Option informatique, cours 2

Théorie de la récursivité

C. MULLAERT

Lycée Saint-Louis

Année 2025-2026

Plan du cours

Fonctions récursives, le problème de la terminaison

Première définition

On dit qu'une fonction est **récursive** lorsque le nom de la fonction apparait dans sa définition. On définit une telle fonction avec la liaison `let rec`, comme dans l'exemple suivant :

```
let rec puissance x = function
        | 0 -> 1
        | n -> x*puissance x (n-1);;
```

Si l'on omet le mot clé `rec`, on obtient un message d'erreur car le nom puissance n'est pas défini.

Première définition

En tant que languague **fonctionnel**, Ocaml fait un usage intensif des fonctions **récursives** (à la place des structures itératives comme les boucles).

La définition d'une fonction récursive pose plusieurs questions :

- Cette écriture définit-elle bien une fonction (d'un point de vue mathématique) ?
- Lors de l'exécution, le programme va-t-il se terminer (point de vue de l'exécution) ?
- La valeur de retour est-elle conforme à ce que l'on attend (correction) ?
- Quelle est la complexité (ressources consommées) de la fonction ?

Terminaison sur un exemple

On reprend notre exemple initial :

```
let rec puissance x = function
        | 0 -> 1
        | n -> x*puissance x (n-1);;
```

L'évaluation de l'expression `puissance 2 10` va nécessiter la valeur de `puissance 2 9`, et ainsi de suite jusqu'à l'évaluation de `puissance 2 0` qui vaut 1.

On peut donc donner un sens à toutes ces quantités ainsi qu'à `puissance x n` pour tout couple $(x, n) \in \mathbb{R} \times \mathbb{N}$ à l'aide d'un raisonnement par **récurrence**.

Terminaison en général

Malheureusement, il n'existe pas d'algorithme qui permette de déterminer si une fonction récursive va se terminer ou non. On dit que le problème de l'arrêt est **indécidable**.

En effet, les fonctions de type `int -> int` forment un ensemble fini. Il existe donc une surjection ϕ de $\mathbb{N}$ dans cet ensemble.
Supposons par l'absurde qu'il existe une fonction `termine` de type `int -> int -> bool` qui prenne en paramètre deux entiers n et p et qui renvoie `true` si le calcul de $\phi(n)(p)$ se termine, et `false` sinon.
Considérons alors la fonction suivante

```
let rec f p = match (termine p p) with
        |true -> f p
        |_    -> 0
```

Il existe alors r tel que $f = \phi(r)$. L'étude de la valeur de `termine r r` conduit alors à une contradiction.

Terminaison en général

La terminaison d'un algorithme récursif doit donc se justifier **au cas par cas**.

Le plus souvent, on a un ensemble de cas de base A sur lequel la fonction est bien définie (sans appel récursif), et on cherche à la **prolonger** à un ensemble plus grand E en utilisant le fait que la suite des arguments des appels récursifs successifs va finir par appartenir à A.

Terminaison en général

Il arrive que la terminaison d'un algorithme récursif soit un problème ouvert. Voici deux fonctions dont on ne sait pas prouver qu'elles terminent sur $\mathbb{N}^*$.

```
(* Fonction de Syracuse *)
let rec syracuse = function
        |1 -> 1
        |n when (n mod 2)=0 -> syracuse(n/2)
        |n -> syracuse(3*n+1);;

(* Fonction q de Hofstadter *)
let rec q = function
        | 1 -> 1
        | 2 -> 1
        | n -> q (n - q (n-1)) + q (n - q (n-2));;
```

Terminaison en général

On reprend notre exemple initial :

```
let rec puissance x = function
        | 0 -> 1
        | k -> x*puissance x (k-1);;
```

Dans ce cas, on comprend que la fonction est bien définie car les appels récursifs se font avec des arguments qui diminuent à chaque fois de 1. On finit donc nécessairement par évaluer le cas de base `puissance x 0` qui est bien défini.

On va tenter de formaliser cela pour une classe fréquente de fonctions récursives avec la notion d'**ordre bien fondé**.

