X Informatique MP 2008 — Corrigé

Ce corrigé est proposé par Vincent Danjean (Enseignant-chercheur en école d'ingénieur) ; il a été relu par Marc Mezzarobba (ENS Ulm) et Vincent Puyhaubert (Professeur en CPGE).

Ce sujet traite de la manipulation de très grandes chaînes que l'on stocke sous forme d'arbres. Dans ces arbres (appelés cordes), un nœud représente la concaténation des chaînes stockées dans les sous-arbres. Ce genre de représentation est essentiel lorsque le coût de recopie des chaînes lors de concaténations « classiques » est trop important. C'est le cas pour la manipulation du génome en bio-informatique ou encore pour les éditeurs de texte qui veulent stocker un document entier.

Le sujet, de difficulté progressive, est découpé en 4 parties.

- La première partie traite de la manipulation des chaînes de longueur habituelle, représentées ici sous forme de listes (au lieu du traditionnel tableau de caractères). Les fonctions demandées dans cette partie permettent d'effectuer des opérations classiques : recherche d'un caractère, extraction d'un prefixe et d'un suffixe.
- La deuxième partie introduit quelques opérations relatives aux cordes, pour les construire ou en extraire des parties.

Les deux parties suivantes ont pour but de déterminer une méthode de rééquilibrage des cordes. Il s'agit d'une opération destinée à minimiser la hauteur de l'arbre afin de diminuer le coût d'accès à ses éléments. La difficulté est qu'il faut préserver l'ordre dans lequel apparaissent les feuilles lors d'un parcours en profondeur.

- La troisième partie présente une première méthode de rééquilibrage. On commence par traiter un exemple à la main, avant de produire le code et de montrer certains résultats sur la hauteur finale de l'arbre.
- La quatrième partie aborde une seconde méthode, qui est optimale. Il n'y a que du code à produire dans cette dernière partie puisque tous les résultats théoriques sont admis.

La plupart des questions demandent d'écrire des fonctions bien spécifiées par l'énoncé ou dont l'algorithme est décrit de manière détaillée. Les codes sont généralement simples mais ils font beaucoup intervenir la récursivité, car le sujet fait travailler sur des listes puis sur des arbres. Très peu de questions portent sur des analyses d'algorithmes ou de leur complexité.

Le rapport du jury souligne que les réponses courtes ont souvent été plus justes et mieux justifiées.

Indications

2 Faire une fonction récursive en remarquant que pour $i > 0$, la i-ième lettre d'une liste est la $(i-1)$-ième de sa queue.

3 Construire le préfixe récursivement. On peut remarquer que le préfixe de longueur $k > 0$ d'une liste est égal à la tête de cette liste suivie du préfixe de longueur $k-1$ de sa queue.

4 Remarquer que le suffixe $a_k, \ldots, a_n$ d'une liste $a_0, \ldots, a_n$ s'obtient pour $k > 0$ par un appel récursif simple sur un entier bien choisi et la queue de la liste.

5 Exploiter les champs disponibles dans le type `corde`.

6 Penser à utiliser les fonctions de la partie I pour calculer la longueur puis utiliser le constructeur adapté.

7 Traiter correctement le cas de la corde vide en argument.

8 Parcourir récursivement la corde et bien penser à mettre à jour la position recherchée quand on descend dans une sous-corde droite.

9 Reconstruire une nouvelle corde en effectuant un parcours en profondeur et en ne gardant que les parties intéressantes.

11 Compléter le tableau avec une boucle en utilisant pour le calcul d'une case les valeurs des deux cases précédentes.

12 Suivre la partie 1 de l'algorithme décrit au début de la partie III.

13 Bien vérifier les hypothèses faites sur les arguments de la fonction `inserer` lors des appels récursifs.

14 Implémenter les deux étapes de l'algorithme décrit au début de la partie III. Les feuilles x_j de la corde initiale pourront être insérées par une fonction annexe récursive.

15 Remarquer que dans l'étape 2 de l'algorithme, pour tout i, la corde obtenue après concaténation de l'élément c_i d'indice i du tableau `file` est de hauteur inférieure ou égale à $i-1$. Montrer ensuite que $Coût(c) \leqslant hn$.

16 Faire un parcours récursif de la corde en maintenant à jour l'indice de la prochaine feuille à stocker dans le tableau `q`.

17 Effectuer un parcours en profondeur de c_1 : à chaque feuille trouvée, noter la profondeur et chercher de quelle feuille il s'agit à l'aide du tableau `q`. Mettre alors à jour le tableau `prof`.

