module 5 :

Récursivité

Nous allons maintenant donner un cadre formel à la notion de récursivité. Ce cadre n'est pas le plus général, mais il permettra néanmoins de décrire la plus-part des fonctions récursives que nous serons amené à rencontrer, et nous démontrerons, dans un cadre un peu restrictif, que tout algorithme récursif possède une version itérative.

• **Récurrence et récursivité**

Les algorithmes récursifs que nous avons rencontré jusqu'à présent sont intimement liés à la notion de récurrence, dont on peut rappeler le principe :

Théorème (principe de récurrence simple) *Soit P un prédicat défini sur $\mathbb{N}$, tel que $P(0)$ est vrai, ainsi que l'implication, $\forall n \in \mathbb{N}$, $P(n) \Rightarrow P(n+1)$. Alors pour tout entier $n \in \mathbb{N}$, $P(n)$ est vrai.*

Ce principe de récurrence permet de démontrer que la donnée d'un élément $a \in E$ et d'une application $f : E \to E$ permet de définir sans ambiguïté une suite $(u_n)_{n\in\mathbb{N}}$ à l'aide des relations :

$$u_0 = a \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+1} = f(u_n).$$

C'est suivant ce principe qu'on été définies les fonctions récursives jusqu'à présent. Nous allons donc commencer par généraliser la notion de récurrence.

1. Fonctions inductives

Définition *On dit qu'un ensemble ordonné $(E, \preccurlyeq)$ est* bien ordonné *lorsque toute partie non vide a un plus petit élément. On dit alors que $\preccurlyeq$ est un* bon ordre.

Commençons par énoncer quelques résultats généraux.

Théorème *Un bon ordre est total.*

Preuve Si $(E, \preccurlyeq)$ est bien ordonné, alors pour tout couple $(a, b) \in E^2$, l'ensemble $\{a, b\}$ admet un plus petit élément, donc $a \preccurlyeq b$ ou $b \preccurlyeq a$.

Théorème *Tout ensemble bien ordonné possède un plus petit élément.*

Par exemple, $(\mathbb{N}, \leqslant)$ est bien ordonné, mais pas $(\mathbb{Z}, \leqslant)$. Voyons maintenant un exemple important :

• **Ordre lexicographique**

Considérons l'ordre défini sur $\mathbb{N}^2$ par :

$$(n, p) \preccurlyeq (n', p') \iff n < n' \text{ ou } (n = n' \text{ et } p \leqslant p').$$

Il s'agit d'un bon ordre car si A est une partie non vide de $\mathbb{N}^2$, A possède un plus petit élément (n_0, p_0) défini par :

$$n_0 = \min\big\{n \in \mathbb{N} \,/\, (n, p) \in A\big\} \quad \text{et} \quad p_0 = \min\big\{p \,/\, (n_0, p) \in A\big\}.$$

Cet ordre est appelé *ordre lexicographique* sur $\mathbb{N}^2$, par référence à l'ordre lexicographique défini à partir de l'ordre alphabétique. Plus généralement, on peut définir l'ordre lexicographique sur l'ensemble des suites finies d'éléments d'un ensemble bien ordonné.
Le principe de récurrence se généralise alors de la manière suivante :

Théorème (principe d'induction) *Soit $(E, \preccurlyeq)$ un ensemble bien ordonné, A une partie de E, et $\varphi : E \setminus A \to E$ une application vérifiant : $\forall x \in E \setminus A$, $\varphi(x) \prec x$. On considère un prédicat P défini sur E, vérifiant :*

- *pour tout $a \in A$, $P(a)$ est vrai ;*
- *pour tout $x \in E \setminus A$, $P\big(\varphi(x)\big) \Rightarrow P(x)$.*

Alors $P(x)$ est vrai pour tout élément x de E.

Preuve Soit X l'ensemble des éléments x de E tel que $P(x)$ est faux, et supposons X non vide. X possède donc un plus petit élément x_0. $P(x_0)$ est faux, donc $x_0 \notin A$. Mais alors $\varphi(x_0) \prec x_0$, donc $\varphi(x_0) \notin X$. $P\big(\varphi(x_0)\big)$ est donc vrai, ce qui implique que $P(x_0)$ l'est aussi, *i.e.* $x_0 \notin X$. On aboutit bien à une contradiction.

• Fonctions inductives

Le principe d'induction permet de justifier qu'on définit sans ambiguïté une fonction $f : E \to F$ en procédant de la manière suivante : si $g : A \to F$ et $h : E \setminus A \times F \to F$ sont deux fonctions quelconques, on pose :

$$\forall a \in A, \quad f(a) = g(a)$$
$$\forall x \in E \setminus A, \quad f(x) = h\big(x, f \circ \varphi(x)\big)$$

Une fonction ainsi définie est dite *inductive*.

