Récursivité

1 Introduction

Nous avons déjà écrit de nombreuses fonctions *récursives*, c'est-à-dire des fonctions intervenant dans leur propre définition, comme par exemple la version suivante de la fonction factorielle :

```
let rec fact n =
  if n = 0
  then 1
  else n * fact (n-1)
;;
```

```
val fact : int -> int = <fun>
```

Se pose alors la question de la *terminaison* : pour une valeur donnée de `n`, l'appel `fact n` termine-t-il ?

Dans le cas de cette fonction, nous pouvons appliquer le même raisonnement que celui souvent utilisé pour la terminaison d'une boucle conditionnelle : pour un entier naturel non nul `n`, la suite des arguments des appels récursifs à la fonction `fact` est une suite strictement décroissante d'entiers naturels, donc ne peut pas être infinie ; par conséquent, `fact n` termine pour tout entier naturel `n`.

On voudrait aussi s'assurer que la fonction renvoie bien le résultat attendu : c'est le problème de la *correction*. Dans le cas de la fonction `fact`, on peut raisonner par récurrence : si n vaut 0, la fonction renvoie bien $n!$, et pour tout entier naturel n, si `fact` n renvoie bien $n!$ alors `fact` $(n+1)$ renvoie $(n+1)!$.

L'objectif des deux premières parties de ce chapitre est de généraliser ce raisonnement à des cas où l'on ne peut pas (ou en tout cas pas facilement) raisonner sur des entiers naturels.

Il faut néanmoins garder en tête qu'il n'existe pas d'algorithme général permettant de déterminer à l'avance si l'appel à une fonction termine. Dans le cas contraire, on pourrait en effet disposer d'une fonction `termine : int -> int -> int -> bool` qui prendrait en argument une fonction `f : int -> int` et un entier `n` et qui renverrait `true` si l'appel `f n` termine, `false` sinon. En définissant la fonction :

```
let rec absurde n =
  if termine absurde n
  then absurde n
  else 0
```

on obtient alors une fonction qui termine si et seulement si elle ne termine pas.

2 Ordres bien fondés

2.1 Rappels sur les relations d'ordre

Définition

— *Une relation binaire $\preccurlyeq$ sur un ensemble E est appelée une* relation d'ordre *sur E lorsqu'elle est :*
  — *réflexive : $\forall x \in E, x \preccurlyeq x$;*
  — *antisymétrique : $\forall (x,y) \in E^2, (x \preccurlyeq y \text{ et } y \preccurlyeq x) \Rightarrow x = y$*
  — *et transitive : $\forall (x,y,z) \in E^3, (x \preccurlyeq y \text{ et } y \preccurlyeq z) \Rightarrow x \mathcal{R} z$.*

— *Une relation d'ordre $\preccurlyeq$ est dite relation d'ordre total lorsque deux éléments sont toujours comparables :*

$$\forall (x,y) \in E^2, x \preccurlyeq y \text{ ou } y \preccurlyeq x$$

On dit alors que $(E, \preccurlyeq)$ est un ensemble totalement ordonné.

— *Dans le cas contraire, on dit que la relation d'ordre est* partielle $\preccurlyeq$ *et que* $(E, \preccurlyeq)$ *est un ensemble partiellement ordonné.*

Proposition 1 *Soient* $(E, \preccurlyeq_E)$ *et* $(F, \preccurlyeq_F)$ *deux ensembles ordonnés. Alors l'*ordre lexicographique *défini sur* $E \times F$ *par :*

$$\forall (a,b),(c,d) \in E \times F, (a,b) \preccurlyeq (c,d) \Leftrightarrow (a \prec_E c \text{ ou } (a = c \text{ et } b \preccurlyeq_F d))$$

où $\prec_E$ *désigne la relation d'ordre strict associé à* $\preccurlyeq_E$*, est une relation d'ordre sur* $E \times F$

Exemple 1
Ordre lexicographique sur $\mathbb{N}^2$.

2.2 Ordre bien fondé

Définition *Soit* E *un ensemble. Un ordre* $\preccurlyeq$ *sur* E *est dit* bien fondé *lorsqu'il n'existe pas de suite strictement décroissante d'éléments de* E *pour cet ordre. On dit alors que* $(E, \preccurlyeq)$ *est un ensemble* bien fondé.

Remarque : Lorsque l'ordre est total, on dit alors que E est *bien ordonné* (et que $\preccurlyeq$ est un *bon ordre*).

Remarque : $\preccurlyeq$ est bien fondé si et seulement si toute suite décroissante pour $\preccurlyeq$ est stationnaire.

