TP 3 : Piles et files en OCaml

1 Files fonctionnelles

Dans cette partie, on réalise une file fonctionnelle en utilisant deux listes, comme décrit dans le cours. On choisit le type le plus simple :

```
type 'a file_fonct = 'a list * 'a list
```

Les fonctions élémentaire sont :

```
(* Une constante égale à la file ne contenant aucun élément. *)
file_vide = ([], [])
```

```
(* ajoute x f renvoie une nouvelle file contenant les éléments de f, plus x *)
ajoute : 'a -> 'a file_fonct -> 'a file_fonct
```

```
(* enleve f renvoie Some (x, f'), où x est l'élément le plus ancien
   de f et f' est la file obtenue en enlevant x à f. Si f est vide, on
   renverra None. *)
enleve : 'a file_fonct -> ('a * 'a file_fonct) option
```

1.1 Programmation des opérations élémentaires

1. Écrire une fonction `miroir : 'a list -> 'a list` se comportant comme `List.rev`. On *exige* une complexité linéaire en la taille de la liste.

   ```
   # miroir [1; 2; 3; 4];;
   - : int list = [4; 3; 2; 1]
   ```

2. Écrire la fonction `ajoute`.
3. Écrire la fonction `enleve`.

1.2 Opérations supplémentaires

Dans cet exercice, sauf mention explicite du contraire, on n'interagira avec les files que par le biais des opérations élémentaires définies plus haut. Autrement dit, on « oublie » qu'une file est représentée par un couple de listes.

1. Écrire une fonction `somme : int file_fonct -> int` renvoyant la somme des éléments d'une file.
2. Écrire une fonction `file_fonct_of_list : 'a list -> 'a file_fonct` qui convertit une liste en file, l'élément de tête de la liste se retrouvant à la sortie de la file.
3. Proposer une version plus simple et plus efficace de cette fonction en s'autorisant à interagir directement avec la représentation concrète du type file.
4. Écrire une fonction

   `itere_file : ('a -> unit) -> 'a file_fonct -> unit`

   telle que l'appel `itere_file f file`, où `file` $= (x_1, \ldots, x_n)$ (avec x_n le plus ancien élément), soit équivalent à appeler la fonction `f` successivement sur x_n, puis x_{n-1}, …, puis x_1.
5. Écrire une fonction `afficher : int file_fonct -> unit` qui affiche tous les éléments d'une file d'entiers, dans leur ordre d'insertion. On rappelle que :
   — `print_int : int -> unit` permet d'afficher un entier.
   — `print_newline : unit -> unit` permet de revenir à la ligne.

```
# afficher (file_fonct_of_list [1; 2; 3; 4]);;
1
2
3
4
- : unit = ()
```

1.3 Complexité amortie

On note $|u|$ la longueur d'une liste u, et l'on définit le *potentiel* d'une file $f = (e, s)$ par $\Phi(f) = 2|e|$. On mesure la complexité d'une fonction par le nombre d'utilisations du constructeur `::`, en comptant à la fois la version « destructive », qui permet de récupérer la tête et la queue d'une liste pré-existante, et la version « constructive », qui permet de construire `x :: xs` à partir de `x` et de `xs`.

1. On considère une file f et une opération qui est soit une extraction, soit l'ajout d'un élément quelconque. On note $C_{op}(f)$ le coût de l'opération op sur la file f et f' la file obtenue après l'opération. Donner une majoration précise de $C_{op}(f) + \Phi(f') - \Phi(f)$.
2. En déduire que le coût total de n opérations successives, ajouts ou suppressions, sur une file initialement vide est en $O(n)$, et donc que le coût moyen par opération, sur cette série de n opérations, est en $O(1)$. On dira donc que l'insertion et l'extraction ont une *complexité amortie* en $O(1)$.

