Récursivité

MPSI Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

13 mars 2025

Lorsque l'on étudie des fonctions, des algorithmes, des programmes, on s'intéresse à plusieurs aspects :

— La **terminaison**, c'est à dire le fait de s'arrêter éventuellement ;

— La **correction**, c'est à dire le fait d'avoir le comportement attendu ;

— La **complexité**, c'est à dire le temps d'exécution.

Comme on ne dispose pas de boucles en OCaml pour le moment, on s'intéressera à l'étude de fonctions récursives.

1 Terminaison et correction

Lorsque l'on écrit un commentaire de fonction, on donne sa **spécification**, c'est à dire que l'on décrit le comportement que cette fonction doit avoir. Rien ne permet de garantir a priori que la fonction suit bien ce comportement : il faut le prouver.
Considérons un exemple simple : la multiplication :

```
1 (* Renvoie x fois y, pour x, y entiers positifs *)
2 let rec mult (x: int) (y: int) : int =
3   if x = 0 then 0
4   else y + mult (x-1) y
```

Notons cette fonction de manière plus mathématique, et étudions-la :

$$\begin{array}{rccl} & \mathbb{N} & \longrightarrow & \mathbb{N} \\ \textbf{mult} \ : & x, y & \mapsto & \begin{cases} 0 & \text{si } y = 0 \\ x + \textbf{mult}(x, y-1) & \text{sinon} \end{cases} \end{array}$$

On dira que cette fonction est **correcte** si elle :

— termine sur toutes ses entrées ;

— renvoie bien ce qui est indiqué dans sa documentation, i.e. si $\forall x, y \in \mathbb{N}, mult(x, y) = xy$.

On remarque qu'un appel à $mult(x, y)$ avec $y > 0$ cause un appel récursif $mult(x, y-1)$. Autrement dit, si l'on regarde la suite des appels récursifs, on remarque que les appels se font avec y entier, positif, strictement décroissant. D'après le principe de descente infinie de Fermat, il y a donc nécessairement un nombre fini d'appels récursifs. Ceci permet de garantir la terminaison de la fonction. Montrons maintenant qu'elle est correcte, i.e. que :

$$\forall x \in \mathbb{N}, \forall y \in \mathbb{N}, \textbf{mult}(x, y) = x \times y$$

Pour cela, posons $x \in \mathbb{N}$, et montrons par récurrence que :

$$\forall y \in \mathbb{N}, P(y) : \textbf{mult}(x, y) = x \times y$$

— Pour $y = 0$, $\textbf{mult}(x, 0) = 0 : P(0)$ est vraie.

— Soit $y \in \mathbb{N}^*$, supposons $P(y-1)$. Alors, $\textbf{mult}(x, y-1) = x \times (n-1)$. Or, $y > 0$ donc $\textbf{mult}(x, y) = x + \textbf{mult}(x, y-1) = x + x \times (y-1)$ par hypothèse de récurrence. Donc, $\textbf{mult}(x, y) = x \times y : P(y)$ est vraie

Voyons un exemple plus complexe, en reprenant la procédure d'insertion utilisée dans le tri par insertion :

```
1 (* Copie de l, où x a été inséré dans l'ordre.
2    l doit être croissante *)
3 let rec insert (x: 'a list) (l: 'a list) : 'a list = match l with
4   | [] -> [x]
5   | y::q -> if x < y then x :: l
6             else          y :: insert x q ;;
```

Commençons par la terminaison. On note $|L|$ la taille d'une liste L. On remarque que lors du calcul de **insert**(x, L), la suite des valeurs successives de $|L|$ lors des appels récursifs est strictement décroissante, entière et positive. On peut dont à nouveau conclure que cette fonction termine.

Montrons maintenant la correction. On raisonne par récurrence sur la taille de la liste L en entrée. On pose x un élément à insérer. Montrons la propriété suivante par récurrence :

$\forall n \in \mathbb{N}, \forall L$ liste de taille n, si L est triée, alors **insert**(x, L) est contient les éléments de L ainsi que x, et est triée.

- Si $|L| = 0$ alors $L = []$, et $[x]$ contient bien x ainsi que les éléments de $[]$, et est triée.
- Soit L une liste avec $|L| > 0$. On écrit $L = y :: Q$. Si $x < y$ alors $x :: y :: Q = x :: L$ est bien une copie de L où x a été ajouté, triée. Sinon, par hypothèse de récurrence, **insert**(x, Q) est une copie triée de Q où x a été ajouté, et comme L est triée, $y \leq \min Q$ et $y \leq x$. Donc, $y ::$ **insert**(x, Q) est triée, et contient bien x ainsi que les éléments de L, à savoir y et les éléments de Q.

En résumé Pour montrer qu'une fonction récursive termine, on doit exhiber une quantité entière, positive, et strictement décroissante lors des appels récursifs.

Pour montrer la correction, il faut commencer par identifier ce que la fonction doit faire, i.e. sa spécification, et en déduire une propriété mathématique sur la fonction, que l'on montre alors par récurrence.