Exemple. La fonction factorielle est définie inductivement par les relations :

$$0! = 1 \quad \text{et} \quad \forall n \geqslant 1,\ n! = n \times (n-1)!.$$

Nous avons ici $E = \mathbb{N}$, $A = \{0\}$, $\varphi : n \mapsto n - 1$, $g : x \mapsto 1$ et $h : (x, y) \mapsto xy$.

Exemple. La fonction pgcd est définie inductivement par les relations :

$$\forall n \in \mathbb{N}, \mathrm{pgcd}(n, 0) = n \quad \text{et} \quad \forall (n, p) \in \mathbb{N}^2,\ 1 \leqslant p \leqslant n \Rightarrow \mathrm{pgcd}(n, p) = \mathrm{pgcd}(p, n \bmod p).$$

Nous avons ici $E = \big\{(n, p) \in \mathbb{N}^2 \,/\, p \leqslant n\big\}$, $A = \big\{(n, 0) \,/\, n \in \mathbb{N}\big\}$, $\varphi : (n, p) \mapsto (p, n \bmod p)$, $g : (x, 0) \mapsto x$ et $h : (x, y) \mapsto y$. C'est la preuve de la validité de l'algorithme d'Euclide :

```
let rec pgcd n = function
   0 -> n
 | _ -> pgcd p (n mod p) ;;
```

Remarque. Si $n < p$, le calcul de `pgcd n p` débute par l'appel récursif à `pgcd p n`, avec $(p, n) \in E$, donc cet algorithme est valable pour $(n, p) \in \mathbb{N}^2$.

Toute fonction définie inductivement va aisément se calculer à l'aide d'un algorithme suivant peu ou prou le schéma suivant :

```
let rec f = function
   x when dans_A x -> g x
 | x               -> h x (f (phi x)) ;;
```

Il suffit pour cela d'avoir défini les fonctions :

```
dans_A : 'a -> bool      phi : 'a -> 'a      g : 'a -> 'b      h : 'a -> 'b -> 'b
```

Au prix d'une généralisation aisée, les appels récursifs multiples rentrent aussi dans ce cadre. Considérons par exemple la suite de Fibonacci définie par :

$$u_0 = u_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ u_n = u_{n-1} + u_{n-2}.$$

Posons $E = \mathbb{N}$, $A = \{0, 1\}$, $\varphi_1 : x \mapsto x - 1$ et $\varphi_2 : x \mapsto x - 2$. Alors la suite $(u_n)_{n \in \mathbb{N}}$ est définie par :

$$\forall n \in A,\ u_n = 1 \quad \text{et} \quad \forall n \in E \setminus A,\ u_n = u_{\varphi_1(n)} + u_{\varphi_2(n)}.$$

• La fonction d'Ackermann

Il s'agit d'une application $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ qui ne rentre pas dans le cadre des fonctions inductives énoncé plus haut. elle est définie par les relations :

$$\forall (n, p) \in \mathbb{N}^2, \qquad \begin{aligned} &A(0, p) = p + 1 \\ &A(n, 0) = A(n-1, 1) \text{ si } n \geqslant 1 \\ &A(n, p) = A\big(n-1, A(n, p-1)\big) \text{ si } n \geqslant 1 \text{ et } p \geqslant 1 \end{aligned}$$

Munissons $\mathbb{N}^2$ de l'ordre lexicographique, et prouvons par induction que pour tout couple $(n, p) \in \mathbb{N}^2$, $A(n, p)$ est défini sans ambiguïté.

Preuve

– $A(0, 0) = 0$ est bien défini sans ambiguïté.

– Si $(n, p) \neq (0, 0)$ supposons le résultat acquis pour tout couple $(n', p') \prec (n, p)$.

Si $n = 0$, alors $A(0, p) = p + 1$ est bien défini.

Si $n > 0$ et $p = 0$, alors $A(n, 0) = A(n-1, 1)$ et $(n-1, 1) \prec (n, 0)$ donc par hypothèse de récurrence $A(n-1, 1)$ est défini sans ambiguïté.

Si $n > 0$ et $p > 0$, alors $(n, p-1) \prec (n, p)$ donc $A(n, p-1)$ est bien défini, et $\big(n-1, A(n, p-1)\big) \prec (n, p)$ donc $A\big(n-1, A(n, p-1)\big)$ est aussi défini, ce qui achève la démonstration.

