Université Paris Diderot
L3 Informatique

Programmation Fonctionnelle
Année 2016-2017

TP n°6

Références

Le but de ce TP est de vous familiariser avec les références en OCaml. Les variables en OCaml ne sont pas mutables – il s'agit simplement de noms donnés à des valeurs d'expressions. Pour manipuler des "variables" au sens usuel de la programmation impérative (*eg.* à la C), on se sert de types OCaml de la forme `'a ref`, où `'a` peut être n'importe quel type. Voici par exemple la manière dont on peut définir une nouvelle référence vers un `char` :

```
let x = ref 'b';;
val x : char ref = {contents = 'b'}
```

On peut ensuite accéder en lecture à la valeur référencée par `x` :

```
!x;;
- : char = 'b'
```

On peut également modifier cette valeur (noter que cette modification est une *action*, de type `unit`) :

```
x := 'c';;
- : unit = ()
```

Deux références peuvent peuvent être *partagées*, c'est-à-dire référencer la même valeur :

```
let y = x;;
val y : char ref = {contents = 'c'}
y := 'd';;
- : unit = ()
!y;;
- : char = 'd'
!x;;
- : char = 'd'
```

Une pile LIFO

Le but de cette section est de mettre en œuvre la structure de données des piles (*stack* an anglais) regroupant leurs éléments dans une liste suivant le principe du "dernier arrivé, premier sorti", (*Last In, First Out* en anglais). La différence par rapport aux listes définissables en fonctionnel pur est que les primitives d'une pile peuvent modifier son état pendant leur execution.

Par example, si `s` est une pile non vide, un appel de la forme `pop s` renverra l'élément en tête de `s` (comme `List.hd` appliqué à une liste non vide), mais en plus, cet appel modifiera `s` en retirant cet élément de la pile.

Exercice 1. Définir le type `'a stack` comme le type des réferences vers `'a list`. Écrire une fonction `create : unit -> 'a stack` renvoyant une nouvelle pile vide. Écrire une fonction `empty : 'a stack -> boolean` renvoyant `true` si et seulement si son argument est une pile vide.

Exercice 2. Écrire une fonction `pop : 'a stack -> 'a` renvoyant, s'il existe, l'élément en tête d'une pile, et retirant cet élément de la pile. Par example, si `s` vaut `{contents = [4; 3; 2]}`, `pop s` renverra `4` et modifiera `s` en `{contents = [3; 2]}`. Que doit faire `pop` si la pile est vide ?

Exercice 3. Écrire une fonction `peek : 'a stack -> 'a` renvoyant, comme `pop`, le premier élément d'une pile, mais sans retirer cet élément de la pile.

Exercice 4. Écrire une fonction `push : 'a -> 'a stack -> unit` permettant d'ajouter un élément en tête d'une pile. Par example, si `s` vaut `contents = [4; 3; 2]`, `push 1 s` renverra `()` et modifiera la valeur de `s` en `{contents = [1; 4; 3; 2]}`.

Exercice 5. Écrire une fonction `is_equal: 'a stack -> 'a stack -> bool` renvoyant `true` si et seulement les deux piles passées en arguments contiennent la même suite d'éléments. Cette fonction peut s'écrire en moins de 50 caractères, espaces compris.

Exercice 6. Écrire une fonction `clone : 'a stack -> 'a stack` renvoyant une copie de la pile passée en argument. Attention, `clone s` doit créer une *nouvelle* pile contenant les mêmes éléments que `s`, et non partager la référence `s`.

Exercice 7. Testez vos fonctions sur le code suivant, et vérifiez que le comportement est bien celui attendu.

```
let t1 = let t = create () in push 3 t; push 4 t; t;;
let t2 = let t = create () in push 3 t; push 4 t; push 1 t; t;;

is_equal t1 t2;;                  (* false *)
is_equal t1 t1;;                  (* true  *)
let _ = pop t2 in is_equal t1 t2;; (* true  *)

let t3 = clone t2;;

is_equal t2 t3;;                  (* true  *)
let _ = pop t2 in is_equal t2 t3;; (* false *)
```

Exercice 8. Écrire enfin une fonction `pop_n : int -> 'a stack -> 'a` qui, étant donnés un entier `n` > 0 et une pile `s`, renvoie le `n`-ième element de `s` (la tête étant le premier élément) en retirant de `s` cet élément et tous ses prédécesseurs. Que doit faire `pop_n` si la pile contient moins de `n` éléments ?

Listes simplement chaînées

Une référence n'est qu'un cas particulier d'*enregistrement à champ mutable*. Un enregistrement peut contenir à la fois des valeurs mutables et des valeurs non mutables. Vous avez vu en cours l'exemple de l'implementation des listes simplement chaînées, dont nous rappelons le code ci-dessous.

Un élément d'une liste simplement chaînée est représenté comme un enregistrement contenant une valeur (non-mutable), et une référence (mutable) vers l'élément suivant dans la liste. Une liste est représentée comme un enregistrement à champs mutables contenant le premier et le dernier élément de cette liste. Pour tenir compte du fait que ces références peuvent être indéfinies (le dernier élément d'une liste n'a pas de successeur, et la liste vide n'a ni premier, ni dernier élément), nous utilisons un type OCaml `option`.

```
type 'a slllist  =
  {mutable first : 'a sllelement option;
   mutable last  : 'a sllelement option }
and 'a sllelement =
  {value : 'a;
   mutable next  : 'a sllelement option };;
```

```
let create_sll () =
  {first = None;
   last  = None };;

let app_sll e l =
  let c = {value = e; next = l.first} in
  l.first <- Some c;
  match l.last with
    None -> l.last <- Some c; l
  | _    -> l;;

let head_sll l = match l.first with
    None   -> failwith "empty list"
  | Some a -> a.value;;

let tail_sll l = match l.first with
    None   -> failwith "empty list"
  | Some a -> l.first <- a.next; l;;
```

Exercice 9. Écrire une fonction `is_equal : 'a slllist -> 'a slllist -> bool` permettant de vérifier que deux listes sont égales (c.à-d. contiennent la même suite de valeurs).

Exercice 10. Écrire une fonction `map : ('a -> 'b)-> 'a slllist -> 'b slllist` se comportant comme `List.map`, mais opérant sur les listes simplement chaînées (testez votre code).

Exercice 11. Écrire une fonction `from_list : 'a list -> 'a slllist` qui étant donnée une liste OCaml, renvoie une liste `slllist` contenant les mêmes valeurs dans le même ordre. Par exemple, `from_list [1; 2; 3]` renverra

```
  Some
   {value = 1; next = Some {value = 2; next = Some {value = 3; next = None}}};
 last = Some {value = 3; next = None}}
```

Exercice 12. Écrire une fonction `to_list: 'a slllist -> 'a list` effectuant la traitement inverse de la fonction `from_list`.