TD XLI

MANIPULATION DE FORMULES LOGIQUES

On considère des formules propositionnelles définies par le type suivant :

```
type formule =
  | Const of bool
  | Var of string
  | Et of formule * formule
  | Ou of formule * formule
  | Non of formule
```

1 Affichage d'une formule logique

Exercice XLI.1

p. 4

1. Écrire une fonction `string_of_formule` renvoyant la représentation infixe d'une expression sous la forme d'une chaîne de caractères. Le parenthésage doit être suffisant pour qu'il n'y ait pas d'ambiguïté (il peut être excessif).

```
string_of_formule : formule -> string
```

```
# string_of_formule (Ou (Et (Var "y", Const false), Non (Var "z")));;
- : string = "((y et false) ou non z)"
```

2. Pour minimiser le nombre de parenthèses utilisées, on définit les priorités suivantes :

- `Non` est prioritaire sur `Et` et `Ou` ;
- `Et` est prioritaire sur `Ou`.

Ainsi, `"non x_1 et x_2 ou non x_3"` signifie `"((non x_1) et x_2) ou (non x_3)"`.
On en profitera également pour se débarrasser des parenthèses rendues inutiles par l'associativité des différents opérateurs : on écrira `"x_1 et x_2 et x_3"` plutôt que `"x_1 et (x_2 et x_3)` ou `(x_1 et x_2) et x_3"`.
On donne la fonction `priorite : formule -> int` suivante :

```
let priorite = function
  | Var _ | Const _ -> max_int
  | Ou _ -> 0
  | Et _ -> 1
  | Non _ -> 2
```

Écrire une fonction `string_priorite` n'utilisant que les parenthèses nécessaires.

```
string_priorite : formule -> string
```

```
# string_priorite antinomie;;
- : string = "non x_0 et (x_2 et x_0 ou x_1) et non x_1 et non x_2"
```

2 Égalité syntaxique modulo associativité et commutativité

Dans cette section, on considère qu'une formule logique est un arbre de type `formule`, mais que deux formules peuvent être syntaxiquement égales sans correspondre au même arbre. Plus précisément, deux formules seront dites syntaxiquement égales si l'on peut passer de l'une à l'autre par l'application répétée des règles suivantes :

- $A \wedge (B \wedge C) \simeq (A \wedge B) \wedge C$ (Asso-$\wedge$)
- $A \vee (B \vee C) \simeq (A \vee B) \vee C$ (Asso-$\vee$)
- $A \wedge B \simeq B \wedge A$ (Com-$\wedge$)
- $A \vee B \simeq B \vee A$ (Com-$\vee$)

Exercice XLI.2 – Traitement artisanal de la commutativité p. 4

Dans cet exercice, on ne considère que les deux règles (Com-$\vee$) et (Com-$\wedge$).

1. Dans chacun des cas suivants, indiquer si les deux expressions sont égales modulo les transformations considérées :
   - **a.** $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_0 \wedge x_1)$;
   - **b.** $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_1 \wedge x_0)$;
   - **c.** $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_0) \wedge (x_1 \wedge x_3)$.
2. Écrire une fonction `egal_com` décidant si deux formules sont égales modulo la commutativité.

```
egal_com : formule -> formule -> bool
```

On devrait avoir :

```
# egal_commut ex2 ex3;;
- : bool = true
# egal_commut gros_ex1 gros_ex2;;
- : bool = false
```

Gérer l'associativité, et plus encore la combinaison de la commutativité avec l'associativité, est plus délicat, surtout si l'on veut un temps de calcul raisonnable. Pour commencer, on ne peut plus se contenter d'arbres binaires. On décide donc de définir un nouveau type :

```
type formule_asso =
  | C of bool
  | V of int
  | EtA of formule_asso list
  | OuA of formule_asso list
  | N of formule_asso
```

L'idée est ensuite de définir un représentant canonique pour chaque classe d'équivalence modulo associativité et commutativité, et une manière efficace de calculer ce représentant.

En OCaml, tous les types à l'exception des types fonctionnels sont dotés d'une relation d'ordre (totale). Cette relation est définie sur les types de base (pour `bool`, on a `false < true`) et ensuite étendue aux types algébriques de la manière suivante :

- pour un type produit `t = a1 * a2 * ... * an`, on prend l'ordre lexicographique induit par les ordres pré-existants sur `a1`, `a2`, . . . `an` ;
- pour un type somme `t = A1 of t1 | ... | An of tn`, on a `A1 x1 < A2 x2 < ... < An xn` quels que soient `x1, ..., xn` (et bien sûr `Ai x < Ai y` ssi `x < y`).
  Autrement dit, l'ordre induit sur un type somme est déterminé par l'ordre dans lequel les variantes apparaissent dans la définition du type.

Une expression de type `formule_asso` sera dite *canonique* si elle vérifie les conditions suivantes :

- aucun nœud `EtA` n'a d'enfant de la forme `EtA xs` ;
- aucun nœud `OuA` n'a d'enfant de la forme `OuA xs` ;
- pour chaque nœud de la forme `EtA enfants` ou `OuA enfants`, la liste `enfants` est triée par ordre croissant.

Exercice XLI.3 – Fonctions préliminaires

p. 5

1. Écrire une fonction `insere : 'a -> 'a list -> 'a list` insérant un élément dans une liste supposée triée.
2. Écrire une fonction `fusionne : 'a list -> 'a list -> 'a list` fusionnant deux listes supposées triées.