Sa définition Caml ne pose donc plus de problème :

```
let rec Ack = fun
    0 _ -> p + 1
  | _ 0 -> Ack (n-1) 1
  | n p -> Ack (n-1) (Ack n (p-1)) ;;
```

Intéressons nous maintenant au coût de cette fonction. On démontre que :

$$A(0,p) = p+1, \qquad A(1,p) = p+2, \qquad A(2,p) = 2p+3, \qquad A(3,p) = 2^{p+3}-3 \quad \text{et} \quad A(4,p) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{p+3 \text{ chiffres}} - 3$$

ce qui laisse imaginer le coût phénoménal de cette fonction ! À titre indicatif, le calcul de $A(4,1)$ nécessite 2 862 986 336 appels récursifs (soit plus de deux milliards !). Quant au calcul de $A(4,2)$, le nombre d'appels récursifs est de l'ordre de $10^{39\,446}$...

2. Coût d'un algorithme récursif

L'exemple précédent a montré qu'il est impératif de prendre garde au coût d'un algorithme récursif. Étudions un second exemple, d'abord plus simple, en considérant de nouveau la suite de Fibonacci définie par la donnée de $u_0 = u_1 = 1$ et la relation : $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$. On peut la définir aisément à l'aide d'un algorithme récursif :

```
let rec fib = function
   0 | 1 -> 1
 | n     -> fib (n-1) + fib (n-2) ;;
```

mais nous allons voir que son coût est très important : notons t_n le nombre d'additions nécessaire pour calculer u_n. Alors $t_0 = t_1 = 0$ et pour tout $n \in \mathbb{N}$, $t_{n+2} = 1+t_n+t_{n-1}$. On a donc : $\forall n \in \mathbb{N}$, $t_n = u_n - 1$. Or $u_n \underset{+\infty}{\sim} \dfrac{1}{\sqrt{5}}\Big(\dfrac{1}{+}\sqrt{5}2\Big)^{n+1}$, donc le coût est *exponentiel*, bien loin donc d'un coût linéaire qu'on serait en droit d'espérer.
De même, le nombre d'appels s_n à la fonction `fib` pour calculer u_n vérifie : $s_0 = s_1 = 1$ et $\forall n \in \mathbb{N}$, $s_{n+2} = 1+s_{n+1}+s_n$, ce qui donne $s_n = 2u_n - 1$, ce qui là encore est bien supérieur aux $n+1$ appels qu'il parait raisonnable d'espérer.
Pour améliorer cet algorithme, nous allons revenir sur la notion d'itérateur associé à la suite de Fibonacci, défini par les relations :

$$(u_0, u_1) = (1,1) \quad \text{et} \quad \forall n \in \mathbb{N},\ (u_{n+1}, u_{n+2}) = (u_{n+1}, u_n + u_{n+1}) = \varphi(u_n, u_{n+1}) \quad \text{avec} \quad \varphi : (x,y) \mapsto (y, x+y).$$

Ainsi, on peut écrire :

```
let fib =
   let rec fib_aux (p,q) = function
      0 -> p
    | n -> fib_aux (q,p+q) (n-1)
   in fib_aux (1,1) ;;
```

et cet algorithme a maintenant un coût linéaire[1].

3. Récursivité terminale

Nous avons vu que tout programme itératif possède une version récursive. Nous allons maintenant nous intéresser, dans un cas simple, à la réciproque de ce résultat (c'est le problème de la *dérécursification*).
Considérons de nouveau un ensemble bien ordonné $(E, \preccurlyeq)$, A une partie de E et $\varphi : E \setminus A \to E$ tel que : $\forall x \in E \setminus A$, $\varphi(x) \prec x$. Si $g : A \to F$ est une fonction quelconque, on définit une unique fonction $f : E \to F$ en posant :

$$\forall a \in A, \quad f(a) = g(a)$$
$$\forall x \in E \setminus A, \quad f(x) = f\big(\varphi(x)\big)$$

Il s'agit d'un cas particulier de fonction inductive. Une telle définition est qualifiée de *récursive terminale* car l'appel récursif est la dernière opération que l'on effectue pour calculer $f(x)$ lorsque $x \in E \setminus A$.

[1]Nous verrons plus tard qu'il existe même un algorithme calculant u_n avec un coût logarithmique.

