1 Programme officiel & planning de l'année

Programme officiel : voir polycop.
Planning :

- 00 - Induction, type Caml preuve de programmes
- 01 - Complexité
- 02 - Le type arbre
- 03 - Mots et automates
- 04 - Logique
- 05 - Problèmes de révision

2 Révision de Caml

voir polycop : Memento Caml.

2.1 Type somme et type produit (enregistrement)

On peut définir un type de deux façons en Caml :

- soit en faisant une union disjointe de types déjà connus, c'est ce que l'on appelle un *type somme* ;
- soit en faisant un produit cartésien de types existants, ce que l'on appelle un *type produit*.

On peut bien sûr mélanger ces deux constructions.
Les définitions de type se font à l'aide du mot clé `type`. L'identificateur de l'union disjointe dans les types somme est la barre verticale '`|`'. On peut par exemple définir un type somme `animal_de_compagnie` ainsi :

```
type animal_de_compagnie = Chien | Chat | Oiseau;;
Le type animal_de_compagnie est d\'{e}fini.
```

Les types produits sont analogues aux *records* de Pascal. Il s'agit d'enregistrements dont chaque composante est nommée (on parle d'*étiquette*). La syntaxe en est la suivante :

```
type  nom-du-type = {Etiquette1 :  type-de-l-etiquette1;
                     Etiquette2 :  type-de-l-etiquette2;
                            ...};;
```

Par exemple, on peut définir les nombres complexes sous la forme de couples de réels dont la première composante est appelée `Partie_reelle` et la seconde `Partie_imaginaire` :

```
#type complexe =
{Partie_reelle : float; Partie_imaginaire : float};;
Le type complexe est d\'{e}fini.
#let i={Partie_reelle = 0.0 ;
Partie_imaginaire = 1.0};; i : complexe = {Partie_reelle = 0.0;
Partie_imaginaire = 1.0}
```

La définition d'objets du nouveau type se fait à l'aide d'accolades et du signe `=`. L'accès aux composantes d'un objet du type enregistrement se fait à l'aide du signe $\cdot$, comme pour un vecteur. La conjugaison dans $\mathbb{C}$ peut être définie par :

```
#let conjugaison {Partie_reelle = a ; Partie_imaginaire = b} =
________        {Partie_reelle = a ; Partie_imaginaire = -.b};;
conjugaison : complexe -> complexe = <fun>
#let i_barre = conjugaison i;;
i_barre : complexe = {Partie_reelle = 0.0; Partie_imaginaire = -1.0}
#i_barre.Partie_imaginaire;;
- : float = -1.0
```

Les composantes d'un enregistrement peuvent contenir tout type connu (en particulier des types somme).
Signalons qu'il existe aussi des *abréviations de type* (qu'il ne faut pas confondre avec une définition de type) sous la syntaxe :

`type` *`nom-de-l-abréviation`* `==` *`type-à-abréger`*

Par exemple :

```
#type point == float * float;;
Le type point est d\'{e}fini.
```

et par la suite l'abréviation `point` pourra être utilisée en lieu et place de `float*float` (notamment lorsque l'on désire forcer un type).

```
#let rec premi\`{e}re_projection = function
   | [] -> []
   | (x,y)::q -> x::(premi\`{e}re_projection q);;
premi\`{e}re_projection : ('a * 'b) list -> 'a list = <fun>
#let rec premi\`{e}re_projection = function
   | [] -> []
   | ((x,y):point)::q -> x::(premi\`{e}re_projection q);;
premi\`{e}re_projection : point list -> float list = <fun>
#let (translation: point -> point) = function
    (x,y) -> x +. 1., y +. 1.;;
translation : point -> point = <fun>
```

2.2 Type mutable

Si l'on désire qu'une étiquette d'un type enregistrement puisse évoluer en cours de session, on doit la déclarer, lors de la définition du type, comme `mutable`. Cela donne :

```
#type pere_de_famille = {Nom : string ; mutable Age : int ;
                         Enfants : string list ;
                         Animaux : animal_de_compagnie list};;
Le type pere_de_famille est d\'{e}fini.
```

