Université de Bordeaux
Année 2016-2017

Mastère 2
T.A.P

TD n°6 - Types inductifs
Programmation orientée par les données

Exercice 1: Types inductifs – Filtrage

Dans cet exercice, on introduit les *types inductifs* en OCaml. On définit un tel type à l'aide de la syntaxe suivante :

```
type ⟨nom_type⟩ =
  | ⟨Constr_1⟩ of ⟨Type_1⟩
  | ⟨Constr_2⟩ of ⟨Type_2⟩
    ...
  | ⟨Constr_n⟩ of ⟨Type_n⟩
```

Exemple :

```
type value =
  | Int of int
  | Float of float;;
```

Pour utiliser les types inductifs, on a recours à un procédé appelé *filtrage* ou *reconnaissance de motifs* (qui peut être utilisé avec d'autres types que les types inductifs) :

```
match ⟨expr⟩ with
  | ⟨p_1⟩ → ⟨expr_1⟩
  | ⟨p_2⟩ → ⟨expr_2⟩
    ...
  | ⟨p_n⟩ → ⟨expr_n⟩
```

Exemple :

```
let string_of_value x =
 match x with
  | Int i   → string_of_int i;
  | Float f → string_of_float f;;
```

Noter que la reconnaissance de motifs permet de filtrer des expressions non triviales comme des listes (par exemple `x::c`) ou des vecteurs (par exemple `(a,b)`) ou bien simplement des expressions constantes (par exemple `Zero`).

1. Écrire un type inductif `weight` permettant de manipuler des poids en kilos, en livres et en carats.

2. Écrire les fonctions de traduction d'un élément de type `weight` en kilos. (*Indice culturel :* 1kg = 2.205 lbs = 5000 carats)

Le compilateur sait vérifier si une reconnaissance de motifs est exhaustive ou non.

3. Tester l'exemple suivant et le corriger :

```
let rec sum_of_values a b = match (a,b) with
  | (Int i,Int j)     → Int(i+j)
  | (Int i,Float f)   → Float((float i)+.f)
  | (Float f,Float g) → Float (f+.g);;
```

4. Écrire un type inductif permettant de manipuler des pièces de monnaie de 1, 7 et 13 zlotys. Écrire un alorithme glouton qui, étant donné un nombre de zlotys n, renvoie une liste de pièces (en zlotys) dont la somme des valeurs est n.

5. Écrire un type inductif permettant de regrouper les types réels et complexes. Inclure ce type dans un autre type inductif permettant de manipuler les solutions des équations des trinômes du second degré, ainsi que la fonction de résolution.

Exercice 2: Arbres

Considérons un type inductif permettant de représenter des arbres binaires. Ce type est fourni avec une fonction `bintree_build` permettant de construire un arbre binaire de hauteur donnée, étant donné :

- une valeur à la racine et
- une fonction `f : 'a → ('a * 'a)` qui, étant donnée la valeur stockée dans un noeud, fournit le couple des valeurs stockées dans les deux noeuds fils.

```
type 'a bintree =
| BinEmpty
| BinNode of 'a * 'a bintree * 'a bintree

let rec bintree_build f h x =
  if (h <= 0)
  then Empty
  else let (x1,x2) = f(x) in
          Node (x,
            (bintree_build f (h-1) x1),
            (bintree_build f (h-1) x2));;
```

1. Tester `bintree_build` en utilisant le fonction de génération `let f x = (2*x,2*x+1)`.

2. Écrire une fonction `bintree_map` qui permet d'appliquer uniformément une fonction `f` à tous les éléments d'un arbre binaire.

Dans le cas des arbres binaires ordonnés, les éléments de l'arbre sont des entiers, et étant donné un noeud x, tous les éléments stockés dans le sous-arbre gauche sont inférieurs à x, tous les éléments stockés dans le sous-arbre droit lui sont supérieurs.

