Corrigé du TP 3 de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

3 février 2014

1 Listes

```
let rec appartient x l = match l with
  [] -> false
| y :: reste ->
    if x = y
    then true
    else appartient x reste

let rec existe p l = match l with
  [] -> false
| y :: reste ->
    if p y
    then true
    else existe p reste
```

La fonction `existe` est une généralisation de `appartient` ; il suffit de remplacer la comparaison de `x` à `y` par le prédicat `p`. On en déduit comment réécrire `appartient` en utilisant `existe` :

```
let appartient2 x l = existe (fun y -> x = y) l;;

let rec associé x l = match l with
  [] -> failwith "Erreur: pas d'associé"
| (y1, y2) :: reste ->
    if x = y1 then y2
    else associé x reste

let rec map f l = match l with
  [] -> []
| x :: l' -> (f x) :: (map f l')

let rec split l = match l with
  [] -> ([], [])
| (x1, x2) :: l' ->
    let (l1, l2) = split l'
    in (x1 :: l1, x2 :: l2)
```

```
let rec partage p l = match l with
  [] -> ([], [])
| x :: l' ->
    let (oui, non) = partage p l'
    in if (p x) then (x :: oui, non)
       else (oui, x :: non)
```

Ci-dessus, l'appel récursif à `partage` fait le partage de la queue de liste `l`. La variable locale `oui` (resp. `non`) contient les éléments de `l` qui vérifient (resp. ne vérifient pas) `p`. Il ne reste plus qu'à accrocher `x` à la bonne liste.

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | x :: l -> x :: (append l l2)

let rec fold_left f accu l = match l with
  [] -> accu
| a::l' -> fold_left f (f accu a) l'

let rec fold_right f l accu =
  match l with
    [] -> accu
  | a::l -> f a (fold_right f l accu)

let append l1 l2 = fold_right (fun x l -> x :: l) l1 l2
```

Si `l1` vaut `[ a1; ...; an ]`, alors d'après la définition de `fold_right`, nous avons

```
append l1 l2 = a1 :: ... :: an :: l2
```

ce qui est bien le résultat recherché.

2 Tri fusion (*Merge sort*)

On donne directement la version paramétrée par une fonction d'ordre.

```
let singletons l = map (fun x -> [x]) l

let rec merge ordre l1 l2 =
  match (l1, l2) with
    ([], _) -> l2
  | (_, []) -> l1
  | (x1 :: rest1, x2 :: rest2) ->
      if ordre x1 x2
      then x1 :: (merge ordre rest1 l2)
      else x2 :: (merge ordre l1 rest2)
```

On réalise un filtrage sur la paire `(l1, l2)` (au lieu de faire un filtrage sur `l1` suivi d'un autre sur `l2`) pour plus de concision. Notons que les deux premières branches du `match` se recouvrent dans le cas `([], [])`, ce qui n'est

nullement gênant ; la première s'appliquera alors. Remarquons que dans le cas où `x1` est inférieur à `x2`, il serait incorrect de renvoyer `x1 :: x2 :: (merge ordre rest1 rest2)`. En effet, `rest1` peut très bien contenir d'autres éléments compris entre `x1` et `x2`.

```
let rec merge2à2 ordre l = match l with
  l1 :: l2 :: rest ->
   (merge ordre l1 l2) :: (merge2à2 ordre rest)
| ll -> ll
```

La première ligne du filtrage s'applique lorsque la liste passée à `merge2à2` contient au moins deux éléments. La seconde ligne s'applique dans tous les autres cas (parce que `ll` est un nom de variable), c'est-à-dire lorsque la liste contient au plus 1 élément.

```
let auplus1 l = match l with
  (_ :: _ :: _) -> false
| _ -> true
```

Cette fonction indique si la liste qu'on lui passe contient au plus 1 élément. Le principe est le même que pour `merge2à2`, mais comme on n'a pas besoin d'utiliser les éléments, on utilise `_` au lieu de variables.

```
let rec répète fusion prédicat l =
  if prédicat (l)
  then l
  else répète fusion prédicat (fusion l)

let mergesort ordre l =
  let résultat = répète (merge2à2 ordre) auplus1 (singletons l)
in match résultat with
     [] -> []
   | l :: _ -> l
```

Ici, l'application partielle `(merge2à2 ordre)` donne une fonction de type $\forall\alpha.\alpha$ `list list` $\rightarrow \alpha$ `list list`, qui constitue une étape de fusion. On utilise ensuite la fonction `répète` pour effectuer cette étape autant de fois que nécessaire. Plus précisément, `répète` attend trois arguments :

- l'action à répéter, ici `(merge2à2 ordre)` ;
- la condition d'arrêt, ici le prédicat `auplus1`, ce qui signifie que l'on s'arrêtera lorsque la liste de listes ne contiendra plus qu'un argument ;
- le point de départ, à savoir `(singletons l)`, la liste des listes à un élément.

Lorsque `répète` s'arrête, le résultat qu'elle renvoie vérifie nécessairement le prédicat `auplus1` ; il s'agit d'une liste de 0 ou 1 listes triées. On utilise un dernier `match` pour traiter ces deux cas. Dans la deuxième ligne, on sait que la liste a exactement 1 élément ; le motif `_` filtrera en fait toujours une liste vide.

Nous pouvons maintenant compter le nombre d'opérations effectuées par chacune des fonctions ci-dessus. Lorsque la fonction `merge` se rappelle récursivement, elle a effectué une comparaison, et les listes passées à l'appel récursif contiennent (au total) un élément de moins. Par ailleurs, `merge` s'arrête lorsque

l'une des deux listes est vide. Par conséquent, `merge ordre l1 l2` réalise au plus $n_1 + n_2$ comparaisons, où n_i est la taille de `li`.

Ceci étant établi, on vérifie que `merge2à2 ordre [ l1; l2; ... ]` effectue au plus $n_1 + n_2 + \ldots$ comparaisons. C'est-à-dire au plus n comparaisons, si n est la taille de la liste que nous sommes chargés de trier.

Pour savoir combien `mergesort ordre l` effectue de comparaisons, il faut déterminer combien de fois `merge2à2` est appelée. Or, à chaque appel, elle divise par deux [1] le nombre d'éléments de la liste de listes. Si celle-ci est au départ de taille n, le nombre d'étapes effectuées est donc au plus $\log_2 n$.

Des deux paragraphes précédents, on déduit que le temps nécessaire à `mergesort` pour trier une liste de taille n est au pire de l'ordre de $n \cdot \log_2 n$, ce que l'on écrit $O(n \cdot \log_2 n)$.

3 Tri rapide (*Quicksort*)

```
let rec quicksort = function
  [] -> []
| pivot :: reste ->
    let (petits, grands) = partage (fun x -> x < pivot) reste in
    (quicksort petits) @ (pivot :: (quicksort grands))
```

L'opérateur `@` est la concaténation de listes prédéfinie en Objective Caml (comme la fonction `append` que nous avons écrite).

1. En fait, si `ll` est de taille n, alors `merge2à2 ordre ll` est de taille $n/2 + 1$ au plus. Ceci ne change pas le principe de la preuve.