Récursivité

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Principe de récurrence simple

Soit $\mathscr{P}$ un prédicat défini sur $\mathbb{N}$, tel que $\mathscr{P}(0)$ est vrai, ainsi que l'implication $\mathscr{P}(n-1) \Longrightarrow \mathscr{P}(n)$. Alors pour tout $n \in \mathbb{N}$, $\mathscr{P}(n)$ est vrai.

Conséquence. La suite $(u_n)_{n\in\mathbb{N}}$ définie par les relations $u_0 = a$ et $u_n = f(u_{n-1})$ est calculable ; la fonction récursive suivante se termine.

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

Principe de récurrence simple

Soit $\mathscr{P}$ un prédicat défini sur $\mathbb{N}$, tel que $\mathscr{P}(0)$ est vrai, ainsi que l'implication $\mathscr{P}(n-1) \Longrightarrow \mathscr{P}(n)$. Alors pour tout $n \in \mathbb{N}$, $\mathscr{P}(n)$ est vrai.

Conséquence. La suite $(u_n)_{n\in\mathbb{N}}$ définie par les relations $u_0 = a$ et $u_n = f(u_{n-1})$ est calculable ; la fonction récursive suivante se termine.

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

Il est aisé de donner des exemples de fonctions qui ne se terminent pas, telle la fonction de Morris :

```
let rec m = function
  | 0 q -> 1
  | p q -> m (p-1) (m p q) ;;
```

En effet, $m(1,0) = m\big(0, m(1,0)\big)$ et le passage d'argument se faisant par valeur, le calcul ne se termine pas.

Principe de récurrence simple

Soit $\mathscr{P}$ un prédicat défini sur $\mathbb{N}$, tel que $\mathscr{P}(0)$ est vrai, ainsi que l'implication $\mathscr{P}(n-1) \Longrightarrow \mathscr{P}(n)$. Alors pour tout $n \in \mathbb{N}$, $\mathscr{P}(n)$ est vrai.

Conséquence. La suite $(u_n)_{n\in\mathbb{N}}$ définie par les relations $u_0 = a$ et $u_n = f(u_{n-1})$ est calculable ; la fonction récursive suivante se termine.

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

Pour des fonctions récursives plus complexes, la preuve de la terminaison peut rester un problème ouvert ; c'est le cas de la fonction Q de HOFSTADTER :

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

Problème de l'arrêt

Existe-t-il un moyen algorithmique de déterminer la terminaison d'une fonction ?

Problème de l'arrêt

Existe-t-il un moyen algorithmique de déterminer la terminaison d'une fonction ?

Considérons l'ensemble $\mathscr{F}$ des fonctions de type *int -> int*. $\mathscr{F}$ est en bijection avec un sous-ensemble de l'ensemble des suites finies sur $\{0,1\}$ donc est dénombrable : il existe une bijection $\varphi : \mathbb{N} \to \mathscr{F}$.

Supposons l'existence d'une fonction **`termine`** de type *int -> int -> bool* qui fonctionne ainsi :

$$\texttt{termine p q} = \begin{cases} \texttt{true} & \text{si le calcul de } \varphi(p)(q) \text{ se termine} \\ \texttt{false} & \text{sinon} \end{cases}$$

Problème de l'arrêt

Existe-t-il un moyen algorithmique de déterminer la terminaison d'une fonction ?

Considérons l'ensemble $\mathscr{F}$ des fonctions de type *int -> int*. $\mathscr{F}$ est en bijection avec un sous-ensemble de l'ensemble des suites finies sur $\{0,1\}$ donc est dénombrable : il existe une bijection $\varphi : \mathbb{N} \to \mathscr{F}$.
Supposons l'existence d'une fonction **`termine`** de type *int -> int -> bool* qui fonctionne ainsi :

$$\texttt{termine p q} = \begin{cases} \texttt{true} & \text{si le calcul de } \varphi(p)(q) \text{ se termine} \\ \texttt{false} & \text{sinon} \end{cases}$$

On définit alors la fonction :

```
let rec f = function
  | p when termine p p -> f p
  | p                  -> 0 ;;
```

Notons $r \in \mathbb{N}$ tel que $f = \varphi(r)$. Dans les deux cas, la considération de la valeur de **`termine r r`** conduit à une absurdité (principe de la diagonale de Cantor).

Ensembles bien fondés

Un ensemble ordonné $(E, \preccurlyeq)$ est bien fondé lorsque toute partie non vide possède un élément minimal, et bien ordonné lorsque toute partie non vide possède un plus petit élément.

