Université de Bordeaux
Année 2019-2020

Mastère 2
T.A.P

TD n°4 - Généricité et 1ère classe

Exercice 1: Curryfication et généricité

Dans cet exercice, on s'intéresse à la construction de fonctions génériques. En OCaml, les types de ces fonctions contiennent des types génériques de la forme `'a` qui peuvent être remplacés par n'importe quel type du langage.

1. Implémenter une fonction qui prend en argument un triplet et qui renvoie le deuxième élément de ce triplet.
2. En utilisant la construction de fonctions anonymes avec le mot-clé `fun`, implémenter la fonction qui "curryifie" une fonction *binaire* quelconque et celle qui la "décurryfie". Pour rappel :

   Exemple de forme curryfiée :

   ```
   f : int → int → int
   let plus x y = x+y;;
   plus 2 3;;  (* → 5 *)
   ```

   Exemple de forme décurryfiée :

   ```
   g : (int*int) → int
   let fois (x,y) = x*y;;
   fois (2,3);; (* → 6 *)
   ```

3. Écrire la fonction qui permet d'inverser l'ordre des arguments d'une fonction binaire. Quel peut être l'intérêt d'une telle fonction ?
4. Implémenter la fonction `iterate` prenant en argument une fonction f et un nombre n, et qui renvoie l'itérée n-ème de f, calculée comme :

   $$f^{(n)} = f \circ f \circ \ldots \circ f$$

   Indice : réfléchir d'abord au type de la fonction f
   Discuter le type de cette fonction, et les liens entre les paramètres de type.

▷ La *reconnaissance de motif* est une construction d'un langage de programmation permettant, étant donnée une expression, de faire un branchement du code selon la forme de cette expression, cette forme étant exprimée par des *motifs*. Dans le code suivant calculant la longueur d'une liste, l'expression `l` est comparée aux motifs `[]` (liste vide) et `x::xs` (liste avec un élément en tête) :

```
let rec length l = match l with  (* Compare l to              *)
    | []      → 0                (* the empty list           *)
    | (x::xs) → 1 + length xs    (* a list such that List.hd l = x  *)
                                 (*           and List.tl l = xs *)
```

De plus, les sous-expressions correspondant à ces motifs peuvent être réutilisés dans le code (ici `xs` permet de calculer le reste de la longueur de la liste).

Exercice 2: Quelques pliages sur les listes

Considérons les exemples de code suivants sur les listes :

```
let rec sum l = match l with
  | []    → 0
  | x::xs → x + (sum xs);;

let rec length l = match l with
  | []    → 0
  | _::xs → 1 + (length xs);;

let maximum l =
  let rec max_rec l m = match l with
    | []    → m
    | x::xs → max_rec xs (max m x)
  in max_rec l (List.hd l);;
```

Ces différents algorithmes appliquent le même processus de parcours d'une liste, en appliquant une transformation locale binaire (respectivement `+`, `+` et `max`) tout au long de la liste. Ce processus de parcours de la liste est le même pour chacun des trois codes ci-dessus, et se nomme un *pliage*. En OCaml, il est possible de séparer le pliage de la transformation effectuée, en utilisant les fonctions `fold`.

1. Examiner le type des fonctions `fold_left` et `fold_right` dans le module `List`. Déterminer, pour chacune d'entre elles, la transformation qu'elles permettent d'effectuer lorsqu'appliquées à une fonction `f`, un élément de départ `init` et une liste `[e1;...en]`.
2. Transformer les fonctions `sum`, `length` et `maximum` en les ramenant à un pliage.
3. Écrire, à l'aide d'un pliage, une fonction `list_or` qui prend en entrée une liste de booléens et renvoie vrai si et seulement si au moins l'un des éléments de la liste est vrai.

Exercice 3: Cache fonctionnel en Java

▷ Pour réaliser l'entièreté de cet exercice, il faut utiliser une version de Java ≥ 9.

