Var "q",Var "r")));;
```

Exercice 1. Écrire une fonction

`string_of_formule : formule -> string`

qui, étant donnée une formule, renvoie une chaîne de caractères qui représente cette formule écrite avec des parenthèses encadrant toutes les sous-formules de connecteur binaire. Par exemple, pour la formule ci-dessus, on obtiendra `"(p Et Neg (q Ou r))"`.

Correction :

```
let rec string_of_formule f = match f with
     Var s -> s
   | Neg f -> "Neg"^(string_of_formule f)
   | Et(f1,f2) -> "("^(string_of_formule f1)^" Et "^(string_of_formule f2)^")"
   | Ou(f1,f2) -> "("^(string_of_formule f1)^" Ou "^(string_of_formule f2)^")"
;;
```

Exercice 2. Écrire une fonction

`list_of_vars : formule -> string list`

qui, étant donné une formule, renvoie une liste sans répétitions de tous ses noms de variables. Par exemple, `list_of_vars (Et (Ou(Var "p",Var "q"), Var "q"))` renverra `["p"; "q"]` à l'ordre des éléments près.

Pour construire l'union sans répétitions des noms de variables de deux sous-formules, vous pouvez vous servir de la fonction `union_sorted` du TP 2 – ou écrire toute autre fonction auxiliaire produisant cette union.

Correction :

```
let rec union_sorted l1 l2 = match l1,l2 with
      _,[] -> l1
    | [],_ -> l2
    | a1::l1',a2::l2' ->
        if a1 < a2 then a1::(union_sorted l1' l2 ) else
        if a2 < a1 then a2::(union_sorted l1  l2') else
        a1::(union_sorted l1' l2')
;;
let rec list_of_vars f = match f with
      Var s -> [s]
    | Neg f -> list_of_vars f
    | Et(f1,f2)
    | Ou(f1,f2) ->
          union_sorted (list_of_vars f1) (list_of_vars f2)
;;
```

2 Evaluation des formules

Appelons *affectation* toute liste d'association de type `(string * bool) list`. Une affectation permet de spécifier les valeurs de vérité des variables d'une formule. Chaque variable apparaît une fois au plus dans une affectation.

Par exemple, une affectation possible pour la formule `Et (Var "p", Var "q")` est `[("p", true), ("q", false)]`.

La *valeur de vérité d'une formule* `f` *relativement à une affectation* `e` est :
- indéfinie, si les variables apparaissant dans `f` n'apparaissent pas toutes dans `e`,
- sinon, cette valeur de vérité est obtenue en remplaçant dans `f` :
  - les variables par les valeurs spécifiées par `e`,
  - les connecteurs logiques par les opérations qui leur sont naturellement associées.

Exercice 3. Écrire une fonction

```
eval_formule : formule -> (string * bool) list -> bool
```

calculant la valeur de vérité d'une formule relativement à une affectation donné. Par exemple, `eval_formule (Et (Var "p", Var "q"))[("p", true); ("q", false)]` donnera **`false`**.

Vous pouvez utiliser `List.assoc v e` qui renvoie la partie droite `b` du premier couple de `e` de la forme `(v, b)` s'il existe, et déclenche une exception si ce couple est introuvable.

Correction :

```
let rec eval_formule f l = match f with
    Var s -> (List.assoc s l)
  | Neg f1 ->  not (eval_formule f1 l)
  | Et(f1,f2) -> (eval_formule f1 l) && (eval_formule f2 l)
  | Ou(f1,f2) -> (eval_formule f1 l) || (eval_formule f2 l)
;;
```

3 Formules satisfiables et tautologies

Une formule de la logique propositionnelle est appelée une *tautologie*, si elle s'évalue à **`true`** pour tout affectation spécifiant les valeurs de vérité de toutes ses variables. Une formule qui s'évalue à **`true`** pour *au moins* une affectation est dite *satisfiable*.

Exercice 4. Ecrire une fonction polymorphe