Deux exemples sur $\mathbb{N}^2$:

- l'ordre produit $(a,b) \preccurlyeq (a',b') \iff a \leqslant a'$ et $b \leqslant b'$ est bien fondé ;
- l'ordre lexicographique $(a,b) \preccurlyeq (a',b') \iff a < a'$ ou $(a = a'$ et $b \leqslant b')$ est bien ordonné.

Ensembles bien fondés

Un ensemble ordonné $(E, \preccurlyeq)$ est bien fondé lorsque toute partie non vide possède un élément minimal, et bien ordonné lorsque toute partie non vide possède un plus petit élément.

Deux exemples sur $\mathbb{N}^2$:

- l'ordre produit $(a, b) \preccurlyeq (a', b') \iff a \leqslant a'$ et $b \leqslant b'$ est bien fondé ;
- l'ordre lexicographique
  $(a, b) \preccurlyeq (a', b') \iff a < a'$ ou $(a = a'$ et $b \leqslant b')$ est bien ordonné.

Preuve : si $A \subset \mathbb{N}^2$ est non vide, on définit :

$$a_0 = \min\Big\{a \in \mathbb{N} \;\Big|\; \exists b \in \mathbb{N} \text{ tq } (a, b) \in A\Big\} \quad \text{et} \quad b_0 = \min\Big\{b \in \mathbb{N} \;\Big|\; (a_0, b) \in A\Big\}$$

(a_0, b_0) est minimal pour l'ordre produit, et est le plus petit élément de A pour l'ordre lexicographique.

Ensembles bien fondés

Un ensemble ordonné $(E,\preccurlyeq)$ est bien fondé lorsque toute partie non vide possède un élément minimal, et bien ordonné lorsque toute partie non vide possède un plus petit élément.

Soit $(E,\preccurlyeq)$ un ensemble bien fondé, $A \subset E$ non vide, et $\varphi : E \setminus A \to E$ vérifiant : $\forall x \in E \setminus A, \varphi(x) \prec x$. On considère un prédicat $\mathscr{P}$ vérifiant :

- pour tout $a \in A$, $\mathscr{P}(a)$ est vrai ;
- pour tout $x \in E \setminus A$, $\mathscr{P}\big(\varphi(x)\big) \Longrightarrow \mathscr{P}(x)$.

Alors $\mathscr{P}(x)$ est vrai pour tout élément x de E.

Ensembles bien fondés

Un ensemble ordonné $(E, \preccurlyeq)$ est bien fondé lorsque toute partie non vide possède un élément minimal, et bien ordonné lorsque toute partie non vide possède un plus petit élément.

Soit $(E, \preccurlyeq)$ un ensemble bien fondé, $A \subset E$ non vide, et $\varphi : E \setminus A \to E$ vérifiant : $\forall x \in E \setminus A,\ \varphi(x) \prec x$. On considère un prédicat $\mathscr{P}$ vérifiant :

- pour tout $a \in A$, $\mathscr{P}(a)$ est vrai ;
- pour tout $x \in E \setminus A$, $\mathscr{P}\big(\varphi(x)\big) \Longrightarrow \mathscr{P}(x)$.

Alors $\mathscr{P}(x)$ est vrai pour tout élément x de E.

Soit $X = \Big\{x \in E \;\Big|\; \mathscr{P}(x) \text{ est faux}\Big\}$, et supposons $X \neq \emptyset$: alors X possède un élément minimal x_0.

$\mathscr{P}(x_0)$ est faux, donc $x_0 \in E \setminus A$ et donc $\varphi(x_0) \prec x_0$.

x_0 est minimal dans X, donc $\varphi(x_0) \notin X$, et $\mathscr{P}\big(\varphi(x_0)\big)$ est vrai.

Ceci implique que $\mathscr{P}(x_0)$ est vrai, donc que $x_0 \notin X$. Contradiction !

Fonctions inductives

Le principe d'induction permet de justifier la terminaison d'une fonction $f : E \to F$ définie par :

$$\begin{aligned} \forall a \in A, &\quad f(a) = g(a) \\ \forall x \in E \setminus A, &\quad f(x) = h\big(x, f \circ \varphi(x)\big) \end{aligned}$$

où $g : A \to F$ et $h : E \setminus A \times F \to F$ sont deux fonctions quelconques et $\varphi : E \setminus A \to E$ vérifie $\varphi(x) \prec x$.

