Université de Bordeaux
Année 2017-2018

Mastère 2
T.A.P

TD n°5 - Types de données et programmation fonctionnelle

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 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.

```
type weight =
  | Kilo of float
  | Livre of float
  | Carat of float;;

let weight_to_float w = match w with
  | Kilo f  → f
  | Livre f → f/.2.205
  | Carat f → f/.5000.;;

type zloty = One | Seven | Thirteen;;

let rec petty_cash n =
  if (n>=13) then Thirteen::(petty_cash (n-13))
  else if (n>7) then Seven::(petty_cash (n-7))
  else if (n>=1) then One::(petty_cash (n-1))
  else [];;

petty_cash 254;;

type numeric  = Real of float | Complex of (float*float)
type solution = One of numeric | TwoReal of (numeric*numeric)
              | TwoComplex of (numeric*numeric)

(* Finds the roots of a trinom ax^2+bx+c *)
let trinom_root (a,b,c) =
  let delta = b*.b -. 4.*.a*.c in
  if (delta > 0.) then
    TwoReal (Real ((-.b+.sqrt(delta))/.(2.*.a)),
             Real ((-.b-.sqrt(delta))/.(2.*.a)))
  else if (delta < 0.) then
    TwoComplex (Complex ((-.b/.(2.*.a), sqrt(delta)/.(2.*.a))),
                Complex ((-.b/.(2.*.a), -.sqrt(delta)/.(2.*.a))))
  else One (Real (-.b/.2./.a));;

trinom_root (1.,2.,1.);;

(* Here the type 'solution' does not ensure that, for example, the
   TwoReal constructors contains effectively real values (in fact they
   could be complex). A more precise choice for types could be : *)

type real = Real of float
type complex = Complex of (float*float)
type solution = One of real | TwoReal of (real*real)
              | TwoComplex of (complex*complex)
```

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 :

1. une valeur à la racine et
2. 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 BinEmpty
  else let (x1,x2) = f(x) in
         BinNode (x,
           (bintree_build f (h-1) x1),
           (bintree_build f (h-1) x2));;
```

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

4. 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.

5. Écrire une fonction `bintree_insert` permettant d'insérer un nombre entier dans un arbre de manière à préserver la propriété d'ordre.

6. Dans une représentation des arbres avec des pointeurs, comment pourrait-on implémenter cette fonction en créant un minimum de nouveaux objets (et dont réutilisant les valeurs passées en paramètre) ?

7. Proposer une fonction `bintree_fold`.

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

8. Écrire un type adapté à de tels arbres, et construire la fonction `tree_build` associée.

9. Écrire un itérateur `tree_map` permettant d'appliquer une fonction `f` uniformément sur tous les éléments de l'arbre.

10. Comment faire pour que cette fonction soit récursive terminale ?

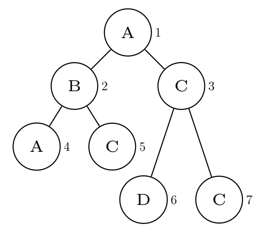
```
(***** Binary trees *****)
type 'a bintree =
  | BinEmpty
  | BinNode of ('a * 'a bintree * 'a bintree)

let rec bintree_build f h x =
  if (h <= 0)
  then BinEmpty
  else let (x1,x2) = f(x) in
          BinNode (x,
            (bintree_build f (h-1) x1),
            (bintree_build f (h-1) x2));;
let bintree_interv h =
  bintree_build (fun x -> (2*x,2*x+1)) h 1;;

let t1 = bintree_interv 3;;

let rec bintree_insert t x = match t with
  | BinEmpty           -> BinNode(x, BinEmpty, BinEmpty)
  | BinNode(y, tl, tr) -> if (y < x)
                 then BinNode(y, tl, bintree_insert tr x)
                 else BinNode(y, bintree_insert tl x, tr)

(* Successively insert the values into the tree *)
let t2 = List.fold_left bintree_insert BinEmpty [4;1;6;0;3;5;7];;

let rec bintree_fold f e t = match t with
  | BinEmpty           -> e
  | BinNode(y, tl, tr) -> f y (bintree_fold f e tl)
                     (bintree_fold f e tr);;

bintree_fold (fun el x y -> el + x + y) 0 t1;; (* sum of values *)
bintree_fold (fun el x y -> el::x@y) [] t2;;    (* tree to  list *)

(***** Arbitrary trees *****)
type 'a tree =
  | TEmpty
  | TNode of 'a * 'a tree list;;

(* Map -- not tail-recursive *)
let rec tree_map f tr = match tr with
  | TEmpty     -> TEmpty
  | TNode(a,l) -> TNode (f(a),List.map (tree_map f) l);;

let rec tree_build f n rac =
  match n with
    | 0 -> TNode(rac,[])
    | _ -> let lrac = f(rac) in
      let ltree = List.map (tree_build f (n-1)) lrac in
        TNode(rac,ltree);;

let n = 11;;
let t = tree_build (fun x -> [x;x+1]) n 1;;
tree_map (float) t;;
```