Proposition 2 *Soit* $(E, \preccurlyeq)$ *un ensemble bien fondé et* $F \subset E$ *non vide. Alors* $(F, \preccurlyeq)$ *est bien fondé.*

Définition *Soit* $(E, \preccurlyeq)$ *un ensemble ordonné et* $A \subset E$*. On dit que :*

— $m \in A$ *est un* élément minimal *de* A *lorsque* $\forall a \in A, a \preccurlyeq m \Rightarrow a = m$.

— $m \in A$ *est un* plus petit élément *de* A *lorsque* $\forall a \in A, m \preccurlyeq a$.

Remarque :

— Si m est un plus petit élément de A, alors m est unique.

— Dans le cas d'un ordre total, si m est un élément minimal de A alors m est un plus petit élément de A.

Proposition 3 *Un ensemble ordonné* $(E, \preccurlyeq)$ *est bien fondé si et seulement si toute partie non vide de* E *admet un élément minimal.*

Démonstration

— Soit $(E, \preccurlyeq)$ un ensemble bien fondé. Soit $A \subset E$ non vide, montrons par l'absurde que A admet un élément minimal. Si A n'admet pas d'élément minimal, alors $\forall m \in A, \exists a \in A, a \prec m$. Comme A est non vide, il existe un élément $u_0 \in A$ et on peut construire une suite strictement décroissante d'éléments de A : absurde. Donc A admet un élément minimal.

— Soit $(E, \preccurlyeq)$ un ensemble ordonné tel que toute partie non vide de E admet un élément minimal. Supposons qu'il existe une suite $(u_n)_{n\in\mathbb{N}}$ strictement décroissante. Soit $A = \{u_n | n \in \mathbb{N}\}$. Alors A admet un élément minimal m et il existe n_0 tel que $m = u_{n_0}$. Or $u_{n_0+1} \in A$ et $u_{n_0+1} \prec u_{n_0}$: absurde. Donc E est bien fondé.

Exemple 2
Considérons $\mathbb{N}^2$ muni de l'ordre lexicographique :

$$(a,b) \preccurlyeq (c,d) \Leftrightarrow (a < c) \text{ ou } (a = c \text{ et } b \leqslant d)$$

Soit $A \subset \mathbb{N}^2$ non vide.
Posons $a_0 = \min\{a \in \mathbb{N} | \exists b \in \mathbb{N}, (a,b) \in A\}$ et $b_0 = \min\{b \in \mathbb{N} | (a_0, b) \in A\}$. Alors pour tout $(a,b) \in A$, $a_0 \leqslant a$ et si $a = a_0$, alors $b_0 \leqslant b$ donc $(a_0, b_0) \preccurlyeq (a,b)$.
Par conséquent, $(a,b) \preccurlyeq (a_0,b_0) \Rightarrow (a,b) = (a_0,b_0)$.
Finalement, $(\mathbb{N}^2, \preccurlyeq)$ est bien fondé (et même bien ordonné).

2.3 Principe d'induction

Définition *On appelle* prédicat *sur un ensemble E une application de E dans l'ensemble des booléens.*

Théorème 4 (Principe d'induction) *Soit $(E, \preccurlyeq)$ un ensemble bien fondé, soit $\mathcal{M}$ l'ensemble de ses éléments minimaux. Si un prédicat $\mathcal{P}$ sur E vérifie :*

- $\forall x \in \mathcal{M}$, $\mathcal{P}(x)$
- $\forall x \in E \setminus \mathcal{M}$, $(\forall y \prec x, \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$

Alors pour tout $x \in E$, $\mathcal{P}(x)$.

Démonstration
Par l'absurde, supposons qu'il existe $x_0 \in E$ tel que $\mathcal{P}(x_0)$ soit faux. Alors x_0 n'appartient pas à $\mathcal{M}$, et il existe $x_1 \in E$ tel que $x_1 \prec x_0$ et tel que $\mathcal{P}(x_1)$ est faux. De même, x_1 ne peut pas appartenir à $\mathcal{M}$. On peut donc construire une suite infinie strictement décroissante, ce qui est en contradiction avec le fait que E est bien fondé.

Remarque : Pour $E = \mathbb{N}$ muni de l'ordre usuel, on retrouve le principe de récurrence forte.

3 Terminaison

On considère une fonction récursive f.

Théorème 5 *Soit $\mathcal{A}$ l'ensemble des arguments de f. Soit φ une application de $\mathcal{A}$ vers un ensemble bien fondé E. Soit $\mathcal{M}_\mathcal{A}$ l'ensemble des arguments x tels que $\varphi(x)$ est un élément minimal de E. Si :*

- *la fonction f termine pour tout $x \in \mathcal{M}_\mathcal{A}$;*
- *pour tout $x \in \mathcal{A} \setminus \mathcal{M}_\mathcal{A}$, le calcul de $f(x)$ ne nécessite qu'un nombre fini (éventuellement nul) d'appels à f, sur des arguments y tels que $\varphi(y) \prec \varphi(x)$, et la terminaison de ces appels entraîne celle de $f(x)$*

alors la fonction f termine sur tout argument de $\mathcal{A}$.

