TP Caml 1: Prolégomènes et Piles

Lionel RIEG
lionel.rieg@ens-lyon.fr

16 & 23 septembre 2008

1 Savez-vous encore tout faire fonctionner ?

Comment vous servir de votre éditeur préféré ? du toplevel Caml Light ?
Si vous ne savez plus, pas ou plus trop, quelques combinaisons efficaces :
- emacs comme éditeur de texte avec le mode Tuareg qui permet d'évaluer tout en regardant son code et fournit la coloration syntaxique
- jEdit avec les raccourcis à la emacs de M. Chilles (donc non reproductible chez soi) et le toplevel Caml Light
- n'importe quel éditeur (avec coloration syntaxique c'est souvent mieux) et le toplevel Caml Light
- si vous utilisez le toplevel Caml Light, n'y mettez que la ligne suivante et ré-exécutez-la à chaque fois que vous voulez évaluez :

```
#include "tpXX.ml"
```

où « tpXX.ml » est le nom du fichier dans lequel vous travaillez.

► **Question 1** Faites fonctionner la combinaison que vous préférez.

Autres remarques :
- Vous pouvez trouver la documentation de Caml Light à l'adresse `http://caml.inria.fr/pub/docs/manual-caml-light/` : les sections *The core library* et *The standard library* sont particulièrement intéressantes.
- Pensez à tester votre code au moins sur des cas simples, une erreur est toujours vite arrivée.
- Mais surtout n'oubliez pas, si vous avez une question, un doute, *etc*, **demandez-moi**, je suis là pour ça !

2 Parlez-vous encore Caml (Light) ?

2.1 Typage

► **Question 2** Qu'est-ce qu'un type ? À quoi sert le typage ?

► **Question 3** Écrire une fonction list_map de type

```
list_map : ('a → 'b) → 'a list → 'b list
```

qui applique son premier argument à tous les éléments de son second argument et renvoie la liste des résultats. Par exemple, list_map succ [1; 2; 3] donne [2; 3; 4].

► **Question 4** Écrire une fonction qui compose deux fonctions à un argument. Son type est :

```
compose : ('a → 'b) → ('c → 'a) → 'c → 'b
```

Peut-on la généraliser pour obtenir une fonction générale de composition ? Pourquoi ?

2.2 Trions !

Puisque les tris sont des algorithmes de base à savoir écrire (presque) les yeux fermés, commençons aujourd'hui par le tri par insertion. Si vous ne vous en souvenez plus, demandez-moi.

► **Question 5** Écrire le tri par insertion sur un tableau. Il aura le type

```
tri_insertion : 'a vect → 'a vect
```

► **Question 6** Modifier votre tri pour qu'on puisse passer en paramètre la fonction de comparaison, *i.e.* à la place de « < », on utilisera une fonction comp donnée en argument dont le type sera 'a → 'a → bool. Par convention, comp a b vaudra true si et seulement si $a < b$.

3 Piles

3.1 Piles par liste

On cherche à définir une pile. On rappelle qu'une pile permet essentiellement de rajouter un élément en première position et d'enlever le premier élément.

► **Question 7** On prend le type pile suivant :

```
exception Empty_stack;;
type 'a list_pile == 'a list ref ;;
```

Écrire une fonction de création de pile et une fonction de vidange de pile dont les types seront

```
list_create : unit → 'a list ref
list_clear : 'a list ref → unit
```

► **Question 8** Écrire les fonctions suivantes :
- push qui ajoute un élément en haut de la pile
- pop qui retire l'élément en haut de la pile
- peek qui renvoie l'élément en haut de la pile sans le dépiler.

Leurs types seront :

```
list_push : 'a → 'a list ref → unit
list_pop : 'a list ref → 'a
list_peek : 'a list ref → 'a
```

3.2 Piles par tableau

On souhaite créer une pile à taille limitée et on choisit pour cela le type suivant :

```
exception Full_stack ;;
type 'a vect_pile = {mutable index : int ; data : 'a vect };;
```

où index indique où se trouve le haut de la pile dans le tableau et data est un tableau de la taille maximale de la pile où on range ses éléments.