La modification d'une étiquette mutable se fait à l'aide du signe `<-` :

```
#let p\`{e}re =
{Nom = "jean" ; Age = 60 ; Enfants = ["marie"; "paul"] ;
                Animaux = [chien]};;
p\`{e}re : pere_de_famille =
 {Nom = "jean"; Age = 60; Enfants = ["marie"; "paul"];
                Animaux = [chien]}
#let rajeunir x = x.age <- 20;;
- : unit = ()
#rajeunir p\`{e}re;;
- : unit = ()
#p\`{e}re;;
- : pere_de_famille =
 {Nom = "jean"; Age = 20; Enfants = ["marie"; "paul"];
                          Animaux = [chien]}
```

2.3 Type avec argument

On peut en Caml également paramétrer les constructeurs de types de la façon suivante :

`type` *`Identificateur`* `of` *`type-déjà-connu`*,

un élément du nouveau type étant désigné par :

`Identificateur(`*`valeur-du-type-déjà-connu`*`)`.

Ceci peut permettre de créer un nouveau type, que nous appellerons `nombre` destiné à contourner les distinctions syntaxiques entre flottants et entiers :

```
#type nombre = Entier of int | Reel of float;;
Le type nombre est d\'{e}fini.
#let x = Entier(1);;
x : nombre = Entier 1
#let y=Reel(2.5);;
y : nombre = Reel 2.5
#let add = fun
    | (Entier n) (Entier m) -> Entier (n+m)
    | (Entier n) (Reel x)   -> Reel ((float_of_int n) +. x)
    | (Reel x)   (Entier n) -> Reel (x +. (float_of_int n))
    | (Reel x)   (Reel y)   -> Reel (x +. y);;
add : nombre -> nombre -> nombre = <fun>
##infix "add";;
#x add y;;
- : nombre = Reel 3.5
```

(remarquer ici l'usage de `fun`).

2.4 Type polymorphe

On peut très bien employer des variables de type dans les constructions précédentes et définir ainsi des types polymorphes. En voici un exemple avec des enregistrements :

```
#type 'a boite = {Provenance : string ; Contenu : 'a list};;
Le type boite est d\'{e}fini.
#let boulier = {Provenance = "chine" ;
                Contenu = [0;1;2;3;4;5;6;7;8;9]};;
boulier : int boite =
 {Provenance = "chine";
  Contenu = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]}
```

Voici un autre exemple :

```
#type 'a paire = Paire of 'a  * 'a ;;
Le type paire est d\'{e}fini.
#let anglais_francais = Paire (["un";"deux";"trois"],
                               ["one";"two"; "three"]);;
anglais_francais : string list paire =
 Paire (["un"; "deux"; "trois"], ["one"; "two"; "three"])
```

Remarquer que l'objet `Paire(1,"un")` n'est pas reconnu comme étant du type `'a paire` puisque les types des deux composantes sont différents :

```
#Paire(1,"un");;
Entr\'{e}e interactive:
>Paire(1,"un");;
>        ^^^^
Cette expression est de type string,
mais est utilis\'{e}e avec le type int.
```

2.5 Type récursif

On peut très bien définir en Caml des objets dont la structure est récursive. Prenons l'exemple de la structure de donnée arbre binaire. Il s'agit d'objets informatiques définis ainsi : un arbre binaire est soit un noeud avec un fils gauche et un fils droit qui sont eux-mêmes des arbres binaires, soit tout simplement une feuille (on décide d'étiqueter les feuilles de l'arbre avec des entiers).

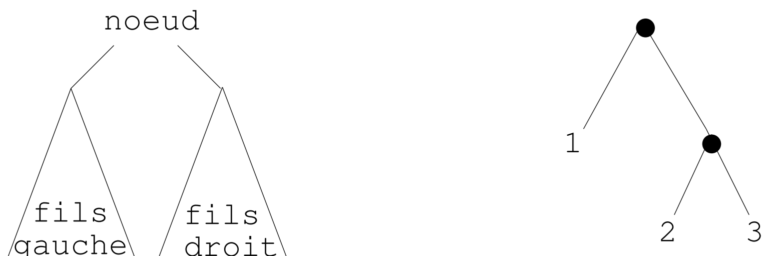

FIG. 1: La structure d'arbre binaire accompagnée d'un exemple.