Exemple. La définition suivante du pgcd est récursive terminale :

$$\forall n \in \mathbb{N}, \quad \mathrm{pgcd}(n, 0) = n$$
$$\forall (n, p) \in \mathbb{N} \times \mathbb{N}^*, \quad \mathrm{pgcd}(n, p) = \mathrm{pgcd}(p, n \bmod p)$$

Une telle fonction se calcule aussi bien par un algorithme récursif que par un algorithme itératif :

– version récursive :

```
let rec f = function
   x when dans_A x -> g x
 | x               -> f (phi x) ;;
```

– version itérative :

```
let f x =
   let y = ref x in
   while not (dans_A !y) do y := phi !y done ;
   g !y ;;
```

Ceci prouve en particulier que tout algorithme récursif terminal possède une version itérative. En outre, en suivant le modèle ci-dessus, on peut traduire itérativement un algorithme récursif terminal, comme par exemple l'algorithme d'Euclide :

```
let pgcd n p =
   let x = ref n and y = ref p in
   while !y > 0 do let aux = !y in y := !x mod !y ; x := aux done ;
   !x ;;
```

Revenons maintenant sur la notion d'itération ; dans le module correspondant j'avais écrit deux versions de la fonction `itère` :

```
let rec itère f u = function
   0 -> u
 | n -> f (itère f u (n-1)) ;;
```

et

```
let rec itère f u = function
   0 -> u
 | n -> itère f (f u) (n-1) ;;
```

Nous pouvons maintenant percevoir la différence qui existe entre ces deux versions, puisque la seconde est récursive terminale, au contraire de la première. De cette manière, nous disposons d'une méthode pour rendre terminal un algorithme récursif qui ne l'est pas : il suffit de chercher l'itérateur correspondant, et de suivre alors le modèle ci-dessus. Effectuons par exemple ce travail avec la fonction factorielle. La définition ci-dessous n'est pas terminale :

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

Nous avons déjà vu que l'itérateur associé est la suite $u_n = (n, n!)$ définie par la donnée de $u_0 = (0, 1)$ et la relation : $u_{n+1} = f(u_n)$ avec $f : (x, y) \mapsto (x + 1, (x + 1)y)$. En suivant le modèle de la fonction `itère` donné ci-dessus, on obtient :

```
let fact n =
  let rec aux (x,y) = function
      0 -> y
    | n -> aux (x+1, (x+1)*y) (n-1) in
  aux (0,1) n ;;
```

qui est maintenant récursive terminale.

On pourra de même écrire des versions récursives terminales des fonctions calculant a^n ainsi que la suite de Fibonacci.

Remarque. La fonction `itère` a pour objet de calculer $u_n = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ fois}}(u_0)$; dans sa première version (non terminale), u_n est calculé par l'intermédiaire de la relation :

$$f^{\circ(n)} = f \circ f^{\circ(n-1)} = f \circ (f \circ (\cdots \circ (f \circ f)) \cdots)$$

alors que dans la seconde (terminale), u_n est calculé par :

$$f^{\circ(n)} = f^{\circ(n-1)} \circ f = (\cdots((f \circ f) \circ f) \circ \cdots \circ f) \circ f.$$

Remarque. En réalité, la différence fondamentale entre un algorithme récursif terminal est un algorithme non terminal n'a pas été évoquée ici : le coût spatial n'est pas le même. Cette différence sera étudiée dans un module ultérieur consacré à la notion de *pile*.

4. Exercices

1 – Déterminer ce que calcule la fonction suivante (et le démontrer) :

```
let rec maccarty n =
   if n > 100 then n-10 else maccarty (maccarty (n+11)) ;;
```

On notera qu'il n'est pas évident qu'une telle fonction donne un résultat.

2 – On dispose d'un stock illimité de pièces et de billets de c_1, c_2, ..., c_p euros, et on souhaite compter le nombre de manières possibles d'obtenir n euros avec ces pièces. L'objectif de cet exercice est d'écrire une fonction Caml `compte` (de type `int -> int list -> int`) effectuant ce calcul.

Par exemple, pour connaitre le nombre de décompositions de 50 € à l'aide de pièces et de billets de 1 €, 2 €, 5 €, 10 € et 20 €, on écrira :

```
# compte 50 [1;2;5;10;20] ;;
- : int = 450
```

On dispose donc de 450 manières différentes de le faire.

Commencez par répondre aux deux questions suivantes : combien y-a-t-il de décompositions de n n'utilisant pas la pièce c_1 ? et au moins une fois ?

En déduire alors un algorithme récursif répondant au problème.

3 – On considère la fonction suivante, calculant le coefficient binomial $\binom{n}{p}$ lorsque $0 \leqslant p \leqslant n$:

```
let rec binôme n p =
   if p = 0 or p = n then 1
                     else binôme (n-1) p + binôme (n-1) (p-1) ;;
```

Démontrer que cette fonction calcule effectivement $\binom{n}{p}$, puis évaluer son coût, en calculant le nombre d'appels à `binôme` effectués lors du calcul de `binôme n p` en fonction de n et p.