Exercice XLI.4 – Égalité syntaxique

p. 5

1. Écrire une fonction `canonique : formule -> formule_asso` mettant une expression sous forme canonique.
2. Écrire une fonction `egal_syntaxe : formule -> formule -> bool` décidant l'égalité syntaxique (modulo associativité et commutativité) de deux expressions logiques.

Solutions

Correction de l'exercice XLI.1 page 1

1. Pas de difficulté particulière :

```
let rec string_of_formule f =
  match f with
  | Const b -> sprintf "%b" b
  | Var s -> s
  | Et (f1, f2) ->
    sprintf "(%s et %s)" (string_of_formule f1) (string_of_formule f2)
  | Ou (f1, f2) ->
    sprintf "(%s ou %s)" (string_of_formule f1) (string_of_formule f2)
  | Non f1 ->
    sprintf "(non %s)" (string_of_formule f1)
```

2.
```
let rec string_priorite formule =
(* La seule possibilité pour priorite expr = prio_parent est que
 * expr et son parent aient le même opérateur à la racine.
 * - si c'est Non (Non f), il est inutile de parenthéser
 *   (opérateur unaire) ;
 * - si c'est Et (Et (f, g), h), il est également inutile de
 *   parenthéser puisque Et est associatif ;
 * - de même pour Ou. *)
  let parenthese f prio_parent =
    if priorite f >= prio_parent then
      string_priorite f
    else
      sprintf "(%s)" (string_priorite f) in
  match formule with
  | Const b -> string_of_bool b
  | Var s -> s
  | Non f -> sprintf "non %s" (parenthese f 2)
  | Et (ga, dr) ->
    sprintf "%s et %s" (parenthese ga 1) (parenthese dr 1)
  | Ou (ga, dr) ->
    sprintf "%s ou %s" (parenthese ga 0) (parenthese dr 0)
```

Correction de l'exercice XLI.2 page 2

1. **a.** Oui, il suffit d'appliquer la règle Com-$\wedge$ à la racine.
   b. Oui, avec deux applications de la règle Com-$\wedge$.
   c. Non, on aurait besoin de l'associativité ici.
2. À chaque nœud pertinent, on essaie les deux possibilités :

```
let rec egal_commut a b =
  match a, b with
  | Non ex, Non ex' -> egal_commut ex ex'
  | Var i, Var j -> i = j
  | Const b, Const b' -> b = b'
  | Et (ga, dr), Et (ga', dr') | Ou (ga, dr), Ou (ga', dr') ->
     (egal_commut ga ga' && egal_commut dr dr')
     || (egal_commut ga dr' && egal_commut ga' dr)
  | _ -> false
```

Au premier abord, la complexité de cette fonction semble être exponentielle en la taille de la formule (au moins dans le pire cas). Cependant, je ne suis pas du tout convaincu que ce soit vrai : si vous arrivez à construire une famille de formules pour laquelle on a effectivement une complexité exponentielle (ou au contraire à prouver que la complexité est polynomiale dans le pire cas,) je suis intéressé.

Correction de l'exercice XLI.3 page 3

1. À savoir faire, bien évidemment :

```
let rec insere formule liste =
  match liste with
  | [] -> [formule]
  | x :: xs when formule <= x -> formule :: x :: xs
  | x :: xs -> x :: insere formule xs
```

2. Tout aussi classique :

```
let rec fusionne enfants enfants' =
  match enfants, enfants' with
  | [], _ -> enfants'
  | _, [] -> enfants
  | x :: xs, y :: ys when x <= y -> x :: fusionne xs enfants'
  | _, y :: ys -> y :: fusionne enfants ys
```

Correction de l'exercice XLI.4 page 3

1. Le code n'est pas compliqué, mais li faut bien réfléchir à ce qu'on fait dans les cas `Et` et `Ou`. Notez que `fusionne [ga'] [dr']` est juste une manière concise de dénoter la liste `[ga'; dr']` ou `[dr'; ga']` (celle des deux qui est croissante).

```
let rec canonique  = function
  | Const b -> C b
  | Var s -> V s
  | Non ex -> N (canonique ex)
  | Et (ga, dr) ->
     begin
       let ga', dr' = canonique ga, canonique dr in
       match ga', dr' with
       | EtA enfants_g, EtA enfants_d -> EtA (fusionne enfants_g
       ↪  enfants_d)
       | EtA enfants_g, _ -> EtA (insere dr' enfants_g)
       | _, EtA enfants_d -> EtA (insere ga' enfants_d)
       | _, _ -> EtA (fusionne [ga'] [dr'])
     end
  | Ou (ga, dr) ->
     begin
       let ga', dr' = canonique ga, canonique dr in
       match ga', dr' with
       | OuA enfants_g, OuA enfants_d -> OuA (fusionne enfants_g
       ↪  enfants_d)
       | OuA enfants_g, _ -> OuA (insere dr' enfants_g)
       | _, OuA enfants_d -> OuA (insere ga' enfants_d)
       | _, _ -> OuA (fusionne [ga'] [ dr'])
     end
```

2. Le problème se ramène à l'égalité structurelle des deux représentants canoniques.

```
let egal_syntaxe f1 f2 =
  canonique f1 = canonique f2
```