► **Question 9** Reprendre les deux questions précédentes pour cette implantation de pile. Les types des fonctions seront :

```
vect_create : 'a → int → 'a vect_pile
vect_clear : 'a vect_pile → unit
vect_push : 'a → 'a vect_pile → unit
vect_pop : 'a vect_pile → 'a
vect_peek : 'a vect_pile → 'a
```

Le gros désavantage de cette seconde méthode se voit dans le type de la fonction create : il n'existe pas de pile vide générique, on doit passer un élément à la fonction de création pour qu'elle sache de quel type est la pile. Pour y remédier, on modifie le type de la pile en

```
type 'a data_type = Not_init of int | Full of 'a vect ;;
type 'a vect_pile = {mutable index : int ; mutable data : 'a data_type };;
```

où Not_init n indique que la pile de taille n vient juste d'être créée et que son type précis n'est pas encore connu.

► **Question 10** Ré-écrire les fonctions d'accès avec ce nouveau type.

3.3 Interface générique

On souhaite éviter que l'utilisateur puisse tricher avec la pile en utilisant directement sa représentation interne, liste ou tableau. Pour cela, on veut cacher leur existence et fournir uniquement les fonctions d'accès, à savoir create, clear, push, pop et peek. On modifie donc le type de pile pour n'en faire qu'une interface (qui de plus est commune aux deux implémentations) : le nouveau type de pile devient alors

```
type 'a pile = {push : 'a → unit;
                pop : unit → 'a;
                peek : unit → 'a;
                clear : unit → unit};;
```

et la représentation interne de la pile est cachée dans sa création avec un **let in**.

► **Question 11** Ré-écrire la fonction de création de pile pour l'implantation par une liste.

► **Question 12** Reprendre la question précédente pour les vecteurs.

Dans la vraie vie, on cache la représentation d'une liste dans un type abstrait (appelé souvent t) dont la définition est masquée par l'interface. Le système de type l'accepte comme opaque et interdit sa déconstruction.

4 Calcul d'aire carrée

On cherche à trouver la plus grande piscine (ou le plus grand réservoir) qu'il est possible de placer dans un verger. Mais on ne veut pas déraciner d'arbre : la piscine doit être totalement visible. Un arbre est représenté par un carré. La carte du verger est donnée comme une matrice de taille $M \times N$ dans laquelle un « 0 » représente de l'herbe et un « 1 » représente un arbre.

► **Question 13** Commencer par écrire en Caml Light un algorithme qui retourne une matrice en comptant le nombre de cases libres au dessus d'une case donnée. Par exemple, si dans la case d'indice (i, j) de la matrice on a 7, cela signifie que $m(i, j-7)$ est occupée par un arbre mais pas $m(i, j-k)$ pour $0 \leqslant k \leqslant 6$.

Ceci permet de savoir en temps constant si un rectangle d'une largeur donnée peut se prolonger. À l'aide de cette matrice, on peut écrire un algorithme qui résout le problème à l'aide de la programmation dynamique[1] mais il est inefficace (sa complexité est de l'ordre de M^2N^2).

Nous allons nous intéresser ici à l'algorithme optimal en $\Theta(MN)$.

► **Question 14** Pourquoi cette complexité est-elle optimale ?

Si vous arrivez là, demandez-moi de vous expliquer l'algorithme en direct.

► **Question 15** Programmer cet algorithme.

[1] Si vous avez oublié ce que c'est, ne vous en faites pas trop, on reverra ça dans un prochain TP.

5 Solutions

▸ **Question 1** À chacun la sienne. Je pourrai d'avantage vous aider avec emacs. D'ailleurs, je devrais déjà vous avoir fait un petit aide-memoire des commandes les plus utiles.