Ordre, élément minimal et plus petit élément d'une partie

Soit $\preceq$ une relation d'ordre sur un ensemble E. On dit que $(E, \preceq)$ est un ensemble ordonné. Soit A une partie de E et $a \in A$.

- On dit que a est un **élément minimal** de A lorsque :

$$\forall x \in A, \quad x \preceq a \Rightarrow x = a$$

- On dit que a est le **plus petit élément** de A lorsque

$$\forall x \in A, \quad a \preceq x$$

Si l'ordre est total et A est finie non vide, alors A possède un plus petit élément.

Soit F un ensemble non vide et $E = \mathcal{P}(F)$ muni de la relation d'inclusion. On peut montrer que E possède un plus petit élément (l'ensemble vide), mais que $E \setminus \{\emptyset\}$ ne possède pas de plus petit élément, bien qu'il possède des éléments minimaux (les singletons).

Ordre bien fondé

Soit $(E, \preceq)$ un ensemble ordonné. On dit que $(E, \preceq)$ est **bien fondé** lorsque toute partie non vide de E possède un élément minimal. Il est équivalent de dire qu'il n'existe pas de suite strictement décroissante.

Les ensembles ordonnés suivants sont bien fondés :

1. $(\mathbb{N}, \leq)$
2. $(\mathbb{N}^2, \preceq)$ avec $\preceq$ définie par $\forall (n_1, p_1, n_2, p_2) \in \mathbb{N}^4$, $(n_1, p_1) \preceq (n_2, p_2)$ si $n_1 \leq n_2$ et $p_1 \leq p_2$ (ordre produit).
3. $(\mathbb{N}^2, \preceq)$ avec $\preceq$ définie par $\forall (n_1, p_1, n_2, p_2) \in \mathbb{N}^4$, $(n_1, p_1) \preceq (n_2, p_2)$ si $n_1 < n_2$ ou $(n_1 = n_2$ et $p_1 \leq p_2)$ (ordre lexicographique).

On peut, de même, à partir d'une relation d'ordre bien fondée sur un ensemble E construire deux relations d'ordre bien fondées sur E^n.

Définition des fonctions inductives

Soit $(E, \preceq)$ un ensemble bien fondé, A une partie non vide de E et $\phi : E \setminus A \to E$ telle que $\forall x \in E \setminus A$, $\phi(x) \prec x$. Soit g une application de A dans F et h une application de F dans F.

D'après ce qui précède, l'écriture suivante définit bien une application de E dans F :

$$f : x \mapsto \begin{cases} g(x) & \text{si } x \in A \\ h(f(\phi(x))) & \text{sinon} \end{cases}$$

En Ocaml, on a prouvé que la fonction suivante termine bien:

```
let rec f x = match (appartientaA x) with
        | true -> g x
        | _ -> h (f (phi x));;
```

Définition des fonctions inductives

De la même façon, si g est une application de A dans F et h une application de $(E \setminus A) \times F$ dans F, l'application suivante est bien définie sur E:

$$f : x \mapsto \begin{cases} g(x) & \text{si } x \in A \\ h(x, f(\phi(x))) & \text{sinon} \end{cases}$$

En Ocaml, on a prouvé que la fonction suivante termine bien:

```
let rec f x = match (appartientaA x) with
        | true -> g x
        | _ -> h x (f (phi x)) ;;
```

Définition des fonctions inductives

On peut aussi généraliser à plusieurs appels récursifs et définir

$$f : x \mapsto \begin{cases} g(x) & \text{si } x \in A \\ h(x, f(\phi_1(x)), f(\phi_2(x))) & \text{sinon} \end{cases}$$

si ϕ_1 et ϕ_2 sont deux applications de $E \setminus A$ dans E telle que $\forall x \in E \setminus A$, $\phi_1(x) \prec x$ et $\phi_2(x) \prec x$.