Dans cet exercice, on considère le contexte d'un magasin de guitares proposant un inventaire d'instruments disponibles à la vente, chaque instrument ayant un nombre de caractéristiques différentes :

— le nom du modèle (de type `String`)

— son prix (de type (`double`)

— sa marque (de type `Trademark`, un enum)

— une distinction entre acoustique et électrique (de type `Kind`, un enum)

— plusieurs types de bois (de type `Wood`, un enum)

Le magasin désire permettre à ses clients de faire des requêtes dans l'inventaire, et permettre autant de types de requêtes que possible. Mieux, il voudrait une architecture qui permette d'ajouter facilement de nouveaux types de requêtes, et ultimement de pouvoir les sauvegarder selon les demandes des clients. Actuellement, les requêtes sont faites de la manière suivante dans une fonction `search` :

```
List<Guitar> matchingGuitars = new ArrayList<>();
for (Guitar guitar : guitars) {
    if (Kind.isSame(guitar.getKind(), Kind.ELECTRIC)
        && Guitar.isSameModel(guitar, "Stratocastor"))
        matchingGuitars.add(guitar);
}
```

Pour gérer la notion de requête, une possibilité consiste à construire une interface fonctionnelle `Criterion` qui représente un (ou plusieurs) critères de recherche sur l'inventaire. Par exemple, sélectionner les guitares acoustiques se ferait à partir de la lambda-expression :

```
(g) → { return Kind.isSame(g.getKind(), Kind.ACOUSTIC); }
```

1. Expliquer pourquoi il n'est pas désirable de créer un critère de recherche par classe de caractéristique (`Kind`, `Trademark`, `Wood` ...)
2. Proposer une interface fonctionnelle `Criterion` contenant une seule méthode abstraite permettant de sélectionner une guitare.
3. Modifier la fonction `search` de façon à prendre en paramètre une liste de `Criterion` et renvoyer la liste des `Guitar` correspondante.
4. Ajouter à la classe `Criterion` une méthode `and` qui permette de construire une requête par conjonction de requêtes, et adapter la fonction `search` pour qu'elle prenne en paramètre une valeur de type `Criterion`.

À partir de maintenant, on se pose la question d'organiser les manières de construire les `Criterion`. En effet, il est assez simple de voir qu'un certain nombre de requêtes peuvent être construites de manière générique. Par exemple :

— tester si une guitare est acoustique ou électrique ;

— tester si une guitare a une marque particulière, comme `"Fender"` ;

— tester si une guitare a un prix inférieur à une valeur donnée ...

Pour résoudre cela, on propose de mettre en place une classe `CriterionFactory` qui contienne des méthodes de fabrique de `Criterion` génériques.

5. Quel est le prototype d'une telle fonction ?
6. Ecrire les deux méthodes de fabrique correspondant aux deux premiers exemples ci-dessus (`Kind` et `Trademark`)

Rapidement, on se rend compte que la fabrique va contenir un ensemble potentiellement important de méthodes de fabrique. De plus, il pourrait être envisageable d'autoriser à rajouter d'autres méthodes de fabrique dynamiquement. Ainsi, on propose de ranger toutes les méthodes de fabrique dans une table de hachage, et de les identifier par une chaîne de caractères. La requête initiale serait identifiée par `"Kind"` et construite ainsi :

```
CriterionFactory.makeCriterion("Kind", Kind.ELECTRIC).
```

7. Quel serait le type de cette table de hachage ?
8. En utilisant une initialisation de la table de hachage ressemblant à la forme suivante, ajouter les deux méthodes de fabrique précédentes à l'intérieur de cette table.

```
private static Map<String, Double> cache = new HashMap<>(
 Map.ofEntries(
               Map.entry("Trademark", Double.valueOf(7)),
               Map.entry("Kind", Double.valueOf(8)),
               Map.entry("Model", Double.valueOf(9))
));
```

9. Adapter le code de génération des critères de recherche à cette modification du code.
10. Comment faire pour permettre d'ajouter dynamiquement de nouvelles méthodes de fabrique, comme le dernier exemple de la liste ci-dessus (`Price`) ?