TD 2 : Types algébriques

Pierre Talbot (pierre.talbot@univ-nantes.fr)

10 avril 2019

Objectif(s)

- ✯ Utiliser et comprendre les types algébriques : type somme et enregistrement.
- ✯ Modélisation de programme avec le paradigme fonctionnel.

Certains exercices sont largement inspirés d'exercices du cours OCaml de Charlotte Truchet et d'Emmanuel Chailloux.

Exercice 1 – Jeu de Belote

1. Définir un type `couleur` pour représenter les couleurs d'un jeu de cartes.
2. Définir un type `carte` pour représenter les cartes d'un jeu de belote : as, roi, dame, valet et petites cartes.
3. Écrire une fonction `valeur` qui prend la couleur de l'atout et une carte et renvoie la valeur de la carte. On rappelle les règles de comptage suivantes : l'as vaut 11, le roi 4, la dame 3, le valet 20 s'il a la couleur de l'atout, 2 sinon, le 10 vaut 10, le 9 vaut 14 s'il a la couleur de l'atout, 0 sinon, les autres cartes valent 0.
4. Tester la fonction précédente sur le jeu suivant : valet de carreau, 10 de trèfle, 7 de cœur, 8 de carreau, 9 de pique avec atout à carreau (on doit trouver 30) .

Exercice 2 – Liste d'entiers : `iter, map, fold`

On se motive pour coder le type des listes d'entiers et des fonctions associées permettant de les manipuler.

1. En utilisant le type somme, définir le type d'une liste dont chaque nœud porte un entier.
2. Définir les fonctions suivantes :

```
(* Liste vide. *)
empty: int_list
(* Ajoute un entier dans la liste. *)
cons: int -> int_list -> int_list
(* Taille de la liste *)
length : int_list -> int
```

3. On aimerait maintenant modifier et parcourir cette liste. Pour cela deux opérations principales `map` et `iter` :

```
map: int_list -> (int -> int) -> int_list
iter: int_list -> (int -> unit) -> unit
```

Voici un exemple de comportement que vous devez obtenir lorsque vous avez défini ces deux fonctions :

```
(* Create a list of 2 elements. *)
let l = (cons 2 (cons 3 empty)) in
(* Add 1 to every element of the list. *)
let l' = map l (fun x -> x + 1) in
(* Print the list. *)
iter l (Printf.printf "%d")
```

4. Une autre operation très importante est le `fold_left` :

   ```
   fold_left: (int -> int -> int) -> int -> int_list -> int
   ```

   Voici un exemple de comportement que vous devez obtenir lorsque vous avez défini cette fonction :

   ```
   (* Create a list of 2 elements. *)
   let l = (cons 2 (cons 3 empty)) in
   (* Sum the elements in the list. *)
   let sum = fold_left l (fun acc x -> acc + x) 0 in
   Printf.printf "%d\n" sum
   ```

   On peut aussi noter la somme de manière plus concise avec `let sum = fold_left l + 0`

Exercice 3 – @, : :, [ ]

Utiliser le type `list` standard de OCaml ainsi que ses raccourcis syntaxiques.

1. Implémenter la fonction `zip` qui prend en paramètre deux listes `l1`, `l2` et retourne une liste de couples tel que pour des listes en entrée `[x1; x2; ...; xn]` et `[y1; y2; ...; yn]` retourne la liste `[(x1, y1); (x2, y2); ...; (xn, yn)]`. Dans le cas où une des deux listes est plus courte que l'autre, la liste résultante aura la taille de la plus petite et les données en trop seront ignorées.
2. Dites si les codes suivant compilent et s'ils se comportent comme ce que leur auteur a voulu écrire. Si non, corrigez-les.

```
let equal_trois = function
    3 -> true
  | _ -> false

let equal_x x = function
  n -> true
| _ -> false

let double_zero n m =
  match n m with
  | 0 0 -> true
  | _ -> false

let rec sorted = function
  [] -> true
| x :: y :: q -> x < y && sorted q

let rec interleave l sep =
  match l with
    | s :: l -> s ^ sep ^ interleave l sep
    | s :: [] -> s
    | [] -> ""
```

Exercice 4 – Tri fusion

On va implémenter une fonction qui permet de trier une liste grâce au tri fusion (*merge sort*).