Les arbres sont amplement étudiés en deuxième année.
Cette définition récursive se traduit immédiatement en Caml par :

```
type arbre_binaire = Feuille of int
                   | Noeud of  arbre_binaire * arbre_binaire;;
```

Voici le codage de l'exemple d'arbre de la figure 1 :

```
#let mon_arbre =
           Noeud ( Feuille 1, Noeud (Feuille 2, Feuille 3));;
mon_arbre : arbre_binaire =
           Noeud (Feuille 1, Noeud (Feuille 2, Feuille 3))
```

et voici, par exemple, une fonction comptant le nombre de feuilles d'un arbre binaire donné :

```
#let rec nombre_de_feuilles  = function
    | (Feuille n) -> 1
    | Noeud (sous_arbre_droit, sous_arbre_gauche) ->
                   (nombre_de_feuilles sous_arbre_droit) +
                   (nombre_de_feuilles sous_arbre_gauche);;

nombre_de_feuilles : arbre_binaire -> int = <fun>
#nombre_de_feuilles mon_arbre;;
- : int = 3
```

Signalons que le type `'a list` prédéfini en Caml pourrait être redéfini simplement de la façon suivante :

```
#type 'a liste = Liste_vide | Conse of 'a*'a liste;;
Le type liste est d\'{e}fini.
#Conse (1, Liste_vide);;
- : int liste = Conse (1, Liste_vide)
```

2.6 Types mutuellement récursifs

Voici un exemple d'école :

```
#type alpha = A | Mot   of alpha * beta
   and beta = B | Suite of beta  * alpha;;
Le type alpha est d\'{e}fini.
Le type beta est d\'{e}fini.
#let essai1 = Mot ( Mot (A, Suite (B,A)), B);;
essai1 : alpha = Mot (Mot (A, Suite (B,A)), B)
#let essai2 = Suite (B,essai1);;
essai2 : beta = Suite (B, Mot (Mot (A, Suite (B,A)), B))
```

3 Preuves de programme

3.1 introduction

Il a été démontré au début du XX$^{\text{ième}}$siècle[1] qu'il était impossible d'espérer trouver un programme universel sachant tester si une fonction récursive quelconque termine son calcul.
Imaginons en effet un monde merveilleux dans lequel existerait une fonction `termine` de type `'a -> 'b -> bool` qui répondrait `true` si la fonction récursive passée en argument (de type général `'a -> 'b`) termine et `false` sinon. Par exemple, `termine(somme)` renverrait `true` et `termine(sans_fin)` renverrait `false` avec `sans_fin` la fonction définie par :

```
let rec sans_fin x = if x then sans_fin x else false;;
```

(considérer l'exécution de `sans_fin true`).
Hélas l'essai de `termine` sur la fonction `test` suivante montre que nous ne vivons pas dans ce monde :

```
let rec test () = if (termine test) then (test ()) else true;;
```

Si `test ()` ne termine pas alors `termine test` retourne `false` et la valeur de `test ()` est `true` et donc `test ()` termine. Et si `test ()` termine alors `termine test` retourne `true` et `test ()` est à nouveau évalué et la fonction boucle. Il y a donc une contradiction et une telle fonction `termine` ne peut, et c'est bien dommage, exister.
Prouver un programme est donc quelque chose de difficile (voire impossible) et nous allons réviser quelques "recettes" qui s'appliquent dans les cas simples du programme officiel.

3.2 Preuve de programme itératif

Ce qui suit est **fortement** inspiré du travail d'Y. Lemaire et ne constitue qu'une petite introduction au *Génie logiciel*.

- **Spécifications d'un bloc d'instructions**

Le contexte d'un bloc de programme est l'ensemble des objets auxquels ce bloc peut accéder ; soit pour utiliser les valeurs de ces objets, soit, éventuellement, pour les modifier.