Fonctions inductives

Le principe d'induction permet de justifier la terminaison d'une fonction $f : E \longrightarrow F$ définie par :

$$\begin{aligned} \forall a \in A, \quad & f(a) = g(a) \\ \forall x \in E \setminus A, \quad & f(x) = h\big(x, f \circ \varphi(x)\big) \end{aligned}$$

où $g : A \longrightarrow F$ et $h : E \setminus A \times F \longrightarrow F$ sont deux fonctions quelconques et $\varphi : E \setminus A \longrightarrow E$ vérifie $\varphi(x) \prec x$.

Exemples de fonctions inductives :

- La fonction factorielle.

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

$E = \mathbb{N}, A = \{0\}, \varphi : n \longmapsto n-1.$

Fonctions inductives

Le principe d'induction permet de justifier la terminaison d'une fonction $f : E \to F$ définie par :

$$\begin{aligned} \forall a \in A, &\quad f(a) = g(a) \\ \forall x \in E \setminus A, &\quad f(x) = h\big(x, f \circ \varphi(x)\big) \end{aligned}$$

où $g : A \to F$ et $h : E \setminus A \times F \to F$ sont deux fonctions quelconques et $\varphi : E \setminus A \to E$ vérifie $\varphi(x) \prec x$.

Exemples de fonctions inductives :

- La fonction pgcd.

```
let rec pgcd = fun
  | 0 q -> q
  | p q -> pgcd (q mod p) p ;;
```

$E = \mathbb{N}^2, A = \big\{(0, q) \bigm| q \in \mathbb{N}\big\}, \varphi : (p, q) \longmapsto (q \mod p, p).$

Fonctions inductives

Le principe d'induction permet de justifier la terminaison d'une fonction $f : E \longrightarrow F$ définie par :

$$\begin{aligned} \forall a \in A, &\quad f(a) = g(a) \\ \forall x \in E \setminus A, &\quad f(x) = h\big(x, f \circ \varphi(x)\big) \end{aligned}$$

où $g : A \rightarrow F$ et $h : E \setminus A \times F \rightarrow F$ sont deux fonctions quelconques et $\varphi : E \setminus A \rightarrow E$ vérifie $\varphi(x) \prec x$.

Exemples de fonctions inductives :

- La fonction de Fibonacci (avec deux appels récursifs).

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

$E = \mathbb{N}, A = \{0,1\}, \varphi_1 : n \longmapsto n-1, \varphi_2 : n \longmapsto n-2.$

Fonctions inductives

Le principe d'induction permet de justifier la terminaison d'une fonction $f : E \to F$ définie par :

$$\begin{aligned} \forall a \in A, \quad & f(a) = g(a) \\ \forall x \in E \setminus A, \quad & f(x) = h\big(x, f \circ \varphi(x)\big) \end{aligned}$$

où $g : A \to F$ et $h : E \setminus A \times F \to F$ sont deux fonctions quelconques et $\varphi : E \setminus A \to E$ vérifie $\varphi(x) \prec x$.

Exemples de fonctions inductives :

- Plus généralement, toute fonction de la forme :

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

Avec

dans_A : *'a –> bool* **phi** : *'a –> 'a* **g** : *'a –> 'b* **h** : *'a –> 'b –> 'b*

Pile d'exécution d'une fonction

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

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :                          etiq_1 :
   dup               ;               dup            ;
   call fonc_dans_A  ;               call fonc_phi  ;
   jz etiq_1         ;               call fonc_f    ;
   call fonc_g       ;               call fonc_h    ;
                                     ret
```

Pile d'exécution d'une fonction

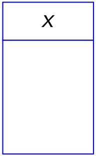
```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :
   dup              ;
   call fonc_dans_A ;
   jz etiq_1        ;
   call fonc_g      ;

etiq_1 :
   dup              ;
   call fonc_phi    ;
   call fonc_f      ;
   call fonc_h      ;
   ret
```

fonc_f :



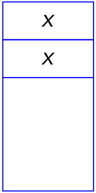
Pile d'exécution d'une fonction

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

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :                              etiq_1 :
   dup                ;                  dup             ;
   call fonc_dans_A ;                    call fonc_phi   ;
   jz etiq_1          ;                  call fonc_f     ;
   call fonc_g        ;                  call fonc_h     ;
                                         ret
```

dup :



