ENSIIE 2012/2013 Programmation fonctionnelle

Examen de rattrapage 11/02/2013 — 1h45

Tout appareil électronique est interdit et doit être rangé dans un sac fermé. Les notes de cours sur papier sont autorisées. 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.

Exercice 1 [Files]

On a vu en cours que les listes étaient adaptées à l'implantation d'un type abstrait de séquence FILO, c'est-à-dire *First In, Last Out* : le premier élément qu'on met dans la séquence est le dernier auquel on pourra accéder.

On voudrait maintenant implanter le type abstrait des *files*. Les files sont des séquences FIFO, (First In, First Out) : *le premier élément qu'on met dans la séquence est cette fois le* premier *qu'on pourra en sortir*. Bref, c'est un tuyau.

On souhaite donc avoir pour le type `'a file` :

- Une fonction constante `vide` retournant une file vide.
- Une fonction `ajoute` qui sur la donnée d'un élément e et d'une file f retourne une file comportant les mêmes éléments que f dans le même ordre mais auxquels on aura ajouté l'élément e (e sera donc le dernier élément auquel on aura accès).
  Par exemple si on note $f = \boxed{e_k \cdots e_0}$ une file à $k+1$ éléments dans laquelle e_0 est rentré en premier, l'ajout de e à f sera $\boxed{e\ e_k \cdots e_0}$.
- Une fonction `extrait` qui sur la donnée d'une file f retourne *une paire* dont le membre de gauche contient le premier élément qu'on peut sortir de f et dont le membre de droite contient une file comportant les mêmes éléments que f dans le même ordre mais sans le premier élément auquel on avait accès. Cette fonction échoue (exception) lorsque f est vide.
  Par exemple si on note $f = \boxed{e_k \cdots e_1\ e_0}$ une file à $k+1$ éléments dans laquelle e_0 est rentré en premier, `extrait` f retournera $(e_0\ ,\ \boxed{e_k \cdots e_1})$.
- Une fonction `applique` qui sur la donnée d'une fonction g et d'une file f retourne une file de même taille que f mais où à la place de chaque élément e_i on trouve $(g\ e_i)$.
  Par exemple si on note $f = \boxed{e_k \cdots e_0}$ une file à $k+1$ éléments dans laquelle e_0 est rentré en premier, `applique` g f retournera $\boxed{(g\ e_k) \cdots (g\ e_0)}$.

1. Donner le type des fonctions précédentes.
2. Proposer une implantation pour chacune des quatre fonctions précédentes si on choisit : **`type`** `'a file = 'a list`.
3. Combien d'opérations (sur le type concret) faut-il pour ajouter un élément ?
4. Combien d'opérations (sur le type concret) faut-il pour extraire sur une file de longueur n (grossièrement) ?

On souhaite pouvoir effectuer l'extraction en un temps constant (la plupart du temps). On peut considérer l'implantation des files à l'aide de *deux* listes, l'une contenant les objets en tête de file dans l'ordre d'entrée, l'autre contenant les objets en queue de file, dans l'ordre d'extraction (donc l'ordre inverse d'entrée).

Par exemple la file d'entiers dans laquelle on a mis successivement $1, 2, 3, 4$ (donc $\boxed{4\ 3\ 2\ 1}$) peut être représentée à l'aide des deux listes `1::2::[]` et `4::3::[]`. Cette représentation n'est pas unique, on peut

également la représenter par les deux listes `1::[]` et `4::3::2::[]` ou même encore par les deux listes (*) `1::2::3::4::[]` et `[]` (exemple sur lequel on reviendra).

5. Définir un type concret pour les files à l'aide de deux listes.
6. Proposer une implantation sur ce type pour chacune des quatre fonctions sur les files. On veillera à n'effectuer de retournement de liste que lorsque cela est nécessaire (c'est-à-dire lorsque la représentation l'exige, comme par exemple dans l'exemple (*)).

On remarque que la structure proposée est symétrique et on peut réaliser les opérations d'ajout et d'extraction *aux deux extrémités* d'une file. On obtient alors ce qu'on appelle une *file à deux bouts*.

7. En appelant respectivement `entree` et `sortie` les extrémités d'entrée et de sortie de la file, coder les fonctions `ajoute_sortie` et `extrait_entree` (les fonctions `ajoute_entree` et `extrait_sortie` correspondent exactement aux fonctions `ajoute` et `extrait` déjà codées à la question 6).

Exercice 2 [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 $f(v_3, g(v_4, v_3))$.