ENSIIE 2014/2015 Programmation fonctionnelle

Examen de rattrapage 04/02/2015 — 1h45

— **Tout appareil électronique est interdit et doit être rangé dans un sac fermé.**
— Seules les notes de cours sur papier sont autorisées.
— La rédaction doit se faire au stylo, pas au crayon.
— Lisez tout l'énoncé (2 pages) avant de commencer, lisez attentivement les questions, vérifiez que les réponses que vous proposez correspondent aux questions posées ! Ça va mieux en le disant. . .
— TOUTES les fonctions doivent être précédées de commentaires indiquant les conditions d'utilisation et le résultat obtenu.

Exercice 1 Mots de Lyndon

Dans cet énoncé, un mot est une liste non vide `u = [u1;u2;...;un]` de caractères. La longueur d'un mot est la longueur ($n \geq 1$) de la liste. Exemple : `['e';'x';'a';'m';'e';'n']` est un mot de longueur 6. On définit la relation d'ordre strict $\prec$ (ordre lexicographique) sur l'ensemble des mots comme une généralisation de l'ordre du dictionnaire : deux mots u et v vérifient $u \prec v$ si u est différent de v et u est avant v dans le dictionnaire.

1. Définir la fonction `inferieur` de type `'a list -> 'a list -> bool` qui, étant donnés deux mots u et v, retourne vrai si $u \prec v$ et faux sinon. On rappelle que la comparaison sur les caractères se comportent ainsi : l'évaluation de `'e' < 'f'` retourne le booléen `true` et l'évaluation de `'f' < 'e'` retourne le booléen `false`.
   `inferieur ['b';'a';'c';'h';'e'] ['b';'r';'i';'n'] = true`
2. Un conjugué d'un mot `u = [u1;u2;...;un]` est un mot de la forme `u = [ui;...;un;u1;...;ui-1]` où $2 \leq i \leq n$. Définir la fonction `conjugue` de type `'a list -> int -> 'a list` qui, étant donnés un mot u de longueur n et un entier i tel que $2 \leq i \leq n$, retourne le conjugué du mot u qui débute par le ième élément de u.
   `conjugue ['a';'v';'e';'c'] 3 = ['e';'c';'a';'v']`
3. Un mot u est un mot de Lyndon si $u \prec v$ pour tout conjugué v de u. Définir la fonction `lyndon` de type `'a list -> bool` qui détermine si un mot est un mot de Lyndon.
   `lyndon ['a';'a';'a';'b'] = true`

Exercice 2 [Démonstration]

On considère la structure de données suivante permettant de représenter des arbres :

```
type 'a arbre = Noeud of 'a arbre * 'a * 'arbre | Vide
```

1. Programmer une fonction `somme` qui calcule la somme de toutes les valeurs présentes dans un arbre.
2. Rappeler la définition en Ocaml de l'itérateur `map` sur les arbres. On pourra s'inspirer de l'itérateur `List.map` des listes. Utiliser cette fonction `map` pour définir une fonction `double` qui construit un arbre dont les noeuds contiennent des valeurs qui sont le double de celles de l'arbre donné en paramètre.
3. Démontrer que pour tout arbre a, `somme (double a)` = 2 * `somme a`.

Exercice 3 Termes

On appelle *signature* un ensemble $\mathcal{F}$ de *symboles* chacun muni d'une *arité* entière (c'est-à-dire de son nombre d'arguments). C'est donc naturellement une association (map) des symboles vers les `int`, peu importe son implantation liste de couples ou à base d'arbres AVL, etc. On la considérera dans la suite munie de ses fonctions `empty_map`, `mem_map`, `find`, `fold_map`... (*pas forcément toutes nécessaires*). On rappelle quelques types :

```
(* empty_map  est l'association vide *)
empty_map : ('a , 'b) map
(* mem_map k m teste si une clé k est dans la map m *)
mem_map : 'a → ('a , 'b) map → bool
(* find k m retourne la valeur associée dans m à la clé k *)
(* PRÉCONDITION : k est dans la map *)
find : 'a → ('a , 'b) map → 'b
(* fold_map f m a compose les applications de f à tous les k et e_k formant
   une liaison dans m et à un accumulateur l'accumulateur initial étant a.
   (Fonctionnement ressemblant à fold_right sur les listes par exemple) *)
fold_map : ('a -> 'b -> 'c -> 'c) -> ('a , 'b) map -> 'c -> 'c
(* etc. *)
```

On suppose l'existence d'un ensemble de *variables* ; on considérera un constructeur particulier `Var` étiqueté par un `int` (son *identificateur*) pour représenter une variable sans chercher à représenter cet ensemble.

L'ensemble des *termes du premier ordre* construits sur une signature et un ensemble de variable est défini inductivement par :

— Une variable est un terme ;

— Si f est un symbole d'arité n et si $t_1 \dots t_n$ est une séquence de n termes, alors $f(t_1, \dots, t_n)$ est un terme. On dit qu'un terme est une constante s'il ne comporte qu'un symbole d'arité nulle.

1. Proposer un type concret pour les termes (sur un type quelconque de symboles).

Un bon programmeur utilise des fonctions futées pour construire des termes toujours bien formés, c'est-à-dire avec le bon nombre d'arguments par symbole. Il peut cependant arriver qu'on ait à vérifier que des termes sont bien formés.

2. On souhaite disposer d'une fonction `bien_forme` qui sur la donnée d'un terme t et d'une signature $\mathcal{F}$ retourne `true` si t est bien formé sur $\mathcal{F}$ et `false` sinon.

   Donner le type de cette fonction et en proposer une implantation.

3. On souhaite disposer d'une fonction `variables` qui sur la donnée d'un terme t retourne la liste des identificateurs de variables apparaissant dans t.

   Donner le type de cette fonction et en proposer une implantation.

4. On dit qu'un terme est *linéaire* si aucune variable n'apparaît plus d'une fois dans ce terme. Proposer une implantation d'une fonction `lineaire` qui sur la donnée d'un terme t retourne `true` si t linéaire et `false` sinon. On ne se souciera pas d'efficacité.

5. On souhaite renommer les variables d'un terme : proposer une fonction qui sur la donnée d'un terme t retourne un terme comportant les mêmes symboles dans le même agencement mais dont toutes les variables sont identifiées par des numéros n'apparaissant pas dans t (tout en conservant bien sûr leur nombre d'occurrences). **Par exemple** si les v_i sont des variables d'identificateur i le terme $f(v_1, g(v_2, v_1))$ pourra être renommé en $f(v_3, g(v_4, v_3))$.