TP #11 — Second contact avec OCaml ; exercices de stiles

Exercice 1. Habiter *des types 2*

Écrivez des expressions (les plus simples possibles et qui ne trichent pas (en cas de doute sur ce qui triche : demandez moi)) ayant pour type :

1. `'a * 'b -> 'a`
2. `'a * 'b -> 'b`
3. `'a -> 'b -> 'a * 'b`
4. `('a -> 'b -> 'c) -> 'a * 'b -> 'c`
5. `('a * 'b -> 'c) -> 'a -> 'b -> 'c`
6. `('a -> 'b * 'c) -> ('a -> 'b) * ('a -> 'c)`
7. `('a -> 'b) -> 'a * ('b -> 'c) -> 'c`
8. `(('a -> 'b) -> 'a) -> (('a -> 'b) -> 'a * 'b)`

Afin de facilement vérifier vos propositions (et ne pas les perdre dans les abîmes d'un interpréteur interactif), on conseille de télécharger puis modifier le fichier mp2i_tp11_ex1_strip.ml, en remplaçant les définitions des identifiants correspondant aux noms des questions par vos réponses.

Exercice 2. *Max dans une liste*

1. Écrivez une fonction `maxl : 'a list -> 'a` qui prend en entrée une liste supposée non vide et « renvoie » la valeur d'un élément maximum de cette liste (pour l'opérateur de comparaison par défaut « < »).
   Cette fonction devra utiliser les fonctions du *module* `List` décrites en cours (`List.is_empty`, etc.)
   (Il peut être intéressant d'introduire une ou plusieurs fonctions intermédiaires.)
2. Testez votre fonction avec des arguments de types variés, *via* une fonction de test dédiée. On pourra profiter pour l'écriture de cette fonction du fait qu'il est aisé en OCaml pour une fonction de prendre une autre fonction en argument. Par exemple, on suggère que cette fonction de test ait pour type : `('a list -> 'a) -> bool`

La manipulation des `list` OCaml uniquement à travers des fonctions est relativement lourde. En pratique, on lui préfère largement l'utilisation du filtrage de motif, typiquement par l'utilisation des deux motifs :

```
match v with
    | [] -> (* liste vide *)
    | x::xs -> (* liste non vide de tête x et queue xs *)
```

Le squelette d'une fonction suivant une telle approche est alors souvent :

```
let rec visit_list v =
    match v with
    | [] -> (* traite le cas d'une liste vide *)
    | x::xs -> if (* conditions impliquant x *)
               then (* ... *)
               else (* ... *)
```

De plus, si l'on sait (par exemple par les spécifications de la fonction) qu'un cas du filtrage est impossible, on peut lui associer une expression `assert false`

3. Écrivez une fonction `maxl_pm : 'a list -> 'a` de mêmes spécifications que `maxl` mais qui utilise le filtrage de motif à la place des fonctions du module `List`.
4. Testez. (Normalement, il suffit d'utiliser votre fonction de test déjà écrite *sans aucun changement*.)

Exercice 3. *Concaténation*

1. Écrivez une fonction `append : 'a list -> 'a list -> 'a list` qui prend en entrée deux listes `[a1 ; ... ; an]` et `[b1 ; ... bm]` et s'évalue en une nouvelle liste `[a1 ; ... ; an ; b1 ; ... ; bn]` contenant les éléments de son second argument à la fin de ceux de son premier, en préservant leur ordre.
2. Testez votre fonction.
3. Exprimez le coût de votre fonction en fonction des longueurs de ses deux arguments.
4. Soit deux listes `l1` et `l2` avec `l1` de taille constante et `l2` de taille linéaire en un certain paramètre n, on souhaite construire une liste `l3` contenant l'ensemble des éléments présents dans `l1` et `l2`, dans n'importe quel ordre. Donnez deux expressions utilisant `append` qui permettent de calculer une telle liste, ainsi que leurs coûts. Laquelle de ces expressions est préférable ?

La fonction implémentée dans cet exercice est déjà disponible en OCaml : il s'agit de `List.append` (qui n'est en principe pas à connaître), disponible à travers l'opérateur infixe `@` (qui est lui à connaître !!) : `l1 @ l2` doit s'évaluer en la même valeur qu'un appel `append l1 l2` avec votre fonction `append`. Cette fonction `List.append` n'est pas (significativement) plus efficace que votre propre fonction `append` (sauf si vous avez vous-même été particulièrement inefficace) : il convient donc de l'utiliser **uniquement quand cela est nécessaire, et au mieux**.

Exercice 4. *Miroir*

Le but de cet exercice est d'écrire une fonction `rev` qui s'évalue en le « miroir » de son argument : la fonction `rev` appelée avec `[a1 ; ... ; an]` comme argument doit s'évaluer en la (nouvelle) liste `[an ; ... ; a1]`.

1. Écrivez une fonction `rev : 'a list -> 'a list` respectant les spécifications ci-dessus. Cette fonction *peut* utiliser la concaténation de listes `@`.
2. Testez votre fonction.

3. Quel est son coût ?
4. Écrivez (au besoin) une fonction `rev' : 'a list -> 'a list` de mêmes spécifications que `rev`, qui n'utilise **pas** la concaténation `@`.
5. Testez votre fonction.
6. Quel est son coût ?

Exercice 5. *~~All the~~ iterators 1*

La bibliothèque standard OCaml fournit de nombreuses fonctions sur les `list`, au sein du *module* `List`. La documentation de ce module se trouve à la page https://ocaml.org/manual/5.4/api/List.html. Comme déjà vu pour (par ex. `List.hd`), il suffit pour utiliser une fonction de ce module de la préfixer par le nom du module, suivi d'un.