▸ **Question 2** Le type indique le "genre" de l'objet (un entier, une fonction de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$, *etc*), il donne une information sur son contenu. Il permet de s'assurer que les opérations sont bien effectuées sur des arguments pour lesquels cela fait sens et que les fonctions appelées savent gérer.

▸ **Question 3**

```
let rec list_map f = function
  | [] → []
  | t :: q → f t :: list_map f q;;
```

▸ **Question 4**

```
let compose f g x = f (g x);;
(* OU *)
let compose f g = fun x → f (g x);;
```

Une fonction généralisée de composition prendrait une fonction f à un nombre quelconque d'arguments et une fonction g a un seul argument (ou bien il faut préciser ce qu'on entend par composition) et crée $f \circ g$. Elle ne peut pas exister en Caml car on ne peut pas la typer : en effet le type d'une fonction contient son nombre d'argument.

▸ **Question 5** Voir la solution de la question suivante en remplaçant les comp a b par des a < b (et en supprimant comp des arguments bien sûr).

▸ **Question 6** On peut choisir de faire une recherche linéaire ou dichotomique. Si on utilise une recherche linéaire, une variante consiste à introduire les décalages dans la recherche en faisant des échanges à la façon du tri bulle.

```
let echange tab i j =
  let stored = tab.(j) in
    for k = j downto i+1 do
      tab.(k) <- tab.(k-1)
    done;
    tab.(i) <- stored ;;

let lin_trouve comp tab elt =
  let i = ref 0 in
    while comp tab.(!i) elt do incr i done;
    !i ;;

let dich_trouve comp tab size elt =
  let rec trouve_aux k size =
    if size = 1
    then if comp elt tab.(k) then k else k+1
    else begin
      if comp tab.(k + (size + 1) / 2) elt
      then trouve_aux (k + (size + 1) / 2) ((size + 1) / 2)
      else trouve_aux k ((size + 1) / 2)
    end
  in
    trouve_aux 0 size ;;

let tri_insertion comp tab =
  for i = 1 to (vect_length tab) - 1 do
    echange tab (dich_trouve comp tab i tab.(i)) i
  done;
  tab ;;
```

Et vu que le code est un peu compliqué, on n'oublie pas de le tester :

```
tri_insertion (prefix <) [|3; 101; 31; 34; 34; -75; -9; 3; -8; 9; 10; 9; 4|];;
```

▸ **Question 7**

```
let list_create () = ref [];;

let list_clear p = p := [];;
```

L'utilisation de référence pour la liste permet de modifier la valeur de la pile. En effet, les listes sont des objets du monde fonctionnel qui ne peuvent être modifiés au contraire des tableaux ou des références qui appartiennent au monde impératif.

▸ **Question 8**

```
let list_push e p = p := e :: !p;;

let list_pop p = match !p with
  | [] -> raise Empty_stack
  | t :: q -> p := q; t;;

let list_peek p = match !p with
  | [] -> raise Empty_stack
  | t :: q -> t;;
```

▸ **Question 9**

```
let vect_create elem n = {index = 0; data = make_vect n elem};;

let vect_clear p = p.index <- 0;;

let vect_push e p =
  if p.index = vect_length p.data
  then raise Full_stack; (* mettre 'else' est non nécessaire *)
  p.data.(p.index) <- e; (* puisque 'raise' stoppe le calcul *)
  p.index <- p.index +1;;

let vect_pop p =
  if p.index = 0
  then raise Empty_stack
  else
    begin
      p.index <- p.index -1;
      p.data.(p.index)
    end;;

let vect_peek p =
  if p.index = 0
  then raise Empty_stack
  else p.data.(p.index -1);;
```