En Ocaml, on a prouvé que la fonction suivante termine bien:

```
let rec f x = match (appartientaA x) with
        | true -> g x
        | _ -> h x (f (phi1 x)) (f (phi2 x)) ;;
```

Applications : preuves de terminaison

Ce formalisme permet de montrer la terminaison de beaucoup de fonctions récursives. Voici quelques exemples :

```
let rec puissance x = function
        | 0 -> 1
        | n -> x*puissance x (n-1);;

let rec fact = function
        | 0 -> 1
        | n -> n*fact (n-1);;
```

Ces cas se traitent en remarquant que l'ensemble $(\mathbb{N}, \leq)$ est bien fondé et en prenant $A = \{0\}$ et $\phi : n \mapsto n - 1$.
On montre ainsi que l'on peut calculer le n-ième terme d'une suite récurrente d'ordre 1 : u_0 est fixé et pour tout $n \in \mathbb{N}$, $u_{n+1} = f(u_n)$.

Applications : preuves de terminaison

```
(* suite de Fibonacci *)
let rec fibo = function
         | 0 -> 1
         | 1 -> 1
         | n -> fibo (n-1) + fibo (n-2);;
```

Ce cas se traite en remarquant que l'ensemble $(\mathbb{N}, \leq)$ est bien fondé et en prenant $A = \{0, 1\}$, $\phi_1 : n \mapsto n - 1$ et $\phi_2 : n \mapsto n - 2$. On montre ainsi que l'on peut calculer le n-ième terme d'une suite récurrente d'ordre 2 : u_0 et u_1 sont fixés et pour tout $n \in \mathbb{N}$, on a $u_{n+2} = f(u_n, u_{n+1})$.

Applications : preuves de terminaison

```
(* pgcd - Algorithme d'Euclide *)
let rec pgcd p q = match q with
        | 0 -> p
        | _ -> pgcd q (p mod q);;
```

Ce cas se traite en remarquant que l'ensemble $\mathbb{N}^2$ muni de l'ordre lexicographique "inversé" est bien fondé et en prenant $A = \{(p, 0), p \in \mathbb{N}\}$ et $\phi : (p, q) \mapsto (q, p \bmod q)$.

Preuve de correction

Une fois convaincu qu'une fonction récursive est bien définie, se pose la question de sa **correction**. La preuve de correction fait souvent intervenir une quantité qui reste invariante au fil des appels récursifs, et que l'on propage par récurrence (éventuellement étendue aux ensembles bien fondés).

Preuve de correction

Soit $(E, \preceq)$ un ensemble bien fondé, A une partie non vide de E et $\phi : E \setminus A \to E$ telle que $\forall x \in E \setminus A,\ \phi(x) \prec x$.

On considère un prédicat H sur E, c'est-à-dire une application de E dans $\{Vrai, Faux\}$ telle que :

- $\forall a \in A,\ H(a)$,
- $\forall x \in E \setminus A,\ H(\phi(x)) \Rightarrow H(x)$

Alors pour tout $x \in E$, on a $H(x)$.

Preuve de correction : exemples

```
let rec puissance x = function
        | 0 -> 1
        | n -> x*puissance x (n-1);;
```

La correction s'obtient en considérant le prédicat défini, pour tout $n \in \mathbb{N}$, par $H(n) =$ «`puissance x n` renvoie x^n. »

- $H(0)$ est vrai à cause du premier motif de filtrage
- Si $n \in \mathbb{N}^*$ et si $H(n-1)$ est vrai, alors, grâce au second motif, `puissance x n` renvoie $x * x^{n-1}$ donc on a $H(n)$.

Le raisonnement est strictement identique pour la correction de la fonction factorielle. C'est un raisonnement par récurrence.

Preuve de correction : exemples

```
(* pgcd - Algorithme d'Euclide *)
let rec pgcd p q = match (p,q) with
        | _ when (q=0) -> p
        | _ -> pgcd q (p mod q);;
```

La correction s'obtient en posant, pour tout $(p, q) \in \mathbb{N}^2$,

$$H(p, q) = \text{«pgcd p q renvoie } p \wedge q. \text{ »}$$

- Pour tout $p \in \mathbb{N}$, $H(p, 0)$ est vrai à cause du premier motif de filtrage
- Soit $(p, q) \in \mathbb{N}^2$ tel que $q \in \mathbb{N}^*$ et $H(q, p \bmod q)$ soit vrai. Alors, grâce au second motif, pgcd p q renvoie $q \wedge (p \bmod q)$, c'est-à-dire $p \wedge q$, ce qui prouve $H(p, q)$.

