# Programmation - TP2: Aspects Mémoire

Simon Halfon

September 15, 2015

## 1 Ce qu'il faut retenir du TP1

Je l'ai déjà dit au TP1, mais on ne vous le répétera jamais assez, donc je vous l'écris ici: Vous allez être notés sur un projet de programmation, c'est à dire qu'un **humain** devra lire votre code. Il faut donc produire un code propre et **lisible**. C'est une bonne habitude à prendre, même pour vous. Quelques conseils:

- Donnez à vos variables et fonctions des noms **explicites** (vos enseignants sont des chercheurs, mais ils ont autre chose à chercher que ce que sont censées faire les fonctions $f_1$, $f_2$ et $f_3$)
- Commentez votre code ! Une ligne pour dire ce que fait la fonction $f$ (qui ne s'appelle pas $f$, cf point précédent)
- Si vous utilisez une fonction auxiliaire à l'intérieur d'une autre fonction, elle NE s'appelle PAS `aux`
- L'indentation, les sauts de lignes, séparer le code en différentes section ou même fichiers améliore grandement la lisibilité
- Rendez votre code modulaire

## 2 Objectifs du TP

Comme on l'a vu au premier TP, Ocaml supporte deux paradigmes de programmation: la programmation *impérative* et la programmation *fonctionnelle*. Il ne faut pas mélanger les deux dans un même code, et il faut choisir le plus adapté à votre programme. Ce choix dépend de la *structure de donnée* que vous utilisez. Ce choix de structure de donné peut être crucial: code plus simple, plus efficace, ... Pensez par exemple au choix Listes ou Tableaux, et au paradigme de programmation qui est le plus naturel pour chaque type. Dans ce TP, on va voir deux sortes de structures de données, obtenues par deux façon de construire des types en Ocaml:

A) Les Types Produits (ou Record)

B) Les Types Sommes

Le premier cas suggère une programmation impérative, le second une programmation récursive.

## 3 Exercices

---

**Exercice - 1** *Mini Librairie Listes circulaires doublement chainées*

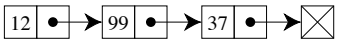
---

Plutôt que d'inclure quelques rappels très incomplets sur les records, je vous invite à consulter l'un des très bon tutoriels que l'on peut trouver sur Internet (par exemple sur https://realworldocaml.org). En particulier, renseignez vous sur les champs mutables. L'exemple le plus simple de type produit, auquel vous êtes habitué, est le type référence. Essayez cette ligne en Ocaml:

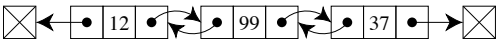
```
let x = ref 0 in x ;;
```

Le point d'exclamation "!x" qui permet d'accéder à la valeur contenu dans "x" est en fait un raccourci syntaxique pour "x.content".

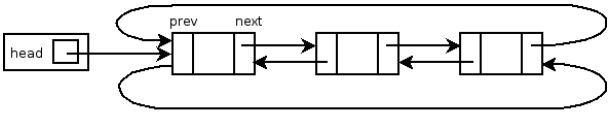
En mémoire, une liste est stockée comme plusieurs *cellules* "éparpillée" dans la mémoire. Chaque cellule contient une valeur d'élément, et l'adresse mémoire de la cellule suivante.




Lorsque Ocaml manipule une liste, il ne connait en réalité que l'adresse de la première cellule. C'est pour cela qu'il est coûteux d'accéder au dernier élément: il faut "sauter" de cellule en cellule, chaque cellule dévoilant l'adresse de la suivante. De plus, connaissant l'adresse mémoire d'une cellule, il est impossible de retrouver l'adresse de la cellule précédente. Il est donc parfois d'utiliser une structure de donnée plus complexe: les listes doublement chainées.




On va en fait utiliser une troisième structure: les listes doublement chainées circulaires.




Ecriture de la librairie:

**1-** On commence par définir les types polymorphes `'a cell` pour les cellules et `'a circ_list` pour les listes. Pour pouvoir traiter le cas de la liste vide, il va falloir utiliser le type `option`.