```
(* Map - tail-recursive / long *)
let rec int_list n =
  let rec int_list_rec n l = match n with
    | 0 → []
    | 1 → 1::l
    | _ → (int_list_rec (n-1) (n::l))
  in int_list_rec n [];;

int_list 15;; (* [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15] *)

let list_concat l1 l2 =
  let rec list_concat_rec l1 l2 r = match (l1,l2) with
    | ([],[])  → List.rev r;
    | (x::c,_) → list_concat_rec c l2 (x::r)
    | (_,y::d) → list_concat_rec l1 d (y::r)
  in list_concat_rec l1 l2 [];;

list_concat [1;2;3] [4;5;6];;

let rec enum_paths tr =
  let rec parcours to_visit visited =
    if (List.length to_visit) = 0 then (List.rev visited)
    else let (p,t) = (List.hd to_visit) in match t with
      | TEmpty     → parcours (List.tl to_visit) visited
      | TNode(a,l) →
          let patl = int_list (List.length l) in
          let newl = List.map2 (fun x y → ((list_concat p [x]),y)) patl l in
            parcours (list_concat (List.tl to_visit) newl) ((p,a)::visited)
  in parcours [([],tr)] [];;

let rec replace_nth l n e =
  let rec replace_rec l n e r =
    match n with
      | 1 → list_concat (List.rev r) (e::(List.tl l))
      | _ → replace_rec (List.tl l) (n-1) e ((List.hd l)::r)
  in replace_rec l n e [];;

replace_nth [1;2;3;4] 3 8;;

let rec is_prefix l1 l2 = match (l1,l2) with
  | ([],_)      → true
  | (x::c,y::d) → (x=y) && is_prefix c d
  | _           → false;;

is_prefix [1;3] [1;2;4];;

let remove_last l = List.rev(List.tl(List.rev l));;
let remove_and_get l x =
  let rec remove_and_get_rec l x rl rx = match l with
    | []       → ((List.rev rl),rx)
    | (a,b)::c → if (b=x) then remove_and_get_rec c x rl (a::rx)
      else remove_and_get_rec c x ((a,b)::rl) rx
  in remove_and_get_rec l x [] [];;

remove_and_get [("a",1);("b",1);("c",5)] 1;;
remove_and_get [("a",1);("b",3);("c",5)] 2;;
let rec insert_along_paths pathlist treelist = match pathlist with
  | []               → failwith "Malformed_graph";
  | ([],value)::e    →
      let (finaltreelist,sons) = remove_and_get treelist [] in
        TNode(value,(List.rev sons));
  | (path,value)::c →
      let (newtreelist,sons) = remove_and_get treelist path in
      let newpath = remove_last path in
        insert_along_paths c ((TNode(value,(List.rev sons)),newpath)::newtreelist);;

let tr = TNode(1,
               [TNode(2,
                       [TNode(4,[]);TNode(5,[])]);
                TNode(3,[])]);;

enum_paths tr;;
insert_along_paths (List.rev (enum_paths tr)) [];;

let rec list_map_second f l r = match l with
  | []       → (List.rev r)
  | (a,b)::c → list_map_second f c ((a,f(b))::r);;

list_map_second (float) [("a",1);("b",2)] [];;

let tree_map f tr =
  let l  = List.rev (enum_paths tr) in
  let nl = list_map_second f l [] in
    insert_along_paths nl [];;

tree_map (fun x → (float x)) tr;;
```

Exercice 3: Méthode du gradient
Considérons la définition de type suivante sur les graphes, utilisant une représentation par liste d'adjacence :

```
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;;
```

```
Graph ([
        ((K 1, "A"), [K 2; K 3]);
        ((K 2, "B"), [K 4; K 5]);
        ((K 3, "C"), [K 6; K 7]);
        ((K 4, "A"), []);
        ((K 5, "C"), []);
        ((K 6, "D"), []);
        ((K 7, "C"), []);
      ])
```

1. 

2. Écrire une fonction qui, étant donné un graphe et une clé, renvoie la liste des clés des voisins.

3. Écrire une fonction énumérant les sommets d'un `graph` par un parcours en profondeur.

4. Écrire une fonction énumérant les sommets d'un `graph` selon un parcours en largeur.

Les deux fonctions précédentes ont en pratique beaucoup de code en commun. On se propose de factoriser les parcours sur les graphes en utilisant une *stratégie*, sous la forme d'une fonction décidant, 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)
```

5. 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.

6. 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.

7. 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.

✓ L'objectif consiste ici à obtenir une fonction de parcours qui dépend du sommet et du graphe, pour obtenir une méthode du gradient : à chaque étape, on se dirige vers son voisin ayant la valeur minimale. Cela justifie un peu plus la généralisation de la fonction de parcours.

```ocaml
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;;

let g = Graph [
  ((K 1,1),[K 2;K 3]);
  ((K 2,2),[K 4;K 5]);
  ((K 3,3),[K 6;K 7]);
  ((K 4,4),[K 8]);
  ((K 5,5),[]);
  ((K 6,6),[]);
  ((K 7,7),[]);
  ((K 8,8),[]);
];;

let graph_nexts (Graph adj) key =
  List.assoc key (List.map (fun ((k,_),l) -> (k,l)) adj);;

let list_of_graph_depth g start =
  let rec aux to_visit visited = match to_visit with
    | []     -> List.rev visited
    | k::ks -> if (List.mem k visited)
      then aux ks visited
      else aux ((graph_nexts g k)@ks) (k::visited)
  in
    aux [start] [];;

list_of_graph_depth g (K 1);;

let list_of_graph_breadth g start =
  let rec aux to_visit visited = match to_visit with
    | []     -> List.rev visited
    | k::ks -> if (List.mem k visited)
      then aux ks visited
      else aux (ks@(graph_nexts g k)) (k::visited)
  in
    aux [start] [];;

list_of_graph_breadth g (K 1);;

(* Strategies *)
type 'a strategy = Strategy of ('a -> 'a list -> 'a list -> 'a list)

let list_of_graph (Strategy s) g start =
  let rec aux to_visit visited = match to_visit with
    | []     -> List.rev visited
    | k::ks -> if (List.mem k visited)
      then aux ks visited
      else aux (s k (graph_nexts g k) ks) (k::visited)
  in
    aux [start] [];;

let strategy_depth = Strategy (fun _ next rest -> rest@next);;
let strategy_breadth = Strategy (fun _ next rest -> next@rest);;
```