2 Piles et files impératives

2.1 Réalisation d'une pile par un tableau

On définit :

```
type 'a pile = {donnees : 'a option array; mutable taille : int}

let capacite p = Array.length p.donnees
```

On rappelle le principe de la réalisation et on pourra se référer au schéma du cours :
— Une pile a une taille maximale, on parlera de capacité, fixée à la création : c'est la taille du tableau `donnees`.
— À tout moment, `taille` donne le nombre d'éléments dans la pile.
— Les éléments sont empilés de la gauche vers la droite du tableau : les cases du tableau situées à droite du sommet actuel peuvent contenir n'importe quoi, `None` typiquement, mais pas nécessairement, et sont considérées comme vides.

Définir les fonctions suivantes :

1. La fonction `nouvelle_pile : int -> 'a pile` qui crée une pile vide, l'entier donne la capacité.

```
# let s = nouvelle_pile 2;;
val s : '_a pile = {donnees = [|None; None|]; taille = 0}
```

2. Les fonctions `pop : 'a pile -> 'a option` et `push : 'a -> 'a pile -> unit`.

```
# push 10 s;;
- : unit = ()
# push 20 s;;
- : unit = ()
# pop s;;
- : int option = Some 20
# push 1 s;;
- : unit = ()
# push 3 s;;
Exception: Failure "Pile pleine".
```

2.2 File dans un tableau circulaire

On définit :

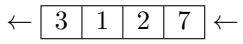
```
type 'a file_i =
  {donnees : 'a option array;
   mutable entree : int;
   mutable sortie : int;
   mutable cardinal : int}
```

Pour une file `f` :
— `f.entree` pointe vers la case où se fera la prochaine insertion.
— `f.sortie` pointe vers la case où se fera la prochaine extraction.
— `f.cardinal` indique la taille actuelle de la file, c'est-à-dire le nombre de cases actuellement utilisées.
— si la file est vide, on aura `f.entree = f.sortie`, mais cette valeur peut être quelconque.

En OCaml, l'opération `n mod p` renvoie un nombre négatif si $n < 0$ et $p > 0$: pour éviter d'avoir à en écrire une variante, on décide que la file se déplacera vers la droite, c'est-à-dire que `f.entree` ou `f.sortie` sera *incrémenté* à chaque opération.

2.2.1 Premières fonctions

1. Représenter deux `'a file_i` de capacité 6 correspondant à la file



   — une avec `f.entree < f.sortie` ;
   — une avec `f.entree > f.sortie`.
2. Il est pratique mais pas nécessaire de disposer des trois entiers `f.entree`, `f.sortie` et `f.cardinal`. Expliquer comment retrouver `f.sortie` si l'on dispose de `f.entree` et `f.cardinal`.
3. Expliquer pourquoi il n'est en revanche pas possible de retrouver `f.cardinal` à partir de `f.sortie` et `f.entree`. On pourra éventuellement suggérer une « astuce » permettant de régler ce problème sans stocker le cardinal.

2.2.2 Implémentation

Écrire les fonctions suivantes :

1. La fonction `file_vide_i : int -> 'a file_i` où l'entier spécifie la capacité de la file.
2. La fonction `capacite_i : 'a file_i -> int` qui renvoie la capacité.
3. La fonction `ajoute_i : 'a -> 'a file_i -> unit`. Si la file est pleine, on le signalera en levant une exception par `failwith "Insertion dans file pleine"`.
4. La fonction `enleve_i : 'a file_i -> 'a option`. On modifiera la file, et renverra `None` si elle est déjà vide.
5. La fonction `de_liste_i : 'a list -> int -> 'a file_i`. On demande que `de_liste_i u n` crée une file de capacité n et y ajoute les éléments de u en commençant par l'élément de tête. On pourra supposer sans le vérifier que $n \geqslant |u|$.

2.3 Fonctions supplémentaires sur les piles

2.3.1 Piles

Techniquement, les trois fonctions définies à la partie **??** (plus la fonction `capacite`) suffisent pour réaliser toutes les opérations imaginables sur des piles. Dans cet exercice, on se restreint à n'interagir avec les piles que *via* ces fonctions : on « oublie » donc l'existence d'un tableau sous-jacent.