**2-** Définir `nil` la liste vide, puis la liste singleton contenant l'entier 1.

**3-** Définir les fonctions:

- `hd: 'a circ_list -> 'a`

  qui retourne le premier élément d'une liste.

- `last: 'a circ_list -> 'a`

  qui retourne le dernier élément d'une liste.

- `nth: int -> 'a circ_list -> 'a`

  qui retourne le nieme élément d'une liste.

- `cell: 'a circ_list -> 'a cell`

  qui retourne la première cellule d'une liste.

- `length: 'a circ_list -> int`

  Vous aurez sans doute besoin de la primitive == de Ocaml.

- `list_of_circ_list: 'a circ_list -> 'a list`

  qui retourne la liste (du type list de Ocaml) correspondant à une liste circulaire. Cela vous permettra d'afficher vos listes de manière lisible, mais il est interdit de l'utiliser pour les fonctions suivantes (seulement pour afficher vos exemples).

- `egal: 'a circ_list -> 'a circ_list -> bool`

  qui teste si deux listes circulaires représentent la même liste.

- `circ_perm: 'a circ_list -> 'a circ_list -> bool`

  qui teste si une liste circulaire est une permutation circulaire de l'autre.

- `circ_perm__memory: 'a circ_list -> 'a circ_list -> bool`

  qui teste si une liste est physiquement une permutation circulaire de l'autre. C'est par exemple le cas pour les listes $\ell_1$ et $\ell_2$ si $\ell_1$ pointe vers une cellule `cell` et $\ell_2$ vers `cell.next`.

- `copy: 'a circ_list -> 'a circ_list`

  qui copie une liste circulaire. Attention, la liste retournée doit être indépendante de la liste donnée en argument: modifier la première liste ne doit pas changer la seconde ! (penseze aux vecteurs).

- `insert: 'a -> 'a circ_list -> 'a circ_list`

  qui insère un élément en tête de liste. Cette fois, pas de copie ! La liste retournée doit être la même. Par exemple, votre fonction doit vérifier ceci (si old_list n'est pas vide):

  ```
  let new_list = insert 3 old_list in
  let c = cell new_list in
  c.next == cell old_list
  ```

  doit renvoyer "true". Est-ce aussi le cas avec:

  ```
  let new_list = insert 3 old_list in
  let c = cell old_list in
  c.prev == cell new_list
  ```

  (ici "next" et "prev" sont deux champs du type "'a cell").

- La réciproque de la fonction `list_of_circ_list`

  ```
  circ_list_of_list: 'a list -> 'a circ_list
  ```

  qui pourrait s'appeller `create`. Elle vous permettra de créer vos exemples sans avoir à en écrire des tartines à la main.

- `append: 'a circ_list -> 'a circ_list -> 'a circ_list`

  qui effectue la concaténation de deux listes. Ici aussi, la fonction doit être destructive, c'est à dire qu'elle altère ses arguments.

- `filter: ('a -> bool) -> 'a circ_list -> 'a circ_list`

  qui sélectionne les élément d'une liste qui vérifie le prédicat donné en argument. La liste en argument NE doit PAS être modifiée.

- En déduire la fonction

  ```
  suppr_elt: 'a -> 'a circ_list -> 'a circ_list
  ```

  qui supprime les occurences d'un élément.

- `map: ('a -> 'b) -> 'a circ_list -> 'b circ_list`

---

**Exercice - 2** *Une représentation optimisée des arbres*

---

Comme à l'exercice précédent, je vais vous demander d'aller faire votre culure générale tout seul. Vous l'aurez compris, il s'agit ici de se familiariser avec les types somme. Un exemple de type somme que vous connaissez très bien, ce sont les listes de Ocaml:

```
type 'a my_list = Nil | Cons of 'a * 'a my_list
```

C'est réellement comme ça que sont codées les listes en Ocaml, et la représentation `[1;2;3]` au lieu de `Cons(1, Cons(2, Cons(3, nil)))` est un cadeau que vous fait Ocaml.
**1-** Recoder les fonctions `::`, `hd` et `tl` que vous connaissez bien.