Il s'agit en fait d'une récurrence simple sur q.

Notion de récursivité terminale et pile d'exécution

Difficultés liées à l'écriture de fonctions récursives

L'écriture de fonctions récursives, bien qu'élégante, peut poser des problèmes lors de l'exécution du programme.

```
(* suite de Fibonacci *)
let rec fibo = function
        | 0 -> 1
        | 1 -> 1
        | n -> fibo (n-1) + fibo (n-2);;
```

Sur cet exemple, l'évaluation de `fibo 50;;` est très lente, bien que la fonction soit parfaitement définie sur le plan théorique. La raison est le grand nombre d'appels récursifs.

On parle de complexité exponentielle, nous verrons cela dans la troisième partie du cours.

Difficultés liées à l'écriture de fonctions récursives

Autre curiosité : voici deux fonctions apparemment très similaires

```
let rec f x = print_int x; print_newline (); f (x+1);;
let rec g x = print_int x; print_newline (); 1 + g (x+1);;
```

Aucune des deux n'est bien définie. Mais l'évaluation de `g 1;;` déclenche une erreur après 200000 appels récursifs, alors que la première fonction continue sans problème bien au-delà.

Pour comprendre ce phénomène, il faut comprendre ce qui se passe lorsqu'un appel récursif est exécuté par l'ordinateur et cela peut varier en fonction de son architecture.

On retiendra que les fonction **récursives terminales** sont plus efficaces que les autres (aussi efficaces que leur équivalent itératif).

Fonction récursive terminale

On dit qu'une fonction récursive est terminale si elle s'écrit de la façon suivante :

$$f : x \mapsto \begin{cases} g(x) & \text{si } x \in A \\ f(\phi(x)) & \text{sinon} \end{cases}$$

En Ocaml, cela donne :

```
let rec f x = match (appartienA x) with
        |true  -> g x
        | _ -> f (phi x)
```

Dans cette écriture, l'appel récursif est la dernière instruction qui est réalisée.

Fonction récursive terminale

Sur ces deux exemples,

```
let rec f x = print_int x; print_newline (); f (x+1);;
let rec g x = print_int x; print_newline (); 1 + g (x+1);;
```

La fonction `f` est sous forme récursive terminale, mais pas la fonction `g` (mais aucune des deux n'est correctement définie).

Lors de l'appel de `g 1;;`, l'ordinateur doit se souvenir de faire l'opération `1+...` avec le résultat de l'appel récursuif suivant.

Toutes ces informations s'accumulent en mémoire à mesure que l'on augmente le nombre d'appels récursifs. Cela ne se produit pas pour l'appel `f 1;;`.

Notion de pile d'exécution

Considérons ces deux versions de la fonction puissance :

```
let rec puissance x = function
    | 0 -> 1
    | n -> x * puissance x (n-1);;

let puissance_terminale x n =
  let rec aux acc = function
    | 0 -> acc
    | n -> aux (acc*x) (n-1)
  in aux 1 n;;
```

Elles sont toutes les deux bien définies et donnent le même résultat.
La deuxième version est récursive terminale.

Mise sous forme récursive terminale

Comment faire pour mettre sous forme récursive terminale une fonction qui ne l'est pas ? Prenons l'exemple de la fonction factorielle :

```
let rec fact = function
    | 0 -> 1
    | n -> n * fact (n-1);;
```

L'idée générale est d'introduire une fonction auxiliaire, qui prend un argument de plus (appelé accumulateur). Habituellement, on définit cette fonction auxiliaire dans le corps de la fonction : on dit qu'on l'encapsule.