2 Complexité

La complexité d'une fonction ou d'un algorithme permet d'évaluer son temps d'exécution. Elle ne s'exprime pas en secondes, en millisecondes, ou autre durée, mais en nombre d'**opérations élémentaires**. On considèrera que les opérations élémentaires sont les additions, soustractions, tests booléens, match with, et de manière générale toutes les opérations "simples" dont on dispose en OCaml.

Définition 1. Soit f une fonction informatique. Pour $n \in \mathbb{N}$, on note $C_f(n)$ le nombre maximal d'opérations élémentaires effectué par f sur une entrée de taille n.

Prenons par exemple la fonction d'insertion :

```
let rec insert (x: 'a list) (l: 'a list) : 'a list = match l with
  | [] -> [x]
  | y::q -> if x < y then x :: l
            else          y :: insert x q ;;
```

$C_{\text{insert}}(n)$ est le coût maximal de la fonction `insert` sur une liste de taille n. Pour calculer `insert x l` avec l de taille n, on doit :

- Faire un match-with (1 opération)
- Tester l'inégalité $x < y$ (1 opération)
- Dans un cas, fabriquer la liste $x :: l$ (1 opération)
- Dans l'autre, rappeler récursivement la fonction, et utiliser :: (1 opération + 1 appel récursif sur une liste de taille $n-1$)

Si l'on considère le pire cas en terme de coût, on peut supposer que l'exécution emprunte toujours la branche "else", et cause un appel récursif. Alors, on obtient l'équation :

$$C(n) = 3 + C(n-1)$$

Ainsi, $C(n) = 3n + C(0)$. En pratique, on s'intéresse seulement à l'ordre de grandeur des complexités, et pas à leurs valeurs précises. Pour cela, on utilise la notation de Landau $\mathcal{O}$, que vous avez vu en mathématiques : pour deux suites $(u_n)_{n\in\mathbb{N}}, (v_n)_{n\in\mathbb{N}}$ de réels positifs, $u_n = \mathcal{O}(v_n)$ signifie que u est bornée par une constante fois v à partir d'un certain rang. Ainsi, on dira simplement que la fonction d'insertion a une complexité en $\mathcal{O}(n)$, ce que l'on appelle une complexité **linéaire**.

On dit que l'on étudie la complexité **pire cas, asymptotique**.

La méthode la plus générale pour analyser la complexité d'une fonction récursive est donc de trouver une relation de récurrence sur la complexité, et de résoudre cette relation.

Exercice 1. On rappelle le code du tri par insertion :

```
let rec insert_sort (l: 'a list) : 'a list =
  match l with
  | [] -> []
  | x :: q -> insert x (insert_sort q)
```

Proposer une relation de récurrence vérifiée par la complexité, et la résoudre.

Exercice 2. On propose le code suivant OCaml pour le tri fusion :

```
(* renvoie deux listes l1 l2 de même taille à 1 près,
   contenant les éléments de l *)
let rec separer (l: 'a list) : 'a list * 'a list =
  match l with
  | [] -> [], []
  | [x] -> [x], []
  | x::y::q -> let l1, l2 = separer q in x::l1, y::l2

(* Fusionne l1 et l2 deux listes supposées triées par ordre croissant *)
let rec fusion (l1: 'a list) (l2: 'a list) : 'a list =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1::q1, x2::q2 -> if x1 < x2 then x1 :: fusion q1 l2
                      else x2 :: fusion l1 q2

let rec tri_fusion (l : 'a list) : 'a list =
  match l with
  | [] | [x] -> l
  | _ -> let l1, l2 = separer l in fusion (tri_fusion l1) (tri_fusion l2)
```

Question 1. Donner la complexité des fonctions `separer` et `fusion` en fonction de la taille de leurs arguments.

Question 2. En notant C_n le nombre d'opérations de `tri_fusion` sur une liste de taille n, justifier que C_n vérifie la relation de récurrence suivante :

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n)$$

On se propose d'étudier la relation simplifiée suivante :

$$C_n = 2C_{\frac{n}{2}} + An$$

Question 3. Pour $n = 2^p$ une puissance de 2, résoudre la récurrence de manière exacte.

Question 4. En admettant que $(C_n)_{n\in\mathbb{N}}$ est croissante, en conclure la complexité asymptotique du tri fusion.

En résumé Pour déterminer la complexité asymptotique d'une fonction récursive, on commence par trouver une relation de récurrence en analysant le code et en comptant les opérations effectuées. Puis, on résout cette récurrence pour trouver une expression asymptotique.