Démonstration
On utilise le principe d'induction pour la propriété suivante sur $z \in E$: $\mathcal{P}(z)$ « les appels $f(x)$ avec $\varphi(x) = z$ terminent. »

Définition *Les éléments x de $\mathcal{A}$ pour lesquels le calcul de $f(x)$ ne nécessite aucun appel à f sont appelés* cas de base *ou* cas terminaux.

Remarque : L'ensemble $\mathcal{A}$ sera souvent lui-même un ensemble bien fondé, et la fonction φ utilisée sera alors la fonction identité.
Dans d'autres cas, l'application φ est souvent une application à valeurs dans $\mathbb{N}$; par exemple, à une liste, on peut associer sa longueur.

Exemple 3
Considérons la fonction :

```
let rec length l =
  match l with
  | [] -> 0
  | t::q -> 1 + length q
;;
```

A une liste, on peut associer sa longueur :

- La fonction `length` termine pour la liste de longueur 0 ;
- Pour une liste de longueur au moins 1, un seul appel récursif est réalisé, et il porte sur une liste de longueur strictement inférieure.

Par conséquent, la fonction termine.

Exemple 4
La fonction d'Ackermann suivante termine :

```
let rec ack n p =
  match (n, p) with
  | 0, _ -> p+1
  | _, 0 -> ack (n-1) 1
  | _, _ -> ack (n-1) (ack n (p-1))
;;
```

```
val ack : int -> int -> int = <fun>
```

En effet, on considère ici $\mathbb{N}^2$ muni de l'ordre lexicographique.

— Les cas de bases sont les couples dont la première coordonnée est nulle.
— Pour $(n, 0)$ avec $n \neq 0$, le seul appel récursif porte sur un couple strictement plus petit.
— Pour (n, p) avec n et p tous les deux non nuls, il y a deux appels récursifs, dont les arguments sont $(n, p-1)$ et $(n-1, \texttt{ack}(n, p-1))$ qui sont bien strictement plus petits que (n, p).

Exercice

Considérons les deux fonctions suivantes calculant $\binom{n}{p}$, rencontrées dans le chapitre 1 :

```
let rec binome1 n p =
  match n, p with
    | n, p when p < 0 || p > n -> 0
    | 0, 0 -> 1
    | 0, _ -> 0
    | _ -> binome1 (n-1) (p-1) + binome1 (n-1) p
;;
```

```
let rec binome2 n p =
  match n, p with
    | n, p when p < 0 || p > n -> 0
    | _, 0 -> 1
    | 0, _ -> 0
    | _ -> n* binome2 (n-1) (p-1)/p
 ;;
```

Montrer que ces fonctions terminent.

Remarque : Il n'est pas forcément facile de trouver un ensemble bien fondé et une application φ associée pour une fonction récursive donnée. Par exemple, la terminaison de la fonction suivante est un problème ouvert :

```
let rec syracuse n =
  match n with
  | 1 -> 1
  | _ -> if n mod 2 = 0
         then syracuse (n/2)
         else syracuse (3*n+1)
;;
```

4 Correction

Pour démontrer qu'une fonction récursive calcule ce qu'elle doit calculer, on peut raisonner comme pour la terminaison, en adaptant le prédicat à démontrer.

Théorème 6 *Soit $\mathcal{A}$ l'ensemble des arguments de f. Soit φ une application de $\mathcal{A}$ vers un ensemble bien fondé E. Soit $\mathcal{M}_\mathcal{A}$ l'ensemble des arguments x tels que $\varphi(x)$ est un élément minimal de E. Pour $z \in E$, on considère le prédicat $\mathcal{P}(z)$: « Pour x tel que $\varphi(x) = z$, $f(x)$ a la bonne valeur. ». Si :*

- *pour tout $x \in \mathcal{M}_\mathcal{A}$, $\mathcal{P}(\varphi(x))$;*
- *pour tout $x \in \mathcal{A} \setminus \mathcal{M}_\mathcal{A}$, le calcul de $f(x)$ ne nécessite qu'un nombre fini (éventuellement nul) d'appels à f, sur des arguments y_1,..., y_N tels que pour tout $k \in [\![1, N]\!]$, $\varphi(y_k) \prec \varphi(x)$, et que $(\forall k \in [\![1, N]\!], \mathcal{P}(\varphi(y_k))) \Rightarrow \mathcal{P}(\varphi(x))$*

alors pour tout $x \in \mathcal{A}$, $\mathcal{P}(\varphi(x)$, donc la valeur de $f(x)$ est correcte.

Démonstration
C'est le principe d'induction.