Définition 3.2.1 *Soit un bloc d'instructions P de contexte E et soient deux propriétés $C1$ et $C2$ portant sur les objets de E. On dit que P satisfait la précondition $C1$ et la postcondition $C2$ quand la propriété suivante est vérifiée : si le contexte E vérifie $C1$, alors, après exécution des instructions du bloc P, il vérifie $C2$.*

On peut schématiser cette propriété par :

$$C_1 \xrightarrow{P} C2$$

Une propriété I qui vérifie $I \xrightarrow{P} I$ est appelée un *invariant* de P.

[1] C'est l'oeuvre des logiciens Gšdel et Turing dans les années 1930.

- **Spécifications d'une fonction**

On appelle spécifications d'une fonction f la donnée
- du type $T1$ de l'argument x ;
- du type $T2$ de la valeur renvoyée ;
- d'une condition $C1$, appelée précondition, portant sur x et, éventuellement, sur les objets du contexte de la fonction ;
- d'une condition $C2$, appelée postcondition, portant sur la valeur renvoyée et, éventuellement, sur les objets du contexte de la fonction. On dit que la fonction f *satisfait* ces spécifications si, pour tout x, de type $T1$, vérifiant la condition $C1$, un appel de $f(x)$ renvoie une valeur de type $T2$ vérifiant la condition $C2$.

- **Affectation**

On a

$$C(a) \xrightarrow{x \leftarrow a} C(x)$$

Par exemple, si x est une variable entière et si c est une valeur entière, on a :

$$(c \geqslant 1) \xrightarrow{x \leftarrow c+1} (x \geqslant 2)$$

- **Séquence**

Soient P et Q deux blocs d'instructions. Notons $P; Q$ ou simplement PQ, le bloc obtenu en ajoutant les instructions du bloc Q à la suite de celles de P :

$$\text{Si } C1 \xrightarrow{P} C2 \text{ et } C2 \xrightarrow{Q} C3, \text{ alors } C1 \xrightarrow{PQ} C3$$

- **Répétition (boucle for)**

Pour ce type de boucle, c'est évidemment la validité, et non la terminaison, qui pose problème.
Considérons l'instruction : pour $k = p$ à q faire action(k) (l'instruction, ou le bloc d'instructions, action(k) est appelé le *corps* de la boucle).
Soit une propriété $I(k)$ un invariant de cette boucle ; c'est à dire que, pour tout $k \in [\![p; q]\!]$, action(k) satisfait la précondition $I(k-1)$ et la postcondition $I(k)$. Alors la boucle satisfait la précondition $I(p-1)$ et la postcondition $I(q)$. Autrement dit

$$[\forall\ k, I(k-1) \xrightarrow{action(k)} I(k)] \Longrightarrow [I(p-1) \xrightarrow{pour\ k=p\quad q\ faire\ action(k)} I(q)]$$

Cela peut être considéré comme une conséquence de la propriété des séquences car la boucle peut aussi s'écrire : action(p) ; action(p + 1) ; ... ; action(q).

Pratique Ainsi, pour montrer que la propriété $I(q)$ est vérifiée après l'exécution de la boucle, il suffit de montrer
- que $I(p-1)$ est vérifiée avant l'exécution de la boucle ;
- et que $I(k)$ est un invariant de la boucle.

Exemple(s). Exponentiation lente.
Montrons que la fonction `puiss : int -> int -> int` suivante

```
let puiss x n = let y = ref 1 in for k = 1 to n do y := !y * x done; !y;;
```

est telle que `puiss x n` renvoie x^n : La propriété $y = x^k$ est un invariant de la boucle car $(y = x^{k-1}) \xrightarrow{y \leftarrow y \times x} (y = x^k)$. De plus, avant l'exécution de la boucle, on a $y = 1 = x^0$; donc, après la boucle, on a $y = x^n$. Dans la pratique, pour démontrer la validité d'un bloc d'instructions, il est préférable, autant que possible, de faire apparaître la preuve dans les commentaires (entre (* *)) du bloc, comme suit :

```
let puiss x n = (* int -> int -> int, puiss x n = x^n *)
let y = ref 1 in (* y = x^0 *)
for k = 1 to n do (* y = x^(k-1) *)
y :=!y *x (* y = x^k *)
done; (* y = x^n *)
!y;;
```

Dans cet exemple, les commentaires sont d'ailleurs trop nombreux. Le seul commentaire indispensable est `(* y = x^(k-1) *)` qui permet d'en déduire les autres.
Complexité : Un appel de `puiss x n` effectue n multiplications.

