La programmation par continuation

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Un peu de recul sur les Le problème

Le problème

Problème

On veut écrire une fonction `maMap` qui prend en entrée une fonction f et une liste `l`, et qui renvoie la liste des $f(x)$ pour $x \in l$ (avec les éléments dans le même ordre que `l`).

Le problème

```
1  let rec maMap f l = match l with
2    | [] -> []
3    | t::q -> (f t)::(maMap f q)
4  ;;
```

Console

```
# maMap (fun x -> 2*x) [1;2;3];;
- : int list = [2; 4; 6]
```

Exemple

La version la plus simple à écrire (ci-dessus) n'est pas récursive terminale.

En effet, on fait une opération `::` après l'appel récursif.

Le problème

```
1  let maMap f l =
2    let rec map_aux acc l = match l with
3      | [] -> acc
4      | x :: q -> map_aux (f x :: acc) q
5    in
6    map_aux [] l
7  ;;
```

Console

```
# maMap (fun x -> 2*x) [1;2;3];;
- : int list = [6; 4; 2]
```

Exemple

L'astuce usuelle d'utiliser un accumulateur pose problème, car on récupère le résultat à l'envers.

Le problème

```
1  let maMap f l =
2    let rec map_aux acc l = match l with
3      | [] -> List.rev acc
4      | x :: q -> map_aux (f x :: acc) q
5    in
6    map_aux [] l
7  ;;
```

Console

```
# maMap (fun x -> 2*x) [1;2;3];;
- : int list = [2; 4; 6]
```

Exemple

Pour palier à ce problème, on peut utiliser `List.rev` dans le cas de base.

Elle est récursive terminale donc on obtient bien une fonction `maMap` **récursive terminale**, mais ce n'est pas optimal car ça demande un renversement en $O(n)$ à la fin.

Un peu de recul sur les accumulateurs

Un peu de recul sur les accumulateurs

Accumulateurs

En général, dans un programme donné, on va toujours faire la même **opération** sur notre accumulateur, et on va l'initialiser par l'**élément neutre** pour cette opération.

Voici les situations qu'on a rencontré jusqu'à présent :

- on initialise acc à 0 et on fait des additions ;
- on initialise acc à 1 et on fait des multiplications ;
- on initialise acc à `[]` et on fait des `::`.

Un peu de recul sur les accumulateurs

```
1  let somme l =
2    let rec aux l acc = match l with
3      | [] -> acc
4      | x::q -> aux q (acc + x)
5    in aux l 0
6  ;;
7
8  let produit l =
9    let rec aux l acc = match l with
10     | [] -> acc
11     | x::q -> aux q (acc * x)
12   in aux l 1
13 ;;
14
15 let miroir l =
16   let rec aux l acc = match l with
17     | [] -> acc
18     | x::q -> aux q (x::acc)
19   in aux l []
20 ;;
```

Récursivité terminale

Cela rend notre fonction **récursive terminale** : au lieu de faire l'opération après l'appel récursif, on la fait dans l'accumulateur et on termine par l'appel récursif :

↪ rien n'est rajouté dans la pile d'appel.

Un peu de recul sur les accumulateurs

Idée

Mais, puisqu'OCaml est fait pour manipuler des fonctions, on pourrait très bien avoir un accumulateur qui n'est pas un nombre ou une liste, mais une fonction :

- on initialise l'accumulateur à la fonction id ;
- petit à petit, on rajoute des choses dans l'accumulateur en faisant des **compositions** de fonctions.

La programmation par continuation

La programmation par continuation

Continuation

Si jamais ce qu'on doit faire après l'appel récursif est plus complexe qu'une simple opération, on peut en fait représenter tout ce qu'il reste à faire après l'appel récursif par une fonction (qu'on appelle la **continuation** de la fonction : c'est ce qu'il faut continuer à faire après l'appel récursif).

La programmation par continuation

Programmation par continuation

Ainsi, pour rendre notre programme récursif terminal, on peut donc utiliser une fonction comme **accumulateur**, dans laquelle on va accumuler (en faisant des **compositions**) tout ce qu'il reste à faire après les **appels récursifs**.

Au niveau des notations, on note en général cette **continuation** k et non pas acc.

La programmation par continuation

```
1  let maMap f l =
2    let rec map_aux k l = match l with
3      | [] -> k [] (* à la fin, il faut appliquer la fonction k *)
4      | x :: q -> map_aux (fun q' -> k (f x :: q')) q
5          (* dans le premier argument, on définie la nouvelle continuation
6             en fonction de l'ancienne :
7             on compose l'ancienne fonction k avec
8             la fonction qui à la liste q' associe la liste f(x)::q' *)
9    in
10   map_aux (fun q' -> q') l (* au départ, la continuation c'est l'identité *)
11 ;;
```

Console

```
# maMap (fun x -> 2*x) [1;2;3];;
- : int list = [2; 4; 6]
```

Exemple

Voici ce que ça donne pour la fonction maMap.

La programmation par continuation

```
1  let maMap f l =
2    let rec map_aux k l = match l with
3      | [] -> k []
4      | x :: q -> map_aux (fun q' -> k (f x :: q')) q
5    in
6    map_aux (fun q' -> q') l
7  ;;
```

f	k
`[1;2;3]`	id
`[2;3]`	k_1
`[3]`	$k_1 \circ k_2$
`[]`	$k_1 \circ k_2 \circ k_3$

Exemple

Soit f une fonction qu'on donnera à `maMap`, et prenons par exemple `l = [1;2;3]`.

Pour $i \in \mathbb{N}$, on note k_i la fonction `q' -> (f i)::q'`, c'est à dire la fonction qui rajoute $f(i)$ au début de la liste donnée en entrée.

Alors voici les différentes valeurs de k et `l` au fur et à mesure des appels récursifs.

La programmation par continuation

```
1  let maMap f l =
2    let rec map_aux k l = match l with
3      | [] -> k []
4      | x :: q -> map_aux (fun q' -> k (f x :: q')) q
5    in
6    map_aux (fun q' -> q') l
7  ;;
```

f	k
[1;2;3]	id
[2;3]	k_1
[3]	$k_1 \circ k_2$
[]	$k_1 \circ k_2 \circ k_3$

Exemple

À la fin, on renvoie donc :

$$\begin{aligned} k_1 \circ k_2 \circ k_3([]) &= k_1 \circ k_2([f(3)]) \\ &= k_1([f(2); f(3)]) \\ &= [f(1); f(2); f(3)] \end{aligned}$$

Conclusion

Conclusion

- La **programmation par continuation** permet de rendre n'importe quel programme **récursif terminal**, mais ça donne des programmes plus compliqués à écrire.
- Cette technique est complètement **hors programme**, mais vous montre la puissance d'un **langage fonctionnel** comme **OCaml**.