5 Pratique de la récursivité

5.1 La pile d'appels

La pile d'exécution (ou pile d'appels) est une structure de données qui sert à enregistrer des informations au sujet des fonctions actives dans un programme, c'est-à-dire des fonctions dont l'exécution n'est pas encore terminée.

Cette pile d'appels permet de connaître l'adresse de retour d'une fonction, ainsi que d'autres valeurs telles que les variables locales de la fonction, ses paramètres, etc.

En particulier, lors qu'une fonction f appelle une fonction g, ce qui est relatif à l'appel de la fonction g sera placé juste au-dessus de ce qui est relatif à la fonction f dans la pile d'appels. Lorsque l'appel à g termine, ce qui est relatif à g est *dépilé* ; comme l'adresse de retour est contenu dans la pile d'appels, l'exécution de f peut reprendre.

Une fonction récursive étant une fonction qui s'appelle elle-même, ses appels s'empilent dans la pile d'appels ; une fois arrivé à un cas terminal, le nombre d'éléments de la pile se réduit. Enfin, la récursion s'arrête lorsque la pile est vide, c'est-à-dire lorsqu'on dépile l'élément correspondant au premier appel de la fonction.

En OCaml, on peut vérifier ce comportement en *traçant* les appels de la fonction :

```
#trace fact;;
```

```
fact is now traced.
```

```
fact 5;;
```

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

```
- : int = 120
```

Remarque : On peut cesser de tracer la fonction 'fact' de la manière suivante :

```
#untrace fact;;
```

Le nombre d'appels imbriqués par une fonction récursive peut être important. La pile d'appels a une capacité limitée ; si la pile est pleine, un appel supplémentaire produit un dépassement de capacité :

```
let rec somme n =
  if n = 0
  then 0
  else n + somme (n-1);;
```

```
fact is no longer traced.
```

```
val somme : int -> int = <fun>
```

```
somme 1000000;;
```

```
Stack overflow during evaluation (looping recursion?).
```

Remarque : Pour éviter le dépassement capacité précédent, on peut écrire une nouvelle version de la fonction 'somme' qui utilise un *accumulateur* :

```
let rec somme_avec_acc n acc =
  match n with
  | 0 -> acc
  | _ -> somme_avec_acc (n - 1) (n + acc)
;;
```

```
val somme_avec_acc : int -> int -> int = <fun>
```

On utilise ici un deuxième argument pour effectuer les additions. Il suffit de prendre pour ce deuxième paramètre la valeur 0 pour calculer `somme n` :

```
let somme n =
  somme_avec_acc n 0
;;
```

```
val somme : int -> int = <fun>
```

```
somme 1000000 ;;
```

```
- : int = 500000500000
```

On remarque ici qu'il n'y a pas de dépassement de la pile d'appels. Cette fonction est en effet dite *récursive terminale*, car l'appel récursif est le dernier calcul réalisé par la fonction f.

L'intérêt d'avoir une fonction récursive terminale est qu'un appel récursif à f ne nécessite pas d'empilement sur la pile d'appels : l'appel récursif peut prendre la place de l'appel en cours dans la pile, puisqu'il n'y a pas d'opération à effectuer une fois l'appel récursif terminé. Certains compilateurs, comme celui-ci d'OCaml, sont capables de détecter les fonctions récursives terminales et limitent alors les empilements dans la pile d'appels.

Lorsqu'on utilise un accumulateur, il n'est pas forcément agréable de devoir faire les appels à la fonction avec un paramètre supplémentaire. On définit souvent la fonction avec accumulateur comme une fonction locale à une autre fonction, qui se contentera de l'appeler avec la bonne valeur initiale pour l'accumulateur. On en profitera généralement pour lui donner un nom plus court :

```
let somme n =
  let rec aux n acc =
    match n with
    | 0 -> 0
    | _ -> aux (n - 1) (n + acc)
  in aux n 0
;;
```

```
val somme : int -> int = <fun>
```

Exercices divers

Exercice 1

Écrire une fonction récursive calculant le pgcd de deux entiers naturels non tous deux nuls par l'algorithme d'Euclide. Justifier qu'elle termine.

Exercice 2

On dispose d'un stock illimité de billets et de pièces de valeurs c_1, ..., c_p. On souhaite dénombrer le nombre de manières possibles d'obtenir une somme donnée avec ces espèces. Écrire une fonction récursive prenant en paramètre la somme à atteindre et la liste des valeurs de pièces et calculant cette quantité.

Il y a par exemple 11 manières possibles de payer 10€ avec des pièces et billets de 1, 2, 5 et 10€ (ce qui est facile à calculer à la main), et 451 manières possibles de payer 50€ avec des pièces et billets de 1, 2, 5, 10, 20 et 50€.