TD #7 — Pile ou file, analyse amortie, références… (avec solutions)

Exercice 1. *File avec deux piles*

On esquisse la stratégie suivante pour implémenter une file avec deux piles :

— La file vide correspond à deux piles vides s_1 et s_2
— On enfile les éléments en les empilant dans s_1
— Pour défiler : si s_2 est vide on dépile tout s_1 et l'empile dans s_2, et on dépile une fois s_2, sinon on dépile juste une fois s_2

1. Illustrez cette stratégie avec quelques dessins.

On souhaite implémenter une file fonctionnelle en OCaml en suivant cette stratégie ; on se dotera pour cela d'un `type 'a queue = 'a list * 'a list`.

2. Donnez une valeur `(empty_queue:'a queue)` qui correspond à une file vide.

```
let (empty_queue:'a queue) = [], []
```

3. Définissez une fonction `push : 'a queue -> 'a -> 'a queue` telle que `push q x` construit une nouvelle file obtenue en ajoutant `x` à la file `q`.

On propose :
```
let push (s1,s2:'a queue) (x:'a) : 'a queue = (x::s1),s2
```

4. De même pour une fonction `pop : 'a queue -> 'a * 'a queue` telle que pour une file `q` non vide `pop q` s'évalue en l'élément en tête de `q` et une nouvelle file égale à la queue de celle-ci.

On utilise un simple renversement de liste dans le cas où il est nécessaire de dépiler tout s_2 dans s_1.
On propose deux versions, la première effectuant un filtrage partiel (qui bien que ne pouvant jamais échouer ici n'est pas autorisé en principe), et la seconde ayant un coût peut-être moins lisible (bien qu'identique) à cause de son utilisation d'une récursion :

```
let pop (s1,s2:'a queue) : 'a * 'a queue
    = match s1, s2 with
    | [], []   -> failwith "empty queue"
    | _, x::xs -> x, (s1, xs)
    | _, []    -> let x::xs = List.rev s1 in x, ([], xs)

let rec pop' (s1,s2:'a queue) : 'a * 'a queue
    = match s1, s2 with
    | [], []   -> failwith "empty queue"
    | _, x::xs -> x, (s1, xs)
    | _, []    -> pop' ([], List.rev s1)
```

On souhaite maintenant prouver (ou tout du moins argumenter) la correction de notre implémentation relativement aux spécifications d'une pile fonctionnelle. Pour cela, on va montrer que *lorsqu'elles n'échouent pas* les fonctions `push` et `pop` préservent certains invariants sur la structure.

5. Montrez que les éléments de s_1 apparaissent dans l'ordre inverse d'arrivée (le sommet de la pile a été `push`ed après tous les autres éléments, récursivement).

— Pour une file vide : trivial
— Après un `push (s1,_) x` : la propriété est vraie pour `s1` par hypothèse, et le reste donc pour la nouvelle file `s1',s2'` construite par `push` comme `(x::s1),s2`
— Après un `pop (s1,_)` : la propriété est vraie pour `s1` par hypothèse, et le reste donc pour la nouvelle file `s1',s2'` construite par `pop` comme `s1,_` ou `[],_` (en fonction des cas).

6. Montrez que les éléments de s_2 sont plus anciens que ceux de s_1 (ont été `push`ed avant).

— Pour une file vide : trivial
— Après un `push (s1,s2) x` la propriété est vraie pour `s1,s2` par hypothèse, et le reste donc pour la nouvelle file `s1',s2'` construite par `push` comme `(x::s1),s2`
— Après un `pop (s1,s2)` : la propriété est vraie pour `s1,s2` par hypothèse. On distingue deux cas : si `x::xs = s2` est non vide, la nouvelle file est `s1, xs` avec `xs` des éléments de `s2`, et la propriété reste vraie. Sinon la nouvelle file est `[], x` avec `x` des éléments ayant été `push`ed par le passé, et la propriété reste également vraie.

7. Montrez que les éléments de s_2 apparaissent dans l'ordre d'arrivée (le sommet de la pile a été `push`ed avant tous les autres éléments, récursivement).

— Pour une file vide : trivial
— Après un `push (s1,s2) x` : la propriété est vraie pour `s2` par hypothèse, et le reste donc pour la nouvelle file construite comme `_,s2`
— Après un `pop (s1,s2)` : la propriété est vraie pour `s2` par hypothèse. On distingue deux cas : si `x::xs = s2` est non vide, la nouvelle file est `_, xs` avec `xs` des éléments de `s2`, et la propriété reste vraie. Sinon soit `x::xs = List.rev s1` la nouvelle file est `_, xs`. Or par 5 les éléments de `s1` y apparaissent par ordre inverse d'arrivée ; ceux de leur miroir `x::xs` apparaissent donc dans l'ordre, et cela reste le cas de ceux de `xs`.

8. Conclure

Il « suffit de montrer » que lorsqu'un élément est renvoyé par `pop`, celui-ci est le plus ancien qui était encore présent dans la file. Par la définition de cette dernière c'est l'élément en sommet de s_2 (ici on triche un peu pour traiter le cas où s_2 était vide, car il faudrait considérer une version « intermédiaire » de celle-ci correspondant à `List.rev`) s_1, qui par 7 est plus ancien que tous les autres éléments de s_2, et par 6 plus ancien que tous les éléments de s_1.

On s'intéresse enfin à l'efficacité de cette stratégie d'implémentation d'une file.

9. Montrez que le coût pire cas d'une opération `pop` sur une file contenant n éléments est un $O(n)$.

Immédiat en considérant une file de la forme `s1, []` où n est donc le nombre d'éléments dans `s1` : `pop` effectue un appel `List.rev s1` de coût $O(n)$, et le reste des opérations sont de coût constant.