1. Écrire une fonction `split` prenant une liste d'entiers `l` et qui retourne cette liste coupée en deux (i.e. un couple de liste d'entier).
2. Écrire une fonction `merge` prenant deux listes d'entiers `l1` et `l2` supposées triées, et retourne la liste triée issue de la fusion de ces deux listes.
3. Écrire la fonction `sort` de tri par fusion (veiller à bien traiter les cas de base en premier).
4. Quelle est la complexité de ce tri ?

Exercice 5 – Jeu de rôles

Nous allons implanter un petit jeu de type RPG dans lequel le joueur a une classe, combat des monstres et ramasse leurs possessions lorsqu'il les a vaincus.

1. Un personnage a une classe, (archer, barbare ou magicien), un nombre de points d'expérience, et un sac contenant divers objets en diverses quantités. Les objets peuvent être des pièces de monnaie, des cuisses de poulet ou des éponges.

   Définissez les types en conséquence.
2. Un monstre a une race (golem, sanglier ou moustique) et possède un objet dans sa poche que le personnage pourra ramasser triomphalement. On considérera des armées de moustiques en nombre variable comme un seul monstre.

   Définissez les types en conséquence.
3. Écrivez la fonction ajoutant un objet au sac.
4. Écrivez la fonction affichant le contenu du sac. On demande un affichage sous la forme suivante :

   ```
   2 eponges
   1 poulet
   2 pieces
   ```
5. Écrivez une fonction générant aléatoirement un monstre. Vous pouvez utiliser la fonction `Random.int` qui prend un paramètre entier n et renvoie un nombre aléatoire entre 0 et n.

   Pour obtenir des tirages pseudo-aléatoires différents entre deux lancement du programme, vous pouvez utiliser la fonction `Random.self_init ()` une fois au début du programme.
6. Écrivez la fonction `frappe` calculant le nombre de points de vie enlevés à l'adversaire lors d'une attaque. Un barbare fait 10 points de dégâts, un archer 4 et un magicien 5. Cependant, avant de faire des dégâts, il doit réussir à toucher sa cible. Un barbare a 30% de chances de toucher, un magicien 50% et un archer 70%. Le personnage gagne 5% de bonus au toucher par point d'expérience.
7. Écrivez la fonction `frappe_monstre` sachant qu'un golem fait 4 points de dégâts, chaque moustique d'une armée $1/2$ point et un sanglier 2 points.
8. Définissez une exception `Mort` à lever quand le personnage meurt.
9. Écrivez la fonction `combat` effectuant un combat entre un personnage p et un monstre m. Chacun des deux commence à 20 points de vie. Si le personnage ne meurt pas, il gagne cinq points d'expérience, ainsi que l'objet contenu dans la poche du monstre.
10. Écrivez la fonction `malheureuse_rencontre`, qui génère un monstre aléatoirement et le fait s'affronter avec le personnage. La fonction devra aussi afficher le monstre généré afin d'admirer les exploits de votre valeureux guerrier.
11. Écrivez la boucle principale du jeu, réalisant un nombre défini de combats avant d'afficher le butin final, à moins que le personnage ait failli dans sa quête, auquel cas vous devrez l'afficher.

    Exemple de partie :

    ```
    Vous tombez nez à nez avec une armée de 7 moustiques !
    Vous tombez nez à nez avec un terrRRrrible golem !
    Vous tombez nez à nez avec un terrRRrrible golem !
    Vous êtes mort !.
    ```

ou bien

```
Vous tombez nez à nez avec une armée de 7 moustiques !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous tombez nez à nez avec une armée de 6 moustiques !
Vous tombez nez à nez avec un terrRRrrible golem !
Vous tombez nez à nez avec un terrRRrrible golem !
Bravo, vous avez vaincu !
Voici votre butin :
2 eponges
1 poulet
2 pieces
```