Un petit nombre de ces fonctions sont au programme : il s'agit de `mem`, `exists`, `for_all`, `filter`, `map` et `iter` ; vous devez être capable de les utiliser si l'on vous en fournit la documentation. Il peut aussi arriver qu'un sujet (ou un TP d'admission…) vous autorise à utiliser *toutes* les fonctions du module `List` sans nécessairement en fournir la documentation. Dans ce cas, il peut aussi être utile de connaître (et savoir utiliser) des fonctions comme `sort`, `rev`, `iteri`, `mapi`, `find`, `find_opt`, `fold_left`, `fold_right`…

Le but de cet exercice est d'implémenter par vous-même les fonctions explicitement au programme (à l'exception de `iter`, qui manque d'intérêt tant que nous n'avons pas abordé les effets en OCaml).

Beaucoup de ces fonctions sont d'*ordre supérieur*, et exploitent le fait qu'en OCaml il est possible (et facile) d'écrire une fonction dont l'argument est une fonction (qui pourra ensuite par exemple être définie comme une fonction anonyme par l'utilisateur ou utilisatrice).

Bien que cette consigne ne soit plus rappelée dans cet exercice, vous devez bien sûr **tester toutes vos fonctions** (par exemple en comparant vos résultats avec ceux des fonctions équivalentes du module `List`).

1. Écrivez une fonction `mem : 'a -> 'a list -> bool` telle que `mem e l` s'évalue à `true` ssi. `e` apparaît comme élément de `l`.
2. Écrivez une fonction `exists : ('a -> bool) -> 'a list -> bool` telle que `exists p l` s'évalue à `true` ss.'il existe un élément de `l` satisfaisant le *prédicat* `p`. Cette fonction `exists` doit être *paresseuse* : elle s'évalue en son résultat dès que celui-ci peut être déterminé. Autrement dit, si l'on note `[a1 ; ... ; an]` les éléments de `l`, `exists p l` s'évalue en le même résultat et a les mêmes effets que `(p a1) || (p a2) || ... || (p an)`.
   On peut remarquer que pour tout `p`, `exists p []` s'évalue à `false`.
3. Écrivez une fonction `for_all : ('a -> bool) -> 'a list -> bool` telle que `for_all p l` s'évalue à `true` ssi. tous les éléments de `l` satisfont le prédicat `p`. Cette fonction `for_all` doit être *paresseuse* : elle s'évalue en son résultat dès que celui-ci peut être déterminé. Autrement dit, si l'on note `[a1`

`; ... ; an]` les éléments de `l`, `for_all p l` s'évalue en le même résultat et a les mêmes effets que `(p a1) && (p a2) && ... && (p an)`.
On peut remarquer que (par les propriétés magiques de l'ensemble vide) l'on a que pour tout `p`, `for_all p []` s'évalue à `true`.

4. Écrivez une fonction :
   `filter : ('a -> bool) -> 'a list -> 'a list`
   telle que `filter p l` s'évalue en une nouvelle liste contenant (dans le même ordre) exactement les éléments de `l` qui satisfont le prédicat `p`.
5. Soit un prédicat `p`, une liste `l`, écrivez une expression non-triviale qui fait intervenir `p`, `l` et les fonctions `for_all` et `filter`, et qui s'évalue toujours à `true`.
6. Écrivez une fonction `map : ('a -> 'b) -> 'a list -> 'b list` telle que si l'on note `[a1 ; ... ; an]` les éléments d'une liste `l`, `map f l` s'évalue en la liste `[f a1 ; ... ; f an]` obtenue en appliquant `f` à chacun des éléments de `l`.
7. Écrivez des variantes non-paresseuses `exists'` et `for_all'` de `exists` et `for_all` en utilisant `map` et `mem`. Quels impacts peut avoir cette non-paresse en terme de coût et d'effets (un exemple d'effets étant une erreur fatale à l'exécution).
8. Écrivez une fonction `init : int -> (int -> 'a) -> 'a list` telle que pour un entier `n` supposé non négatif, `init n f` s'évalue en la liste des `n` éléments `[f 0 ; ... ; f (n - 1)]`.

   La fonction de même nom du module `List` satisfait la contrainte supplémentaire que les évaluations de `f` se font « de gauche à droite », mais nous l'ignorerons ici.
9. Écrivez une fonction :
   `find_opt : ('a -> bool) -> 'a list -> 'a option`
   telle que `find_opt p l` s'évalue en `Some e` avec `e` le premier élément de `l` satisfaisant le prédicat `p`, et `None` si aucun tel élément n'existe.
10. Écrivez une fonction :
    `filter_map : ('a -> 'b option) -> 'a list -> 'b list`
    telle que `filter_map f l` s'évalue en une nouvelle liste obtenue en évaluant `f` sur les éléments de `l` (dans l'ordre), filtrant ceux pour lesquels l'évaluation est `None`, et incluant `v` pour ceux dont l'évaluation est `Some v`.
11. Une question du sujet d'option informatique MP du concours commun centrale supélec 2020 demandait indirectement d'écrire une fonction `ue` de trois paramètres `int` `i`, `j` et `n` construisant une liste de deux listes `[i,j] @ sij` et `rsij @ [i,j]`, avec `i` et `j` deux entiers $\in [\![0, n-1]\!]$; `sij` la liste ordonnée des `n` entiers `0...n - 1` dont on a retiré les entiers `i` et `j` ; et `rsij` la liste miroir de `sij`.
    Écrivez une telle fonction `ue` en utilisant les fonctions du module `List` (ou vos propres implémentations de ces fonctions).