3. On s'intéresse à la fonction `bintree_insert` permettant d'insérer un nombre entier dans un arbre de manière à préserver la propriété d'ordre. Écrire une telle fonction.

Booch, dans *Object-oriented analysis and design*, propose la définition suivante :

> **Persistence** is the property of an object through which its existence transcends **time** (i.e. the object continues to exist after its creator ceases to exist) and/or **space** (i.e. the objects location moves from the address space in which it was created).

4. Comment fonctionne la persistance lors de l'exécution de l'algorithme d'insertion d'un nombre dans un arbre ?

Plaçons nous maintenant dans le cas où les noeuds ne possèdent plus un nombre fixé, mais un nombre arbitraire de fils.

5. Écrire un type adapté à de tels arbres, et construire la fonction `tree_build` associée.
6. Écrire un itérateur `tree_map` permettant d'appliquer une fonction `f` uniformément sur tous les éléments de l'arbre.
7. Comment faire pour que cette fonction soit récursive terminale ?

Exercice 3: Méthode du gradient

Reprenons la définition de type sur les graphes vue précédemment :

```
type 'k key = K of 'k
and ('k, 'a) vertex = 'k key * 'a
and ('k, 'a) graph = Graph of (('k, 'a) vertex * 'k key list) list;;
```

1. Écrire une fonction énumérant les sommets d'un graphe selon un parcours en profondeur.
2. Écrire une fonction énumérant les sommets d'un graphe selon un parcours en largeur.

Les deux fonctions précédentes ont effectivement beaucoup de code en commun. On se propose de factoriser les parcours sur les graphes en utilisant une stratégie, c'est-à-dire une fonction d'une stratégie définie par une fonction qui décide, étant donné le sommet courant, la liste de ses voisins et les sommets restant à parcourir, la façon dont ces voisins vont s'adjoindre à la liste restant à parcourir.

```
type 'a strategy = Strategy of ('a → 'a list → 'a list → 'a list)
```

3. Écrire un parcours de graphe qui soit générique.

Pour illustrer ces parcours de graphe, on s'intéresse à une méthode apparentée à la *méthode du gradient*, qui consiste à rechercher un élément localement minimal dans un graphe.

4. Écrire une fonction `until_reached` qui applique une stratégie `s` lors du parcours d'un graphe, jusqu'à ce qu'une fonction d'arrêt (dépendant uniquement du sommet et du graphe) décide d'arrêter le parcours.

5. Écrire la fonction d'arrêt permettant de s'arrêter lors du parcours d'un graphe lorsque l'élément courant est plus petit que tous ses voisins.

Exercice 4: Files d'attentes fonctionnelles

Considérons l'implémentation suivante de files d'attente fonctionnelles[1]. Un type associé à une file d'attente est défini par :

```
type 'a queue = Q of 'a list * 'a list;;
```

Une *file d'attente* est une structure de donnée dans laquelle on peut insérer des éléments, et les faire sortir dans l'ordre dans lequel ils ont été insérés. L'implémentation utilise deux listes : une liste d'entrée dans laquelle on insère les éléments et une liste de sortie à partir de laquelle on les sort. Lorsqu'il n'y a plus d'éléments dans la liste de sortie, on remplace la liste de sortie par la liste d'entrée *retournée*. Par exemple :

État de la file	Liste d'entrée	Liste de sortie
File vide	`[]`	`[]`
Insertion de 1	`[1]`	`[]`
Insertion de 2	`[2;1]`	`[]`
Retournement	`[]`	`[1;2]`
Ejection	`[]`	`[2]`
Insertion de 3	`[3]`	`[2]`

Idéalement, l'opération de retournement est effectuée uniquement lorsque l'on doit éjecter un élément, et que la liste de sortie est vide. Les opérations d'insertion et d'éjection sont naturellement en temps constant.

1. Écrire une fonction `q_insert` qui insère un élément dans une file d'attente fonctionnelle.

