ENSIIE 2014–2015 Programmation fonctionnelle

Examen 15/01/2015 — 1h45

— **Tout appareil électronique est interdit et doit être rangé dans un sac fermé.**
— Seules les notes de **ce cours** sur papier sont autorisées.
— La rédaction doit se faire au **stylo**, pas au crayon.
— Lisez tout l'énoncé (3 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.
— TOUTES LES FONCTIONS DOIVENT ÊTRE PRÉCÉDÉES DE LEUR TYPE (sinon c'est 0).

Exercice 0 [Darwin]

1. Lisez les instructions en début de sujet et toutes les questions avant de commencer.

Exercice 1 [Arbres binaires et localité]

On travaille sur des arbres binaires qui sont soit l'arbre vide, soit constitués d'un nœud contenant une valeur chaîne de caractères (c'est-à-dire de type `string`) et deux sous-arbres binaires (qu'on considérera comme respectivement le sous-arbre gauche et le sous-arbre droit). Deux arbres respectivement sous-arbre gauche et sous-arbre droit d'un nœud sont dits *frères*.

1. Proposez un type `tree` pour ces arbres binaires.
2. Proposez une fonction **`value`** qui sur la donnée d'un arbre retourne la valeur du nœud au sommet ou bien lève l'exception `Not_found` si l'arbre est vide.
3. Proposez une fonction qui sur la donnée d'un arbre retourne la chaîne de caractères correspondant à la concaténation (la mise bout-à-bout) de toutes les chaînes contenues dans l'arbre, obtenues dans l'ordre fils gauche, racine, fils droit. On utilisera pour la concaténation de deux chaînes la fonction notée `^` en position infixe et `(^)` en position *préfixe* et de type `string -> string -> string`.

Un arbre de ce type peut par exemple être utilisé pour représenter un document structuré et découpé en parties, sections, etc. Travailler localement (donc à une position de travail) dans un tel arbre entraîne cependant à chaque fois (puisque l'arbre est persistant) une copie en mémoire de tout le chemin depuis la racine jusques au nœud considéré...

Pour limiter les copies en mémoire on peut remarquer qu'on a besoin de connaître deux choses :
— Le sous-arbre de type `tree` dont on est actuellement à la racine,
— Le chemin jusques à la racine.

Ce chemin doit lui-même donner plusieurs informations :
— Il est simplement réduit à `T` (pour *top*), si on est au sommet (global),
— Il est construit à partir du constructeur `L`, si on est un fils *gauche*, sur le triplet : valeur du père, reste du chemin vers la racine, arbre de type `tree` représentant le frère droit,
— Il est construit à partir du constructeur `R`, si on est un fils *droit*, sur le triplet : valeur du père, arbre de type `tree` représentant le frère gauche, reste du chemin vers la racine.

4. Proposez un type `path` pour les chemins ; on considère dès lors les objets du type `zip` suivant :

```
type zip = Z of (tree * path)
```

Avec un chemin ainsi défini, il est facile de remonter d'un cran dans le `zip` : on sait reconstruire le `tree` père puisqu'on dispose des deux fils et de la valeur à stocker, et qu'on connaît le reste du chemin vers la racine.

5. Quel est l'objet de type `zip` ne contenant aucune valeur ?
6. Proposez une fonction `moveup` qui sur la donnée d'un objet de type `zip` retourne l'objet de type `zip` correspondant à un déplacement vers la racine ou bien lève l'exception `Top` si on est déjà au sommet.
7. Proposez une fonction `movedownleft` qui sur la donnée d'un objet de type `zip` retourne l'objet de type `zip` correspondant à un déplacement vers le fils gauche ou bien lève `Empty` le cas échéant.
8. Proposez une fonction `movedownright` qui sur la donnée d'un objet de type `zip` retourne l'objet de type `zip` correspondant à un déplacement vers le fils droit ou bien lève `Empty` le cas échéant.
9. Proposez une fonction `jumpleft` qui sur la donnée d'un objet de type `zip` retourne l'objet de type `zip` correspondant à un déplacement vers le *frère* gauche ou bien lève `Left` ou `Top` suivant les cas.
10. Proposez une fonction `jumpright` qui sur la donnée d'un objet de type `zip` retourne l'objet de type `zip` correspondant à un déplacement vers le *frère* droit ou bien lève `Right` ou `Top` suivant les cas.
11. Proposez une fonction `insertvalleft` qui, sur la donnée d'un objet `Z(`$t, chemin$`)` c'est-à-dire de type `zip` et comportant un arbre t à la position de travail, d'une valeur v de type `string` et d'un arbre l de type `tree`, retourne un objet de type `zip` correspondant à l'insertion à la position de travail de la valeur v avec comme fils gauche l et comme fils droit t.