▸ **Question 10**

```
let vect_create n = {index = 0; data = Not_init n};;

let vect_clear p = p.index <- 0;;

let vect_push e p =
```

```
  match p.data with
    | Not_init n →
        begin
          p.data <− Full (make_vect n e);
          p.index <− p.index +1
        end
    | Full v →
        if p.index = vect_length v
        then raise Full_stack;
        v.(p.index) <− e;
        p.data <− Full v;;

let pop_peek p b = (* b est un booléen qui indique *)
  if p.index = 0    (* si on enlève l'élément ou non *)
  then raise Empty_stack
  else
    begin
      match p.data with
        | Not_init _ → failwith "Never reached"
        | Full v →
            if b then p.index <− p.index −1;
            v.(p.index)
    end;;

let vect_pop p = pop_peek p true ;;

let vect_peek p = pop_peek p false ;;
```

► **Question 11** On veut cacher la définition de la pile et on va donc utiliser pour cela une construction en **let in** qui nous permet de définir une valeur locale, utilisée ensuite pour définir les fonctions du type demandé.

```
let create_list_stack () =
  let stack = ref [] in
    {push = (fun e → list_push e stack);
     pop = (fun () → list_pop stack);
     peek = (fun () → list_peek stack);
     clear = (fun () → list_clear stack)};;
```

► **Question 12**

```
let create_vect_stack n =
  let stack = vect_create n in
    {push = (fun e → vect_push e stack);
     pop = (fun () → vect_pop stack);
     peek = (fun () → vect_peek stack);
     clear = (fun () → vect_clear stack)};;
```

► **Question 13**

```
let sum_mat m =
  let sum = make_matrix (vect_length m) (vect_length m.(0)) 0 in
    for j = 0 to (vect_length m.(0)) − 1 do
      sum.(0).(j) <− if m.(0).(j) = 0 then 1 else 0
    done;
    for i = 1 to (vect_length m) − 1 do
      for j = 0 to (vect_length m.(0)) − 1 do
        sum.(i).(j) <− if m.(i).(j) = 0 then 1 + sum.(i−1).(j) else 0
      done;
```

```
    done;
    sum;;
```

▶ **Question 14** Il nous faut lire la carte qui fait $M \times N$ cases.

▶ **Question 15** L'idée de l'algorithme est de savoir grâce à la pile jusqu'où peut être prolonger un rectangle de hauteur donnée dont le côté bas se trouve sur la ligne courante. L'algorithme est ci-dessous, si vous voulez d'avantage de précisions, demandez-moi de vous l'expliquer en direct, le faire sur papier est facilement incompréhensible.

```
let pop_and_compute stack sum_j i j maxi =
  let height = ref (fst (stack.peek ())) in
  let pos = ref 0 in
    while !height > sum_j do
      pos := snd (stack.pop ());
      if fst !maxi < (!height * (j - !pos))
      then maxi := ((!height * (j - !pos)), (i + 1, j - 1));
      height := fst (stack.peek ())
    done;
    stack.push (sum_j, !pos);;

let compute_line stack i sum_line maxi =
  stack.clear ();
  stack.push (0,0);
  for j = 1 to vect_length sum_line do
    match sum_line.(j - 1) - fst (stack.peek ()) with
      | 0 → ()
      | n when n > 0 → stack.push (sum_line.(j - 1), j)
      | _ → pop_and_compute stack sum_line.(j - 1) i j maxi

  done;;

let compute_map mat =
  let sum = sum_mat mat in
  let maxi = ref (0, (0, 0)) in
  let stack = create_list_stack () in
    for i = 0 to (vect_length mat) - 1 do
      compute_line stack i sum.(i) maxi
    done;
    !maxi;;
```

Si vous voulez tester ce code, voici de quoi :

```
#open "random";;

let random_mat densite n m =
  let mat = make_matrix n m 0 in
    for i = 0 to n - 1 do
      for j = 0 to m - 1 do
        mat.(i).(j) <- if random__int densite = 0 then 1 else 0
      done;
    done;
    mat;;

let test densite n m =
  let mat = random_mat densite n m in
    compute_map (sum_mat mat), mat;;

test 4 13 20;;
```