Conseils et erreurs courantes

Samuele Giraudo

Avril 2022

Les points qui suivent ici sont en vrac, sans véritablement de hiérarchie.

1 Listes

1.1 Parcours de listes et fonctions d'ordre supérieur

Tout parcours manuel de liste effectué par une (sous-)fonction récursive comme

```
let rec aux lst =
    match lst with
        |[] -> ...
        |x :: lst' -> ... (aux lst') ...
```

peut quasi toujours être remplacé par une expression faisant intervenir un ou plusieurs appels à des fonctions d'ordre supérieur du module `List` comme entre autres `List.map`, `List.filter`, `List.fold_left`, `List.exists` ou `List.for_all`. Il faut donc éviter au maximum la pratique illustrée ci-dessus et rendre son programme plus élégant et concis à l'aide de ces fonctions.

1.2 Concaténation de listes

Si nous disposons de `lst` qui est une liste de listes de `'a` et que nous souhaitons en faire une simple liste de `'a` en concaténant chacune de ses listes (par exemple, `[[1]; [2; 3]; []; [4; 5; 6]]` doit devenir `[1; 2; 3; 4; 5; 6]`), il est possible d'utiliser

```
lst |> List.fold_left (@) []
```

Il y a cependant mieux :

```
List.concat lst
```

ou encore

```
List.flatten lst
```

Les expressions `List.fold_left (@) []`, `List.concat` et `List.flatten` sont toutes trois équivalentes et, par le principe de **transparence référentielle**, l'une peut être substituée à l'autre dans tout programme sans changer la valeur calculée (mais dans ce cas, autant prendre la formulation syntaxique la plus simple et courte).

2 Fonctions et appels

2.1 Utilisation de l'application partielle de fonctions

Au lieu d'écrire, étant donnée par exemple une fonction `fact : int -> int`

```
List.map (fun x -> fact x) [0; 1; 2; 3]
```

il est préférable d'écrire

```
List.map fact [0; 1; 2; 3]
```

Les expressions `(fun x -> fact x)` et `fact` sont en effet absolument équivalentes.

Une exploitation de ça consiste à écrire directement

```
List.map ((+) 8) [0; 1; 2; 3]
(* Au lieu de List.map (fun x -> 8 + x) [0; 1; 2; 3] *)
```

pour incrémenter de `8` les éléments de la liste en argument.

Questton : comment faire, de manière similaire, pour décrémenter de `32` les éléments de la liste en argument ? Attention, l'opérateur `(-)` n'est pas commutatif.

2.2 Opérateur d'application inversée

Il y a un opérateur très utile en Caml qui permet d'appliquer des fonctions de manière itérée sur une expression dans un ordre naturel. Supposons par exemple que nous ayons des fonctions `fact`, `fibo` et `cat`, toutes paramétrées par un entier `n`, et qui renvoient respectivement la factorielle de `n`, le `n`e nombre de Fibonacci et le `n`e nombre de Catalan. Notre objectif est de calculer la factorielle de Fibonacci de Catalan de `8`. Nous pouvons donc écrire

```
fact (fibo (cat 8))
```

Mais, il y a mieux : il est possible d'enchaîner de manière plus naturelle les transformations. En effet, nous partons de `8`, nous calculons son image par Catalan, puis nous calculons l'image de ce que nous obtenons par Fibonacci, puis la factorielle de cette valeur. Cela donne ceci :

```
8 |> cat |> fibo |> fact
```

Exemple bien plus édifiant avec des listes : nous avons une liste `lst` de chaînes de caractères. Notre objectif est de calculer la concaténation de ses éléments, en ajoutant au préalable le caractère `'*'` à la fin de chacune des chaînes et en se basant sur l'image miroir de la liste (par exemple, si `lst = ["a"; "bb"]`, on doit obtenir comme résultat `"bb*a*"`). De base, Nous pouvons répondre au problème par

```
List.fold_left (^) "" (List.map (fun u -> u ^ "*") (List.rev lst))
```

mais comme nous pouvons le constater, ce n'est pas naturel ni très lisible. Avec l'opérateur `|>` ça donne

```
List.rev lst |> List.map (fun u -> u ^ "*") |> List.fold_left (^) ""
```

qui est beaucoup plus simple à écrire et à relire.

Bonus : l'opérateur `|>`, vu en cours, est de type `'a -> ('a -> 'b) -> 'b` et ce dernier exemple exploite le fait que la liste est le dernier argument de toutes ces fonctions de manipulation sur

ces listes. Ça explique pourquoi ces fonctions d'ordre supérieur sur les listes sont construites de sorte à avoir la liste qui figure en dernière position.