Quelques complexités classiques :

— $\mathcal{O}(1)$: constante
— $\mathcal{O}(n)$: linéaire (recherche du maximum d'une liste, somme des éléments d'une liste, etc...)
— $\mathcal{O}(n^2)$: quadratique (tri insertion, tri sélection)
— $\mathcal{O}(n^k)$ avec $k \in \mathbb{N}$: polynomiale
— $\mathcal{O}(\log n)$: logarithmique
— $\mathcal{O}(a^n)$ avec $a > 1$: exponentielle

3 Récursivité terminale

Considérons la fonction suivante :

```
1 (* factorielle n = n! *)
2 let rec factorielle n =
3   if n = 0 then 1
4   else n * factorielle (n-1)
```

Pour évaluer `factorielle 3`, il faut :

1. Calculer factorielle 2 :
   (a) Calculer factorielle 2 :
      i. Caluler factorielle 1
         A. Caluler factorielle 0
         B. On renvoie donc 1
      ii. On obtient 1, on multiplie par 1, on renvoie donc 1
   (b) On obtient 1, on multiplie par 2, on renvoie donc 2
2. On obtient 2, on multiplie par 3, on renvoie donc 6

On remarque qu'il y a deux phases dans les appels : une première phase de descente, où l'on crée les appels récursifs, et une phase de remontée, où l'on sort des appels récursifs. De plus c'est dans la phase de remontée que les calculs sont faits. Il faut donc **se souvenir** de tous les appels récursifs qui ont été faits, afin de savoir qu'à la phase de remontée, il faudra multiplier par 1, par 2, par 3, etc... Ainsi, lorsque l'ordinateur exécute `factorielle n`, il doit réserver des endroits dans la mémoire pour **chaque appel récursif**[1]. Autrement dit, il faut une mémoire $\mathcal{O}(n)$. La mémoire de votre ordinateur étant finie, on ne pourra pas utiliser la fonction `factorielle` sur des entrées arbitrairement grandes. Si l'on essaie d'exécuter `factorielle 100000000`, on déclenche une erreur indiquant que l'ordinateur n'a plus de mémoire pour exécuter le programme.

Étudions maintenant la fonction suivante :

```
1 (* factorielle_bis n accu = n! × accu *)
2 let rec factorielle_bis n accu =
3   if n = 0 then accu
4   else factorielle_bis (n-1) (accu*n)
5
6 (* On remarque que factorielle_bis n 1 = factorielle n *)
```

Pour évaluer `factorielle_bis 3 1`, on doit :

1. Calculer factorielle_bis 3 1
   (a) Calculer factorielle_bis 2 3
      i. Calculer factorielle_bis 1 6
         A. Calculer factorielle_bis 0 6
         B. On renvoie donc 6
      ii. On renvoie donc 6
   (b) On renvoie donc 6
2. On renvoie donc 6

1. La zone mémoire dédiée aux données des appels de fonction s'appelle la **pile d'appel**, mais l'étude précise de son fonctionnement sort un peu du cadre du programme.

Ici, la phase de remontée est triviale : elle sert juste à transmettre le résultat vers l'appel initial. Autrement dit, on n'a pas besoin de sauvegarder les différentes données des appels récursifs. OCaml est capable de détecter cela, et réutilise l'espace. Ceci permet d'exécuter la fonction `factorielle_bis` en espace constant, peu importe la valeur de l'entrée. Si l'on évalue `factorielle_bis 100000000 1`, aucun problème de pile !

Définition 2. On appelle fonction ***récursive terminale*** (en anglais : ***tail-recursive***) toute fonction récursive ne nécessitant aucun traitement à la remontée d'une valeur.

Proposition 1. Une fonction récursive terminale peut s'exécuter en n'utilisant qu'une mémoire de taille constante pour gérer les appels récursifs.

Certains langages comme OCaml peuvent détecter et gérer automatiquement les fonctions récursives terminales.

Remarquons que pour `factorielle_bis`, on n'écrit pas directement la fonction factorielle, mais une fonction ***plus générale***, que l'on applique avec un bon paramètre ensuite. La programmation récursive terminale nécessite souvent d'utiliser une méthode par ***accumulateur*** qui consiste à stocker le résultat temporaire dans un ou plusieurs des paramètres, appelés les accumulateurs.

Exemple 1. Écrivons une fonction récursive terminale calculant la somme d'une liste d'entiers. On rappelle la version basique :

```
let rec somme (l: int list) : int =
  match l with
  | [] -> 0
  | x :: q -> x + somme q
```

On introduit un nouveau paramètre qui va servir à stocker la somme des éléments déjà vus :

```
(* Renvoie la somme des éléments de l, plus r *)
let rec somme_plus (l: int list) (r: int) : int =
  match l with
  | [] -> r
  | x :: q -> somme_plus l (x + r)

let somme l = somme_plus l 0
```

Exercice 3.

Écrivons quelques fonctions récursives sur les listes, en cherchant systématiquement une version récursive terminale.

Question 1. Définir une fonction `longueur` permettant de calculer la longueur d'une liste.

Question 2. Définir une fonction `reverse` permettant de renverser une liste.

Question 3. La fonction `map: ('a -> 'b) -> 'a list -> 'b list` qui applique une fonction à tous les éléments d'une liste.

Question 4. Une fonction `fibonacci: int -> int` calculant les termes de la suite de Fibonacci.