18 La nouvelle corde doit être construite de manière récursive par un nouveau parcours en profondeur. Utiliser comme argument du programme la profondeur actuelle dans l'arbre en construction `p` et l'indice de la nouvelle feuille à insérer `i`. On distinguera plusieurs cas suivant les valeurs respectives de `p` et `prof.(i)`.

I. Préliminaires sur les mots

1 Un mot étant représenté sous la forme d'une liste, la longueur d'un mot est la longueur de la liste.

```
let rec (longueurMot : mot -> int ) = function
    [] -> 0
  | _::t -> 1 + longueurMot t;;
```

Cette fonction a pour type :

```
longueurMot : mot -> int
```

> Le type de la fonction `longueurMot` est forcé ici. Ce n'est pas une obligation mais cela permet de restreindre le typage que `camllight` aurait inféré. En l'absence de typage explicite, il serait en effet
>
> ```
> longueurMot : 'a list -> int
> ```
>
> Un meilleur typage permet des vérifications du code lors de la compilation. C'est toujours une bonne chose.
>
> On pourrait penser ne contraindre que le type de l'argument :
>
> ```
> let rec longueurMot (m:mot) =
>   match m with
>       [] -> 0
>     | _::(t:mot) -> 1+longueurMot t;;
> ```
>
> Mais, contrairement à `OCaml`, `camllight` souffre de quelques bugs pour le typage forcé. Ainsi, si l'on ne force pas aussi le type de la variable `t` (comme cela est fait dans le code ci-dessus), `camllight` renverrait comme type `longueurMot : int list -> int`, ce qui est équivalent mais moins joli : il est mieux de faire afficher l'alias que l'on a spécialement créé pour le type.

2 Recherchons par une récursion terminale le i-ième élément de la liste.

```
let rec (iemeCar : int -> mot -> int) = fun i m ->
  match i with
      0 -> hd m
    | _ -> iemeCar (i-1) (tl m);;
```

Cette fonction a pour type :

```
iemeCar : int -> mot -> int
```

> Même remarque que précédemment : le typage forcé de `(tl m)` n'est là que pour contourner le bug de `camllight`.
>
> L'énoncé précise que l'on suppose que l'entier i a une valeur correcte ($0 \leqslant i < n$). Si ce n'est pas le cas, l'implémentation donnée déclenchera une exception lors de l'exécution de `tl m` ou de `hd m` (si i est trop grand), ou alors elle partira dans une récursion infinie (si i est négatif).

3 Construisons récursivement une liste constituée des k premiers éléments de la liste de départ.

```
let rec (prefixe : int -> mot -> mot) = fun k m ->
  if k <= 0
  then []
  else (hd m)::(prefixe (k-1) (tl m));;
```

Cette fonction a pour type :

```
prefixe : int -> mot -> mot
```

> On peut aussi utiliser une récursion terminale pour plus d'efficacité (le compilateur évite alors de gérer une pile d'appels de fonction). Il faut alors utiliser un paramètre supplémentaire (dans une fonction auxiliaire) qui stocke (en ordre inverse) le début du mot.
>
> ```
> let (prefixe : int -> mot -> mot) = fun k m ->
>   let rec prefixe_rec k m accu =
>     if k <= 0
>     then rev accu
>     else prefixe_rec (k-1) (tl m) ((hd m)::accu)
>   in prefixe_rec k m [];;
> ```

4 Il suffit de renvoyer la fin de la liste après avoir éliminé les k premiers éléments. Comme on souhaite obtenir la fin de la liste de départ, il n'y a pas besoin d'en construire une nouvelle.

```
let rec (suffixe : int -> mot -> mot) = fun k m ->
  if k = 0
  then m
  else suffixe (k-1) (tl m);;
```

Cette fonction a pour type :

```
suffixe : int -> mot -> mot
```

II. Opérations sur les cordes

> L'énoncé impose le type `Caml` suivant :
>
> ```
> type corde =
> | Vide
> | Feuille of int*mot
> | Noeud of int*corde*corde;;
> ```
>
> Il précise ensuite que les sous-cordes d'un `Noeud` ne peuvent pas être `Vide`.
>
> Le rapport du jury souligne l'importance dans cette partie de toujours respecter cette propriété.
>
> On peut remarquer que cette propriété aurait pu être directement intégrée dans les types utilisés, par exemple en écrivant :