TD 1 : Programmation fonctionnelle

Michael Monerau

2 octobre 2010

Introduction

L'objectif de nos TDs cette année va être de bien maîtriser Caml pour implémenter des algorithmes assez simples sur les structures de données vues en cours ou que vous pourrez rencontrer en concours.

Quelques trucs en vrac :

- Retrouvez les TDs et corrigés sur http ://www.eleves.ens.fr/ monerau
- L'aide de Caml se trouve en ligne sur http ://-caml.inria.fr
- Vous pouvez m'envoyer vos fichiers par mail si vous voulez que je relise votre code (adresse sur ma page)
- Ecrivez toujours des commentaires sur ce que vous faites : hypothèses sur les arguments, valeur de retour
- Quand vous écrivez une fonction, testez la systématiquement
- Essayez de séparer vos fonctions en un maximum d'opérations logiques différentes (et donc de fonctions différentes), ça limite les erreurs et permet plus de ré-utilisation

1 Fonctions comme objets

Question 0. Currification *Décrassage*

Les fonctions prenant des couples en entrée peuvent aussi être vues comme des fonctions partielles successives de leurs arguments. C'est-à-dire qu'on peut avoir les deux définitions équivalentes suivantes :

```
let plus (x,y) = x + y;;
let plus x y = x + y;;
```

La première a pour type `(int * int) -> int` alors que la deuxième a pour type `int -> int -> int`. La première est dit sous forme "*décurryfiée*" alors que la première est sous forme "*curryfiée*".

Écrire les deux fonctions qui permettent de passer d'une version à l'autre :

```
decurryfie : (('a * 'b) -> 'c) -> ('a -> 'b
-> 'c)
curryfie : ('a -> 'b -> 'c) -> (('a * 'b)
-> 'c)
```

En programmation fonctionnelle, on va souvent préférer la version curryfiée car elle permet des interactions faciles et pratiques. Par exemple on peut facilement définir une fonction qui ajoute 2 à son argument : `let add2 = plus 2;;`. `add2` a alors le type attendu `add2 : int -> int`.

Question 1. Map

Écrire une fonction
`map : ('a -> 'b) -> 'a list -> 'b list`
telle que `map f [`l_1`, ..., `l_n`]` soit la liste `[`$f(l_1)$`, ..., `$f(l_n)$`]`.

L'utiliser pour faire une fonction qui double les éléments d'une liste.

Question 2. Accumulation à gauche

Écrire une fonction
`foldl : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
qui itère sur la liste en accumulant une retenue. Le premier argument est une fonction prenant un élément de la liste en argument, et la valeur de l'accumulateur. Elle renvoie la nouvelle valeur de l'accumulateur. Puis la suite de la liste est traitée séquentiellement.

Cela revient à transformer `foldl f l a` en
`f` l_n `(...(f` l_2 `(f` l_1 `a))...)`.

L'utiliser pour faire une fonction qui calcule la somme des éléments d'une liste.

Question 3. Accumulation à droite

Écrire une fonction
`foldr : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
qui itère sur la liste toujours en accumulant une retenue, mais cette fois "par la droite". Ainsi, `foldr f l a` doit se transformer en `f` l_1 `(f` l_2 `...` `(f` l_n `a))`.

L'utiliser pour faire une fonction qui calcule le maximum des éléments d'une liste d'entiers positifs.

Remarque : `foldl` est préférable à `foldr` lorsque c'est possible car elle est *tail-recursive*.

Question 4. Sélection

En utilisant les fonctions précédentes, écrire une fonction
`select : ('a -> bool) -> 'a list -> 'a list`
qui ne conserve parmi les éléments de la liste passée en argument ceux pour qui le prédicat passé en premier argument vaut `true`.

Question 5. Séparation *Plus difficile*

On définit un type somme comme suit :
`type direction = Left | Right;;`
Le but de la fonction
`split : (direction list) -> 'a list -> ('a list * 'a list)`
est de séparer la liste en entrée en deux listes : une avec les éléments dont l'étiquette dans la liste de directions (premier argument) est `Left`, l'autre avec les éléments `Right`.

On suppose que la liste de directions et la liste des éléments ont toujours la même taille.

Écrire cette fonction en utilisant les fonctions précédentes.

2 Tri paramétré

Les fonctions en paramètres sont particulièrement utiles pour paramétrer un algorithme. Par exemple, dans un tri, on peut vouloir décider de l'ordre dans lequel trier les éléments. Mais l'algorithme de tri en lui-même est indépendant de l'ordre utilisé.

En utilisant un paramètre fonctionnelle à la fonction de tri, un *prédicat*, qui décide lorsqu'un élément est inférieur à un autre, on peut ainsi découpler l'algorithme de tri de la décision de l'ordre.

Par conséquent, on n'est pas obligé d'écrire une fonction de tri par ordre, ce qui serait inutilement long.

Nous allons ici implémenter le tri fusion à l'aide seulement d'outils de programmation fonctionnelle.

Question 1. Fusion

Un prédicat est une fonction `less : 'a -> 'a -> bool` qui renvoie `true` si le premier argument est strictement inférieur au deuxième dans l'ordre représenté.

Écrire une fonction `fusion` prenant en argument deux listes triées selon un prédicat `less` passé en argument, et renvoyant une seule liste triée (toujours selon `less`) contenant l'union des deux autres listes.

Question 2. Tri

Écrire le tri par insertion en utilisant simplement `fusion` et `foldl` et `map`.

Question 3. Exemple

Écrire une fonction qui trie une liste de chaînes de caractères dans l'ordre croissant de la taille des mots.

3 Entiers de Church

Il est même possible de faire de l'arithmétique simplement en utilisant seulement des fonctions. Pour ceci, on définit l'entier n par :

$$\begin{aligned}\widehat{n} &= f \mapsto x \mapsto f^n(x) \\ &= \lambda f.\lambda x.f^n(x)\end{aligned}$$

Ainsi, le type d'un entier sous cette forme est `('a -> 'a) -> 'a -> 'a`. On va donc définir :
`type 'a churchint == ('a -> 'a) -> 'a -> 'a;;`

Question 1. Constantes

Définir à la main $\widehat{0}$, $\widehat{1}$ et $\widehat{2}$.

Question 2. Conversion

Écrire les fonctions de conversion `c_to_int : churchint -> int` et `int_to_cint : int -> churchint`.

Question 3. Addition

Écrire la fonction d'addition `addc : churchint -> churchint -> churchint` qui renvoie $\widehat{x+y}$.

Question 4. Multiplication

Écrire la fonction de multiplication `multc : churchint -> churchint -> churchint` qui renvoie $\widehat{x*y}$.

Question 5. Puissance *Plus difficile*

Écrire la fonction puissance `powc : churchint -> churchint -> churchint` qui renvoie $\widehat{x^y}$.

Question 6. Prédécesseur *Très difficile*

Écrire la fonction prédécesseur `pred : churchint -> churchint` qui à un entier $\widehat{n}$ renvoie $\widehat{n-1}$ (et $\widehat{0}$ si $n = 0$).

En déduire la fonction de soustraction (renvoyant $\widehat{0}$ si le résultat est censé être négatif).