Récursivité terminale

Revenons sur le calcul de la factorielle en CAML :

OCAML
```
let rec factorielle n =
  match n with
  | 0 -> 1
  | _ -> n * factorielle (n - 1)
```

Il y a exactement n appels récursifs, chacun de complexité constante, tout comme le cas de base, et la complexité temporelle est donc linéaire en $\Theta(n)$.

Question 1

Mais quelle est la complexité *spatiale* ?

À première vue, on pourrait penser que celle-ci est constante, puisque les seules opérations sont un filtrage, une soustraction et une multiplication, qui ne consomment pas particulièrement de mémoire supplémentaire. Regardons plus en détail la pile d'appels (que l'on peut obtenir avec `trace` en CAML).

OCAML
```
factorielle <-- 5
  factorielle <-- 4
    factorielle <-- 3
      factorielle <-- 2
        factorielle <-- 1
          factorielle <-- 0
          factorielle --> 1
        factorielle --> 1
      factorielle --> 2
    factorielle --> 6
  factorielle --> 24
factorielle --> 120
```

On s'aperçoit qu'il faut *mémoriser* les appels récursifs pour pouvoir ensuite « remonter » les valeurs calculées. C'est le rôle de la *pile d'exécution*. Comme il y a n appels récursifs, il y a donc besoin d'une capacité mémoire linéaire en $\Theta(n)$ dans la pile.

☞ **Remarque 1**

La réalité est un peu plus complexe, il faut également sauvegarder le contexte (adresse de retour, paramètres et variables locales) puis, au retour de l'appel récursif, restaurer ce contexte. Cela est caché dans la constante du Θ, mais il faut quand même retenir qu'une fonction récursive est a priori moins efficace qu'une version itérative.

Outre l'efficacité, cela pose quand même un sérieux problème :

Exemple 1

OCAML
```
factorielle 1000000;;
Stack overflow during evaluation (looping recursion?).
```

☞ **Remarque 2**

Il y a eu débordement de pile (en anglais stack overflow*).*

Exemple 2

La fonction `length : 'a list -> int` *naïve (celle du cours) ne fonctionne pas si la taille de la liste est supérieure à la limite de la pile de récursion (≈ 130000).*

On aimerait pourtant conserver l'élégance de l'écriture récursive sans sacrifier l'efficacité ni la faisabilité.

Définition 0.1

Une fonction récursive est dite *récursive terminale* si, lorsqu'il y a appel récursif, celui-ci est effectué *uniquement* en tout dernier.

☞ **Remarque 3**

Il ne peut donc y avoir au plus qu'un seul appel récursif.

Exemple 3

La fonction `factorielle` *ci-dessus n'est pas récursive terminale.*

Exemple 4

Le calcul du pgcd de deux entiers à l'aide de l'algorithme d'Euclide est directement récursif terminal :

OCaml
```
let rec pgcd a b =
  match b with
  | 0 -> a
  | _ -> pgcd b (a mod b)
```

On peut souvent se ramener à une fonction récursive terminale à l'aide d'une *fonction auxiliaire* et d'un *accumulateur* :

Exemple 5 (*Factorielle version récursive terminale*)

OCaml
```
let factorielle n =
  (* factorielle_aux n acc renvoie (n! * acc) *)
  let rec factorielle_aux n acc =
    match n with
    | 0 -> acc
    | _ -> factorielle_aux (n - 1) (n * acc)
  in
  factorielle_aux n 1

let _ = factorielle 1000000
- : int = 0   (* ?!? *)
```

☞ **Remarque 4 (*Invariant*)**

`factorielle_aux n acc = factorielle_aux (n - 1) (n * acc)`.

✔ Le retour des appels récursifs se fait par simple transmission, sans calcul supplémentaire.

☞ **Remarque 5**

Les calculs se font « dans l'autre sens ».

Regardons la pile d'appel, où l'on voit bien que cela se fait désormais en espace de pile constant.

OCaml
```
factorielle <-- 5
factorielle_aux <-- 5 1
factorielle_aux <-- 4 5
factorielle_aux <-- 3 20
factorielle_aux <-- 2 60
factorielle_aux <-- 1 120
factorielle_aux <-- 0 120
factorielle_aux --> 120
factorielle --> 120
```

✔ CAML[a] « optimise » automatiquement les fonctions récursives terminales, qui se font alors en espace de pile constant et de manière efficace. On peut voir cela comme une transformation en itération (boucle while).

a. Mais pas PYTHON !

Exemple 6 (*Factorielle version itérative en* PYTHON)

On peut faire cette transformation en algorithme itératif « à la main » :

PYTHON
```
def factorielle(n):
    acc = 1
    k = n
    while (k > 0):
        acc *= k
        k -= 1
    return acc
```

☞ **Remarque 6**

Au concours[1], ne pas chercher à tout prix à avoir des fonctions récursives terminales. Privilégier simplicité et élégance.

1. Sauf à l'épreuve sur machine d'algorithmique des ENS.