Il faut bien comprendre que l'obligation de prouver, en introduisant le bon invariant de boucle, qu'une boucle effectue bien le travail qu'on lui demande n'est pas une contrainte ; c'est aussi une aide à la mise au point de cette boucle. Ce point de vue est expliqué dans l'exemple suivant.

Exemple(s). Evaluation d'un polynôme.
$P = \sum_{i=0}^{n} p_i X^i \in \mathbb{R}[X]$ et $x \in \mathbb{R}$ étant donnés, on se propose de calculer $P(x) = \sum_{i=0}^{n} p_i x^i$. On introduit une variable réelle y et on essaie d'écrire une boucle pour $k = 1$ à n faire action(k) admettant la propriété ($y = \sum_{i=0}^{k} p_i x^i$) pour invariant, de manière à récupérer à la sortie de la boucle la valeur cherchée dans y. Pour cela, il suffit de prendre l'instruction : $y \leftarrow y + p_k x^k$ pour action(k). Mais cette instruction est coûteuse car elle nécessite le calcul de x^k et elle ne tient pas compte du fait que x^{k-1} a été calculé à l'étape précédente. Il est préférable d'introduire une deuxième variable z et d'imposer l'invariant $I(k)$: ($z = x^k{}_\wedge\ y = \sum_{i=0}^{k} p_i x^i$). action($k$) devient alors : $z \leftarrow x \times z; y \leftarrow y + p_k z$. Pour que $I(n)$ soit vérifiée à la sortie de la boucle, il suffit donc que $I(0)$ soit vérifié à l'entrée : on écrit avant la boucle les instructions d'initialisation : $z \leftarrow 1; y \leftarrow p_0$.
En Caml , cela donne le programme :

```
let eval_polynome p x = (* float vect -> float -> float *)
(* eval_polynome p x renvoie pn x^n + ... + p0 o\'{u} n = longueur(p) - 1 *)
let z = ref 1.
and y = ref p.(0) in
for k = 1 to vect_length p - 1 do
z := x *. !z;
y := !y +. p.(k) *. z
(* z = x^k et y = pk x^k + ... + p0 *)
done;
!y;;
```

Complexité : Un appel de `eval_polynome p x` nécessite 2n multiplications et n additions. La méthode de Hšrner est plus efficace : n multiplications et n additions.

```
let eval_polynome_Horner p x =
let n = vect_length p - 1 in
let y = ref p.(n) in
for k = n-1 downto 0 do
y := !y *. x +. p.(k)
(* y = pn x^(n-k) + ... + pk *)
done;
!y;;
```

• Répétition (boucle while)

Ici (et tout le monde l'a expérimenté au moins une fois !) en plus de la validité, la terminaison est une propriété foncdamentale à prouver. Nous reprendrons ceci dans le paragraphe suivant consacré à l'induction.
Considérons l'instruction : tant que C faire action (action est appelé le *corps* de la boucle).

Proposition 3.2.2 (Correction partielle) *Soit une propriété I, un invariant de la boucle C, dans le sens où $I_\wedge\ C \xrightarrow{action} I$ alors, en supposant que la boucle termine (ne soit pas une boucle infinie), on a $I \xrightarrow{tant\ que\ C\ faire\ action} I_\wedge\ \neg C$.*

Pratique En d'autres termes, pour montrer qu'une propriété P est vérifiée à la fin de l'exécution de la boucle (en supposant qu'elle termine), il suffit de trouver une propriété I telle que
- I est vérifiée à l'entrée dans la boucle ;
- $I_\wedge\ C \xrightarrow{action} I$ (I est un invariant) ;
- $I_\wedge\ \neg C \Rightarrow P$.