   Remarque : vu la façon dont les files d'attente sont définies, on peut écrire ces fonctions en utilisant la reconnaissance de motifs de la manière suivante :

[1] Cf. *"An efficient functional implementation of FIFO queues"* de F. W. Burton (1982).

```
let q_fun  (Q (l1, l2)) = l1@l2
```

2. Écrire une fonction `q_transfer` qui effectue l'opération de retournement, c'est-à-dire retourne la première liste dans la seconde, et ceci uniquement lorsque la seconde liste est vide.

3. Écrire la fonction `q_pop` qui éjecte un élément de la file d'attente. La fonction générera une exception avec la commande `failwith` si la liste est vide.

Exercice 5: Interprète `BASIC`

Une utilisation habituelle des types inductifs consiste à représenter des grammaires, qui à leur tour permettent d'écrire des compilateurs. Sur la page `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/book-ora059.html` se trouve un exemple d'interpréteur du langage `BASIC`.

Pour nous simplifier la vie, nous allons alléger la grammaire utilisée. Dans un premier temps, les expressions du langage seront de la forme :

```
type op_bin = PLUS | MOINS | EQUAL | MULT | DIV;;
type expression =
| ExpInt of int
| ExpVar of string
| ExpBin of expression * op_bin * expression  ;;
```

On ne gère que des expressions numériques ou booléennes. Les valeurs gérées par l'environnement seront du type :

```
type valeur = Vint of int | VBool of bool;;
```

1. Comment gérer l'environnement dans lequel sont évaluées les expressions ?

2. Écrire une fonction qui prend en argument une expression et un environnement et qui renvoie la valeur correspondant à cette expression (en remplaçant les variables par les valeurs qu'elles possèdent dans l'environnement).

La syntaxe des expressions du langage se résume à :

```
type instruction =
| Goto of int
| Print of expression
| Input of string
| If of expression * int
| Let of string * expression  ;;
type ligne = { num : int ; inst : instruction }  ;;
type program = ligne list  ;;
```

Nous allons tenter d'écrire un mini-interpréteur de ce langage. Pour cela, nous allons utiliser les définitions suivantes pour les valeurs et l'environnement. Remarquer que les instructions conditionnelles `IF` sont obligatoirement suivies d'un `GOTO`. De plus, les lignes sont numérotées de 10 en 10.

3. Écrire une fonction qui prend en argument un programme, une ligne et un environnement, et renvoie le numéro de ligne correspondant à l'instruction suivante, ainsi que l'environnement modifié par le traitement de cette instruction.

Voilà un petit programme `BASIC` qui calcule la factorielle d'un entier :

```
10 INPUT N
20 LET I = 1
30 LET S = 1
40 LET I = I + 1
50 LET S = S * I
60 IF (I = N) THEN GOTO 80
70 GOTO 40
80 PRINT S

let prog = [
  { num = 10; inst = Input "n" };
  { num = 20; inst = Let ("i",ExpInt 1) };
  { num = 30; inst = Let ("s",ExpInt 1) };
  { num = 40; inst = Let ("i",(ExpBin ((ExpVar "i"),PLUS,(ExpInt 1)))) };
  { num = 50; inst = Let ("s",(ExpBin ((ExpVar "s"),MULT,(ExpVar "i")))) };
  { num = 60; inst = If ((ExpBin ((ExpVar "i"),EQUAL,(ExpVar "n"))),80) };
  { num = 70; inst = Goto 40 };
  { num = 80; inst = Print (ExpVar "s") };
];;
```

4. Écrire une fonction permettant de simuler le fonctionnement d'un programme et l'appliquer à ce programme.

Les types inductifs du langage OCaml permettent de faciliter l'écriture de cet interpréteur. La question se pose alors de savoir quels problèmes se posent dans un langage différent.

5. Reprendre les 4 premières questions en langage C#, en adaptant les types utilisés pour qu'ils correspondent au langage.

6. Quelles sont les avantages et inconvénients de la programmation utilisant des types inductifs en C# par rapport à OCaml ?