Félicitations, vous avez réalisé un zipper sur arbres.

Exercice 2 [Démonstration]

1. Proposez une fonction `mem` qui sur la donnée d'un élément et d'une liste retourne `true` si l'élément est dans la liste et `false` sinon.
2. On considère la fonction suivante :

```
let rec dbl = fun l -> match l with
| [] -> []
| a::r -> let itbe = dbl r in
          if mem a r then
             if (mem a itbe) then itbe
             else a::itbe
          else itbe
```

Démontrez que le résultat de l'application de cette fonction à une liste l contient une seule occurrence de chacun des éléments de l qui apparaissent plusieurs fois dans l.

Exercice 3 [Récriture dans les mots] On dispose des types `('a,'b) pmap` et `'a avl` tels que vus en cours, avec toutes leurs fonctions (qui ne sont pas toutes utiles ici) :

```
empty_map       empty_avl       add_map         add_avl
mem_map         mem_avl         remove_map      remove_avl
fold_map        fold_avl
find_map
```

telles que vues en cours (**et dont vous n'oublierez pas de rappeler les types**). On rappelle que ces fonctions peuvent lever l'exception `Not_found`.

On appelle *alphabet* un ensemble fini de symboles (notés a_k dans la suite) et un *mot* une **séquence** finie de ces symboles a_1a_2 etc. On lit les mots de la gauche vers la droite.

Une *règle* $l \to r$ associe un mot r (le membre à droite) à un mot l (le membre à gauche), non vide et pas forcément de la même taille que r.

Si un mot $w = a_1a_2 \cdots a_{n-1}a_n$ contient un sous-mot égal à l alors w peut se récrire par $l \to r$ en w' où w' est comme w mais avec r à la place de l. Par exemple par la règle $CDC \to BBA$ le mot $JAIMEACDCYEAH$ se récrit en $JAIMEABBAYEAH$ (chacun ses goûts).

On appelle *système* un **ensemble** de telles règles. C'est en particulier une structure **associant** des séquences à des séquences.

1. Proposez un type pour les séquences de symboles (il est forcément paramétré par le type des symboles).
2. Proposez un type pour les systèmes (il est forcément paramétré par le type des symboles).
3. Proposez une fonction `rule` qui sur la donnée d'un mot w retourne le membre à droite associé si w est un membre à gauche ou lève l'exception `Not_found` sinon.
4. Proposez une fonction `rewshort` qui sur la donnée d'un mot récrit ce mot une fois grâce à un système. On remplacera s'il est présent le membre à gauche situé le plus à gauche dans w et si deux membres à gauche commencent au même endroit, on choisira le plus court des deux.
5. Proposez une fonction `rew` similaire à la précedent mais cette fois en choisissant le plus long des deux.
6. Proposez une fonction `reduce` qui récrit un mot tant que c'est possible (toujours pour la stratégie « plus à gauche plus long »).

   Par exemple pour le système sur un alphabet de chiffres :

   $$\left\{ \begin{array}{lcl} 12 & \to & 33 \\ 23 & \to & 11 \\ 13 & \to & 22 \end{array} \right\}$$

   On obtiendra à partir de 2123 le mot 3311.

   Cette fonction est-elle toujours définie ? (c'est-à-dire s'arrête-t-elle toujours en donnant un résultat ?)