2.3 Fonctions récursives et appels

Quand une fonction récursive a besoin de faire exactement le même appel plusieurs fois, il est important de lier un nom à cette valeur pour l'utiliser plusieurs fois plutôt que faire plusieurs appels. Par exemple, pour l'exponentiation dichotomique, nous souhaitons calculer `x^(n / 2)` pour l'utiliser deux fois pour calculer `x^(n / 2) * x^(n / 2)` qui est donc égal à `x^n` (quand `n` est pair). Nous n'écrivons donc pas

```
(puissance x (n / 2)) * (puissance x (n / 2))
```

mais plutôt

```
let tmp = puissance x (n / 2) in
tmp * tmp
```

Cette remarque s'applique, par exemple, à une fonction `developper` traitant un arbre unaire binaire pour laquelle, quand nous visitons un nœud unaire, nous souhaitons créer un nœud binaire où ses deux fils sont calculés par appel récursif. Nous n'allons donc pas écrire

```
|Unaire a -> Binaire (developper a, developper a)
```

mais plutôt

```
|Unaire a ->
    let tmp = developper a in
    Binaire (tmp, tmp)
```

dans le filtrage. Ceci réduit la complexité de beaucoup. Ces exemples sont très simples, mais il faut garder ces schémas de base en tête pour la pratique courante qui peut être plus complexe.

3 Filtrage

3.1 Accès à la tête et à la queue par déconstruction

Lors d'un parcours de liste (à ne faire manuellement que si la tâche demandée ne peut pas être faite raisonnablement par l'usage de fonctions d'ordre supérieur), l'accès à la tête de la liste et la queue de la liste doit se faire par déconstruction via un filtrage. Il est en effet maladroit d'utiliser les fonctions `List.hd` et `List.tl`. Par exemple, dans

```
match lst with
    ...
    |x :: lst' -> (* (A) *)
```

à l'endroit `(A)`, par déconstruction, nous avons bien accès via `x` et `lst'` à la tête et la queue de la liste. C'est la manière correcte de faire. Ce qui serait maladroit serait de faire

```
let x = List.hd lst and lst' = List.tl lst in
...
```

En plus de cela, ce code n'est pas sûr car `lst` pourrait être la liste vide (dans ce cas les évaluations des expressions `List.hd lst` et `List.tl lst` lèveraient des exceptions) !

3.2 Renvoi efficace de valeurs d'un type somme

Quand, lors d'un filtrage, figure une clause qui renvoie directement la valeur filtrée, il est moins efficace de la reconstruire plutôt que de renvoyer directement la valeur filtrée. Concrètement, au lieu d'écrire

```
match x with
    |Feuille v -> Feuille v
    ...
```

il est préférable de procéder par

```
match x with
    |Feuille v -> x
    ...
```

Cela évite de reconstruire `Feuille v`.

4 Divers

4.1 Opérateur de comparaison

L'opérateur `=` du Caml est vraiment très évolué : il teste l'égalité **sémantique** entre deux valeurs de type quelconque mais du même type. Et ceci, de manière récursive. Donc si nous avons par exemple deux arbres unaires binaires `a1` et `a2`, le test d'égalité `a1 = a2` va véritablement comparer les deux arbres récursivement et de manière automatique. Il n'est donc pas nécessaire de réécrire soi-même le test d'égalité (comme demandé par certains autres langages).

Au passage, ne jamais utiliser en programmation fonctionnelle l'opérateur `==` qui lui ne teste pas l'égalité sémantique mais plutôt l'égalité **physique** (même position en mémoire). Par exemple,

```
# Binaire (Feuille 8, Feuille 16) = Binaire (Feuille 8, Feuille 16);;
- : bool = true
# Binaire (Feuille 8, Feuille 16) == Binaire (Feuille 8, Feuille 16);;
- : bool = false
```

Pour ce qui est des opérations de test d'inégalité, elles marchent selon les mêmes principes et `<>` est l'opérateur d'inégalité sémantique et `!=` est celui d'inégalité physique (qui est donc à bannir ici dans notre cadre de programmation fonctionnelle).

4.2 Style d'écriture

Quelle est la différence entre un programmeur fonctionnel et un programmeur impératif qui s'essaie à la programmation fonctionnelle ? Le premier écrit

```
fct_1 (fct_2 arg) (* Tres beau. *)
```

où `fct_1` et `fct_2` sont deux fonctions et `arg` un argument quelconque, tandis que le second écrit

```
fct_1(fct_2 arg) (* Beaucoup moins beau. *)
```

Tout est dans l'**espace supplémentaire** du premier exemple ! Les parenthèses ne servent en effet pas à réaliser l'appel de fonction (comme dans le cas du C ou du Python par exemple) mais servent simplement à écrire les expressions sans ambiguïté. En effet, sans les parenthèses, l'expression

`fct_1 fct_2 arg` (équivalente à `(fct_1 fct_2) arg`) serait comprise comme l'application de `fct_1` sur les deux arguments `fct_2` et `arg` (de manière équivalente par curryfication, elle est comprise comme l'application de `(fct_1 fct_2)` sur `arg`).

5 Conseils pratiques

5.1 Noms des fichiers à rendre

Quand des champs `Nom` et `Prenom` figurent à remplir dans le fichier source à rendre, il faut le faire.

5.2 Bonne compilation

Attention à faire bien en sorte que le rendu s'interprète sans erreur. En cas de doute pour l'écriture d'une fonction, il est possible de la laisser en commentaires pour que le programme compile. Elle sera tout même regardée.