1. Écrire une fonction `peek_1 : 'a pile -> 'a option`, qui renvoie `Some x` si on l'appelle sur une pile de sommet x, `None` si on l'appelle sur une pile vide. Cette fonction **ne doit pas modifier son argument** :

```
# let s = nouvelle_pile 3;;
val s : '_a pile = {donnees = [|None; None; None|]; courant = -1}
# push 3 s;;
- : unit = ()
# peek_1 s;;
- : int option = Some 3
# peek_1 s;;
- : int option = Some 3
```

2. Écrire une fonction `est_vide_1 : 'a pile -> bool` qui détermine si son argument est vide.

3. On souhaite écrire une fonction `egal : 'a pile -> 'a pile -> bool` qui détermine si deux piles sont égales, c'est-à-dire contiennent les mêmes éléments dans le même ordre, sans nécessairement avoir la même capacité. La fonction ci-dessous a deux problèmes : quels sont-ils ?

```
let egal_faux s t =
  while not (est_vide_1 s) && not (est_vide_1 t) && pop s = pop t do
    ()
  done;
  est_vide_1 s && est_vide_1 t
```

4. Écrire une fonction `iter_destructif : (f : 'a -> unit) -> (s : 'a pile) -> unit` qui applique la fonction f successivement à chacun des éléments de s en commençant par le sommet. À la fin de l'appel, s sera vide. Par exemple, si $s = (12, 35, 1)$ (où le sommet est 1), un appel à `iter_destructif print_int s` doit afficher les entiers 1, 35 et 12 dans cet ordre.
5. Compléter la fonction suivante pour qu'elle renvoie une copie de son argument. Cet argument ne doit pas être modifié au cours de l'appel. Plus précisément, il doit être remis dans son état initial avant la fin de l'appel.

```
let copie s =
  let s_copie = nouvelle_pile (capacite s) in
  let miroir = nouvelle_pile (capacite s) in
  iter_destructif (fun x -> .........) s;
  iter_destructif (fun x -> .........) miroir;
  s_copie
```

6. Écrire alors une version correcte de la fonction testant l'égalité de deux piles. On n'hésitera pas à réutiliser la version « fausse » pour écrire le moins de code possible.

2.3.2 Quelques fonctions sur les piles, pour plus d'efficacité

L'exercice précédent illustre assez bien la différence entre « possible » et « raisonnable » : certes, nous avons réussi à écrire une fonction d'égalité, mais au prix de contorsions rocambolesques. Comme l'utilisateur de notre structure de données n'aura effectivement pas accès au type concret sous-jacent (*a priori*, il ne saura même pas que les piles sont implémentées à l'aide de tableaux), il est de notre responsabilité de lui fournir *celles des fonctions qui sont beaucoup plus simples à écrire en ayant accès à l'implémentation.* Typiquement :

— `iter_destructif` s'écrit de manière très naturelle avec uniquement des `pop` et n'est pas d'utilisation très courante : on peut laisser l'utilisateur l'implémenter.

— une version non destructive de `iter`, qui ferait la même chose mais ne viderait pas la pile, est en revanche pénible à écrire avec l'interface minimale : essentiellement, on va devoir copier la pile et faire un `iter_destructif` sur la copie, ce qui pose aussi un problème d'efficacité. Cette fonction gagnerait donc à être rajoutée à l'interface de manière à pouvoir utiliser la représentation concrète des piles.

Dans cet exercice, on s'autorise donc de nouveau à utiliser la représentation concrète des piles, et on rajoute quelques fonctions à l'interface. On essaiera d'avoir des implémentations simples et efficaces.

1. Fonction `est_vide : 'a pile -> bool` (une ligne).
2. Fonction `flush : 'a pile -> unit` qui vide son argument sans rien faire avec ses éléments. On attend une implémentation en temps constant.
3. Fonction `egal : 'a pile -> 'a pile -> bool`. Attention, il faut bien tester l'égalité des éléments des piles, ce qui n'est pas la même chose que l'égalité des contenus des tableaux.
4. Fonction `itere : ('a -> unit) -> 'a pile -> unit`, version non destructive de `iter_destructif`.