Pile d'exécution d'une fonction

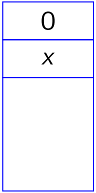
```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :
   dup               ;
   call fonc_dans_A  ;
   jz etiq_1         ;
   call fonc_g       ;
```

```
etiq_1 :
   dup               ;
   call fonc_phi     ;
   call fonc_f       ;
   call fonc_h       ;
   ret
```

fonc_dans_A :



Pile d'exécution d'une fonction

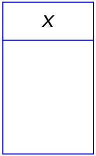
```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :
   dup              ;
   call fonc_dans_A ;
   jz etiq_1        ;
   call fonc_g      ;

etiq_1 :
   dup              ;
   call fonc_phi    ;
   call fonc_f      ;
   call fonc_h      ;
   ret
```

jz etiq_1 :



Pile d'exécution d'une fonction

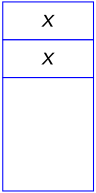
```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :
   dup              ;
   call fonc_dans_A ;
   jz etiq_1        ;
   call fonc_g      ;
```

```
etiq_1 :
   dup              ;
   call fonc_phi    ;
   call fonc_f      ;
   call fonc_h      ;
   ret
```

dup :



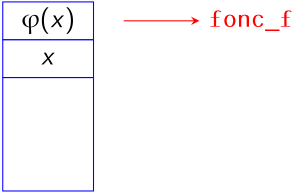
Pile d'exécution d'une fonction

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

La traduction en assembleur de cette fonction ressemble à :

```
fonc_f :                              etiq_1 :
   dup               ;                   dup               ;
   call fonc_dans_A ;                   call fonc_phi     ;
   jz etiq_1         ;                   call fonc_f       ;
   call fonc_g       ;                   call fonc_h       ;
                                         ret
```

fonc_phi :



Pile d'exécution d'une fonction

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

Illustration du débordement de capacité de la pile :

```
# let n = ref 0 ;;
n : int ref = ref 0
# let rec f () =
    n := !n + 1 ; 1 + f () ;;
f : unit -> int = <fun>
# try f () with Out_of_memory -> !n ;;
- : int = 131027
```

Au bout de 131 027 appels l'interprète de commande est à cours de mémoire.

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
```

$\boxed{5}$

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
```

4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
```

3
4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
```

2
3
4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
```

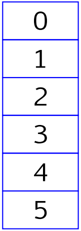
1
2
3
4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
```

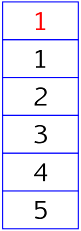


Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
```



Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
```

1
2
3
4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
```

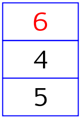
2
3
4
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
```



Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

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

24
5

Pile d'exécution d'une fonction

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

On peut « suivre à la trace » la fonction factorielle :

```
# 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
```

$\boxed{120}$

Récursivité terminale

Une fonction inductive est dite terminale lorsque l'appel récursif est la dernière opération qu'on effectue :

$$\begin{aligned} \forall a \in A, \quad & f(a) = g(a) \\ \forall x \in E \setminus A, \quad & f(x) = f\big(\varphi(x)\big) \end{aligned}$$

Récursivité terminale

Une fonction inductive est dite terminale lorsque l'appel récursif est la dernière opération qu'on effectue :

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

Dans ce cas, la pile d'exécution de la fonction ne croit pas : il n'y a pas de débordement de capacité.

```
# let n = ref 0 ;;
n : int ref = ref 0
# let rec f () =
    n := !n + 1 ; f () ;;
f : unit -> 'a = <fun>
# try f () with Out_of_memory -> !n
Interruption.
```

(il faut interrompre manuellement l'exécution de la fonction).

Récursivité terminale

Une fonction inductive est dite terminale lorsque l'appel récursif est la dernière opération qu'on effectue :

$$\begin{aligned} \forall a \in A, &\quad f(a) = g(a) \\ \forall x \in E \setminus A, &\quad f(x) = f\big(\varphi(x)\big) \end{aligned}$$

La fonction pgcd est récursive terminale :

```
let rec pgcd = function
  | (0, q) -> q
  | (p, q) -> pgcd (q mod p, p)  ;;
```

Récursivité terminale

Une fonction inductive est dite terminale lorsque l'appel récursif est la dernière opération qu'on effectue :

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

La fonction pgcd est récursive terminale :

```
let rec pgcd = function
  | (0, q) -> q
  | (p, q) -> pgcd (q mod p, p)  ;;
```

```
# pgcd (95, 115) ;;
pgcd <-- 95, 115
pgcd <-- 20, 95
pgcd <-- 15, 20
pgcd <-- 5, 15
pgcd <-- 0, 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
- : int = 5
```