L'accumulateur contiendra les produits partiels $n(n-1)...(n-k)$. au lieu de les stocker dans l'attente de l'évaluation de la fonction récursive, ce produit est passé en argument supplémentaire de la fonction récursive.

Mise sous forme récursive terminale : exemples

On obtient la fonction suivante :

```
let fact_terminale n =
  let rec aux acc = function
    |0 -> acc
    |n -> aux (acc*n) (n-1)
  in aux 1 n;;
```

On note que cette forme se rapproche d'une écriture itérative (avec une boucle `for` par exemple). Le compilateur le voit et optimise le code afin que l'exécution ne fasse plus appel à des appels récursifs.

En Ocaml, l'écriture d'une fonction récursive terminale n'est qu'un **style d'écriture**. Pour les fonctions récursives non terminales, cette optimisation n'est pas possible.

Complexité d'une fonction récursive

Difficultés liées à l'écriture de fonctions récursives

On a vu que la fonction fibo ci dessous, en plus de ne pas être récursive terminale, est très lente à cause du grand nombre d'appels récursifs inutiles qui sont réalisés.

```
(* suite de Fibonacci *)
let rec fibo = function
  | 0 -> 1
  | 1 -> 1
  | n -> fibo (n-1) + fibo (n-2);;
```

On va tenter de quantifier cela en parlant de **complexité algorithmique**.

Exemples de calculs de complexité

Si l'on note $C(n)$ le nombre d'additions nécessaires au calcul de fibo n, on a $C(0) = C(1) = 0$ et, pour tout $n \geq 2$,

$$C(n) = C(n-1) + C(n-2) + 1.$$

On en déduit que pour tout entier n,

$$C(n) = f_n - 1 = \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} + \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - 1$$

puis que

$$C(n) \sim \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1}.$$

On parle de complexité exponentielle.

Exemples de calculs de complexité

Pour le calcul de $n!$, en notant $C(n)$ le nombre de multiplications, on a $C(n) = C(n-1) + 1$, on a une complexité linéaire. C'est également le cas de l'algorithme naïf du calcul de x^n.
En utilisant un algorithme d'exponentiation rapide

```
let rec puissance_expo x n =
  if n=0 then 1 else match n mod 2 with
    | 0 -> let y = puissance_expo x (n/2) in y*y
    | 1 -> let y = puissance_expo x (n/2) in x*y*y;;
```

on a en notant $C(n)$ le nombre de multiplication nécessaire au calcul de x^n, $1 + C(n//2) \leq C(n) \leq C(n//2) + 2$, ce qui donne une complexité logarithmique.

Exemples de calculs de complexité

En général, lorsque l'on utilise une stratégie dite de «diviser pour régner», la complexité obtenue avec la méthode vérifie une relation de récurrence de la forme

$$C(n) = a\,C\left(n//b\right) + O(n^c)$$

où a est le nombre de sous-problèmes considérés, b le rapport de taille entre un sous-problème et le problème initial, et $O(n^c)$ le coût de création des sous-problèmes et de combinaison des résultats.

On peut alors montrer le résultat suivant (Théorème "maître") :

- Si $a > b^c$, alors $C(n) = O(n^{\log_b(a)})$
- Si $a < b^c$, alors $C(n) = O(n^c)$
- Si $a = b^c$, alors $C(n) = O(n^c \log(n))$

Quelques exercices

1. Justifier que la fonction d'Ackermann définie sur $\mathbb{N}^2$ par

$$A : (n,p) \mapsto \begin{cases} p+1 & \text{si } n = 0 \\ A(n-1,1) & \text{si } p = 0 \\ A(n-1, A(n,p-1)) & \text{sinon} \end{cases}$$

est bien définie et écrire une telle fonction.

2. Écrire une fonction `quotient_reste` prenant en argument deux entiers n et p et renvoyant le quotient et le reste de la division euclidienne de n par p.
Écrire une version récursive terminale. Vérifier la terminaison et la correction.

3. Écrire une fonction `binomial` prenant en argument deux entiers k et n et renvoyant $\binom{n}{k}$.
Écrire une version récursive terminale. Vérifier la terminaison et la correction.