Ceci ne paraît pas mieux qu'une implémentation naïve comme celle vue en TP. On va cependant montrer que l'on a ici un bien meilleur coût *amorti*.

10. Esquissez une preuve du fait que le coût pire cas de n opérations `push` et/ou `pop` (c'est à dire, le pire cas de la somme des coûts) à partir d'une file vide est un $O(n)$.
Indication : comptez pour chaque élément le nombre d'opérations dans lesquelles il est impliqué (sa contribution au coût global) au cours de sa « présence dans la structure ».

En suivant l'indication, on remarque qu'au cours de sa présence dans la structure, un élément est au plus ajouté une fois dans s_1, transféré une fois dans s_2, et extrait une fois depuis s_2. Chacune de ces opérations est implémentée par une ou deux opérations de pile, qui sont de coût constant pour l'implémentation utilisée ici. Ainsi, la contribution d'un élément au coût total des opérations est une constante, et le coût de n opérations *à partir d'une file vide* est un $O(n)$. (Ceci ne serait pas nécessairement vrai pour n opérations manipulant une file non vide, car les éléments déjà présents dans la file pourraient également contribuer au coût total.)

Remarques

Il faudrait (pour rendre cette esquisse plus rigoureuse) préciser ce qu'on entend par « présence dans la structure » pour une structure de données fonctionnelle comme celle étudiée ici.

Une analyse de complexité amortie peut en principe s'appliquer à n'importe quel algorithme, mais elle n'est souvent pertinente que quand on s'intéresse à des opérations manipulant et faisant évoluer un certain « état » (par mutation explicite ou non, comme c'est le cas ici), et donc souvent dans le cas d'analyse de structures de données.

Il existe plusieurs approches « classiques » pour établir une complexité amortie (dont aucune n'est officiellement au programme). Celle utilisée ici se rapproche probablement le plus de la méthode dite « comptable » (on attribue un budget à chaque élément lui permettant de payer pour toutes les opérations dans lesquelles il sera impliqué) ; d'autres approches sont par exemple la méthode « du potentiel » (on définit une fonction de potentiel sur les états de la structure de donnée (typiquement), telle qu'un état ne peut jamais être de potentiel négatif, et les opérations qui font augmenter le potentiel ne coûtent pas trop cher, et les opérations qui coûtent cher font baisser le potentiel), ou « de l'agrégat » (on fait une moyenne…).

Exercice 2. *Références et pile impérative en OCaml*

On décrit brièvement les *références* OCaml par les trois fonctions suivantes :

- `ref : 'a -> 'a ref` telle que `ref x` s'évalue en une (nouvelle) *référence* de valeur `x` ;
- `(!) : 'a ref -> 'a` opérateur unaire préfixe tel que `!r` s'évalue en la valeur associée à la référence `r` ;
- `(:=) : 'a ref -> 'a -> unit` opérateur binaire infixe tel que `r := x` *modifie* la valeur référencée par `r` en `x`. (Toute ressemblance avec la notion de pointeur n'est pas fortuite.)

On définit alors le `type 'a stack = 'a list ref` d'une pile *impérative* (ou mutable).

1. Définissez une fonction `make : unit -> 'a stack` qui construit et renvoie une *nouvelle* pile vide. Pourquoi une telle fonction n'était-elle pas nécessaire pour une pile fonctionnelle (et le devient maintenant) ?

   On propose :
   ```
   let make () : 'a stack = ref []
   ```
   Une telle fonction n'est pas nécessaire pour une structure de données fonctionnelle (immuable), car une *valeur* de pile vide peut être **partagée** par toutes les instances de la structure. Ce partage n'est plus possible quand la mutabilité entre en jeu : plusieurs instances de piles distinctes ne peuvent pas toutes modifier le même état (sans créer des problèmes).

2. Définissez une fonction `is_empty : 'a stack -> bool` qui teste si son argument est une pile vide. Pourquoi une telle fonction n'était elle pas nécessaire etc. ?

   On propose :
   ```
   let is_empty (s:'a stack) : bool = !s = []
   ```
   Une telle fonction n'était pas nécessaire pour une structure fonctionnelle, car il suffisait alors de comparer une instance de pile à l'« unique » pile vide. Une telle valeur n'existant plus (ou plutôt, n'ayant plus aucun intérêt à être définie), une fonction dédiée devient nécessaire.

3. Définissez une fonction `push : 'a stack -> 'a -> unit` qui met à jour (modifie) la valeur référencée par son premier argument pour y ajouter son second en sommet.

   On propose :
   ```
   let push (s:'a stack) (x:'a) : unit = s := x::!s
   ```

4. Définissez une fonction `pop : 'a stack -> 'a` qui (si possible) met à jour (modifie) la valeur référencée par son premier argument pour en supprimer la tête, et s'évalue en cette dernière.
   Indication de syntaxe : rappelez vous qu'un `let` lie vraiment un motif, et que l'on peut par exemple écrire `let () = e1 in e2` pour toute expression `(e1:unit)`.

   On propose :
   ```
   let pop (s:'a stack) : 'a
       = match !s with
       | [] -> failwith "empty stack"
       | x::xs -> let () = s := xs in x
   ```

Exercice 3. *Valores menores más cercanos*

Essayez de prouver **rigoureusement** (et en autonomie) la correction d'une de vos fonctions `vmmc2` (en OCaml ou C) du TP d'hier (indication : vous devriez avoir besoin d'au moins deux invariants).
Prouvez aussi que la version « tableau » en C est (également) de coût linéaire en la longueur de son argument (procédez par analogie avec la version utilisant explicitement une pile).