Le type option utilisé dans l'exercice précédent est également un exemple de type somme simplissime:

```
type 'a option = None | Some of 'a
```

La façon élégante de déconstruire un type somme, c'est avec la syntaxe `match _ with`, avec laquelle vous DEVEZ vous familiarisez si ce n'est pas déjà le cas.

On va s'intéresser dans cet exercice à un objet que l'on rencontre dans plusieurs domaines de l'informatique et qui a une structure naturellement arborescentes: les termes. Vous les rencontrerez en logique.

Une *signature* est un ensemble fini $\Sigma$ de *symboles* muni d'une fonction arité $a : \Sigma \to \mathbb{N}$. On se donne un ensemble dénombrable de variables $\mathcal{X}$. L'ensemble des termes $\mathcal{T}(\Sigma, \mathcal{X})$ sur la signature $\Sigma$ et l'ensemble de variable $\mathcal{X}$ est défini inductivement comme suit:

- $x$ est un terme pour $x \in \mathcal{X}$
- Si $f$ est un symbole de $\Sigma$ d'arité $n$ et si $t_1, \ldots, t_n$ sont des termes, alors $f(t_1, \ldots, t_n)$ est un terme.

Par exemple, soit $\Sigma = \{0(0), s(1), +(2), *(2)\}$ une signature (on note entre parenthèse l'arité du symbole), et soit $x, y, z \in \mathcal{X}$ des variables, voici des exemples de termes:

- $0$
- $s(s(s(0)))$
- $+(s(s(x)), y)$
- $+(s(x), *(y, z))$

**2-** Proposez un type pour les variables, pour les symboles et pour les termes.
**3-** Ecrire une fonction

```
well_formed: (symbol -> int) -> term -> bool
```

qui vérifie qu'un terme est bien formé vis à vis d'une signature. On ne se souciera plus de la signature par la suite.

Une substitution est une fonction $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. On définit le domaine d'une substitution $Dom(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. Toute substitution $\sigma$ s'étend en un morphisme $\hat{\sigma}$ de $\mathcal{T}(\Sigma, \mathcal{X})$ dans lui même définit par:

- $\hat{\sigma}(x) = \sigma(x)$
- $\hat{\sigma}(f(t_1, \ldots, t_n)) = f(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n))$

Dans la suite, on note $\sigma$ au lieu de $\hat{\sigma}$.
**4-** Ecrivez le type des substitutions
**5-** Ecrire les fonctions

- `in_domain: var -> substitution -> bool`
  qui teste si une variable est dans le domaine d'une substitution.
- `occur: var -> term -> bool`
  qui teste si une variable apparait dans un terme.
- `subst: substitution -> term -> term`
  qui implémente la fonction $\hat{\sigma}$.
- `equal: term -> term -> bool`
  qui teste l'égalité de deux termes.

**6-** Supposons que l'on dispose d'un symbole $f$ d'arité 2.
**a-** Ecrivez une fonction `big_tree` qui prenne en entrée un entier $n$ et retourne le terme correspondant à l'arbre binaire complet de profondeur $n$ dont tous les noeuds sont étiquetés par $f$ et les feuilles par une variable $x$.
**b-** Appliquer la substitution $\sigma$ qui à $x$ associe $f(x, x)$ à `big_tree 1000`. D'où vient le problème ?

Pour contourner le problème, nous allons choisir une structure de donnée qui permet de partager les sous arbres, et de ne pas appliquer la fonction de substitution aux deux sous arbres s'ils sont identiques.
**7-** Proposez une telle structure de donnée (la structure courrament utilisée s'appelle *DAG*, comme *Directed Acyclic Graph*).
**8-** Réécrivez les fonctions précédentes pour cette structure de donnée.