```
add_to_all : 'a -> ('a list) list  -> ('a list) list
```

telle que `add_to_all x [l1; ... ; ln]` renvoie la liste de listes `[x::l1; ... ; x::ln]`. Noter que cette fonction est très simple à écrire avec `List.map`.

Exercice 5. En vous servant de la fonction `add_to_all`, écrire

```
affectations_vars : string list  -> ((string * bool) list) list
```

qui, étant donnée une liste de noms de variables, construit liste de tous les affectations possibles associés. Par exemple, `affectations_vars ["q"]` donnera

```
[[("q", false)];
 [("q", true) ]]
```

et `affectations_vars ["p"; "q"]` donnera

```
[[("p", false); ("q", false)];
 [("p", false); ("q", true) ];
 [("p", true);  ("q", false)];
 [("p", true);  ("q", true) ]]
```

En déduire une fonction

`affectations : formule  -> (string * bool) list list`

qui étant donnée une formule donne une liste de tous les affectations associés à ses variables.
Correction :

```
let add_to_all x ll = List.map (fun l -> x::l) ll
;;
let rec affectations_vars l = match l with
    [] -> [[]]
  | v::l' ->
      let le = affectations_vars l' in
      (add_to_all (v, false) le)@
      (add_to_all (v, true)  le)
;;
let affectations f =
   affectations_vars (list_of_vars f)
;;
```

Exercice 6. A l'aide de la fonction `affectations`, de `List.exists` et de `List.for_all`, écrire

`satisfaisable: formule -> bool`

qui, étant donnée une formule, détermine si elle est satisfaisable, et

`tautologie : formule -> bool`

qui, étant donnée une formule, détermine si elle est une tautologie. Par exemple,
- `satisfaisable (Et (Var "p", Neg(Var "p")))` donnera **`false`**.
- `tautologie (Ou(Var "p", Neg(Var "p")))` donnera **`true`**.

Correction :

```
let satisfaisable f =
  let le = affectations f in
  List.exists (eval_formule f) le;;

let tautologie f =
  let le = affectations f in
  List.for_all (eval_formule f) le;;
```

4 Conséquences logiques et équivalences.

Si `f` et `g` sont deux formules de la logique propositionnelle, on dit que `g` *est conséquence logique de* `f` si, pour toute affectation pour laquelle `f` s'évalue à **`true`**, `g` s'évalue aussi à **`true`** . Les affectations considérées doivent spécifier à la fois les valeurs des variables de `f` et de celles de `g` – ces deux ensembles de variables peuvent être joints par exemple à l'aide de la fonction `union_sorted`. Si `g` est conséquence logique de `f` et `f` est conséquence logique de `g`, on dit que ces deux formules sont *équivalentes*.

Exercice 7. Écrire une fonction

`consequence: formule -> formule -> bool`

qui, étant données une formule `f` et une formule `g`, détermine si `g` est conséquence logique de `f`. En déduire une fonction

`equivalentes: formule -> formule -> bool`

qui détermine si ses deux arguments sont des formules équivalentes.

Correction :

```
let consequence f g =
let vf = list_of_vars f
a□t v = list _of_vars
                let v =non   _otef
                              let ff21(s)-598(=)-597aoffet2[(a60tl)2(i)2(o)2(n)2(s)]TJ96.54851.998Td
                                  t2a[0fo
```

Frmnen(rm(ale)-66(conjonectv)30(le)-66((f)]TJ/405 4.34626 Tf27221761 0 Td [?f)]TJ/485 4.34626 Tf7.0341 0 Td [)e

```
    Et(g,h) -> Et(desc_ou g, desc_ou h)
  | Ou(g,h) ->
    let g' = desc_ou g
    and h' = desc_ou h in
      (match g',h' with
       _,Et(g,h) -> Et(desc_ou (Ou(g',g)),desc_ou (Ou(g',h)))
     | Et(f,g),_ -> Et(desc_ou (Ou(f,h')),desc_ou (Ou(g,h')))
     | _ -> Ou(g',h'))
  | f -> f
;;

let fnc f = desc_ou (desc_neg f);;
```