Proposer de meilleures solutions.

4 – Prouver que l'algorithme suivant calculant le produit de deux entiers est correct :

```
let rec produit x y = match x with
    0 -> 0
  | _ -> produit (x/2) (2y) + y*(x mod 2) ;;
```

Évaluer son coût.

Donner une version récursive terminale de cet algorithme.

5 – On considère deux points distincts P et Q du plan. La courbe du dragon d'ordre 0 entre P et Q est le segment $[P, Q]$. Si $n \geqslant 1$, la courbe du dragon d'ordre n entre P et Q est obtenue en déterminant le point R tel que PRQ soit un triangle isocèle direct rectangle en R, et en traçant les courbes du dragon d'ordre $(n-1)$ entre P et R et entre Q et R.

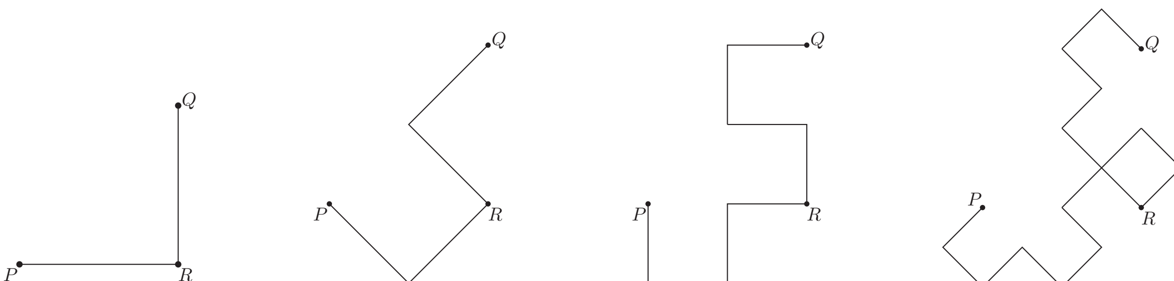

Fig. 1 – Les courbes du dragon d'ordres 1, 2, 3, 4

Définir une procédure récursive `dragon n p q` traçant la courbe du dragon d'ordre n entre P et Q (les instructions nécessaires pour le graphisme avec Caml sont données en annexe).

6 – Le défaut de la fonction écrite dans l'exercice précédent est de provoquer un tracé fort discontinu : les segments ne sont pas tracés dans un ordre naturel (voir l'illustration figure 2). Réécrire cet algorithme en s'imposant en plus de ne jamais lever le crayon lors du tracé, c'est à dire en n'utilisant plus la fonction `moveto`. Pour ce faire, on définira

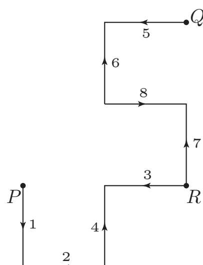

Fig. 2 – Le sens et l'ordre dans lesquels sont tracés les segments pour $n = 3$

la courbe du nogard d'ordre n entre P et Q comme étant la courbe du dragon d'ordre n entre Q et P, et on définira deux fonctions mutuellement récursives `dragon` et `nogard`.

• Procédures graphiques avec Caml

Les primitives graphiques ne sont pas accessibles directement ; il faut au préalable « ouvrir » le module correspondant de la bibliothèque du système par l'instruction : `#open "graphics";;` (attention, le `#` n'est pas le prompt du système interactif ; il fait partie de l'instruction).

On ouvre ensuite la fenêtre graphique à l'aide de l'instruction `open_graph "";;`

Les points de cette fenêtre sont repérés par des coordonnées entières (x, y) où $0 \leqslant x \leqslant x_{max}$ et $0 \leqslant y \leqslant y_{max}$, le point de coordonnées $(0, 0)$ se trouvant en bas à gauche de l'écran. Les primitives `size_x()` et `size_y()` donnent respectivement les valeurs de $x_{max} + 1$ et $y_{max} + 1$ (ces valeurs dépendent de votre ordinateur).

À l'ouverture de la fenêtre graphique, le point courant se trouve en $(0, 0)$. la primitive `moveto x y` lève le crayon et le déplace au point de coordonnées (x, y) ; la primitive `lineto x y` déplace le crayon au point de coordonnées (x, y) en traçant un segment de droite.

`clear_graph ()` permet d'effacer le contenu de la fenêtre graphique.

■