Cela se comprend aisément : la propriété I est constamment vérifiée (avant et après chaque exécution du corps de la boucle). En particulier, elle est vérifiée à la fin de l'exécution de la boucle. De plus la propriété $\neg C$ est aussi vérifiée après la boucle car c'est la condition de sortie.

Exemple(s). Exponentiation rapide.
Soit à calculer x^n avec $x \in \mathbb{Z}$ (mais on pourrait supposer que x appartient à un monoïde quelconque) et $n \in \mathbb{N}$. Dans notre premier programme, effectuons le changement d'indice $p = n - k + 1$ et transformons la boucle for en une boucle while. On obtient le programme suivant :

```
(* Exponentiation lente avec boucle while *)
let puiss' x n =
let y = ref 1 and p = ref n in
while !p > 0 do
(* invariant : y = x^(n-p) *)
y :=!y *x;
p :=!p -1
done;
!y;;
```

Preuve d'arrêt : La variable entière p reste > 0 et décroît strictement dans le corps de la boucle.
Preuve de correction partielle : L'invariant étant donné en commentaire dans l'algorithme, la preuve de correction partielle consiste seulement à vérifier que
- la propriété $I : y = x^{n-p}$ est bien un invariant de la boucle : $x^{n-p} \times x = x^{n-(p-1)}$;
- I est vérifiée à l'entrée : $1 = x^{n-n}$;
- si I est vérifiée et si, de plus (p >0) n'est pas vérifiée, alors $y = x^n$: $p = 0$.

En règle générale, il suffit de préciser l'invariant qui sert à faire la preuve de correction partielle car la preuve elle-même ne consiste qu'en vérifications.
L'inefficacité du programme précédent (nombre de multiplications = n) provient du fait que, pour maintenir l'invariant $y = x^{n-p}$, qui peut aussi s'écrire : $yx^p = x^n$, la variable p ne décroît que d'une unité à chaque exécution du corps de la boucle. Introduisons une nouvelle variable z et tâchons de maintenir l'invariant $J : yz^p = x^n$ tout en diminuant p de motié :

- si p est pair, $yz^p = y(z^2)^{p/2}$ donc $J \xrightarrow{z \leftarrow z^2; p \leftarrow p/2} J$
- si p est impair, $yz^p = yz(z^2)^{(p-1)/2}$ donc $J \xrightarrow{y \leftarrow yz; z \leftarrow z^2; p \leftarrow (p-1)/2} J$.

Cela montre la correction partielle de la boucle du programme suivant :

```
(* Exponentiation rapide version inp\'{e}rative *)
let puiss_rapide x n =
let y = ref 1 and z = ref x and p = ref n in
while !p > 0 do
(* Invariant : y z^p = x^n *)
if !p mod 2 = 1 then y := !y * !z;
z :=!z *!z;
p :=!p /2
done;
!y;;
```

Preuve d'arrêt : $p \in \mathbb{N}$ décroît strictement dans le corps de la boucle.
Complexité : Le nombre T_n de multiplications est majoré par $2U_n$ où U_n est le nombre d'exécutions du corps de la boucle. U_n est aussi le nombre d'exécutions du corps de la boucle simplifiée : `while !p > 0 do p := !p / 2 done`.
On voit alors que $U_n = 1 + U_{\lfloor n/2 \rfloor}$, ce qui montre (cf. le polycop de complexité) que $U_n = O(\ln n)$ et $T_n = O(\ln n)$.

3.3 Fonction récursive

Soit une fonction récursive c'est-à-dire dont l'expression fait elle-même appel à la fonction f. Nous reprendrons les preuves et la récursivités dans la section consacrée à l'induction. Donnons simplement, pour clore cette section, la version récursive de l'exponentiation rapide.

Exemple(s). Pour $x \in \mathbb{Z}$ et $n \in \mathbb{N}$, $x^n = (x^{n/2})^2$ si n est pair et $(x^{(n-1)/2})^2 x$ sinon; ce qui prouve la correction partielle de la sous fonction récursive p suivante :

```
(* Exponentiation rapide version r\'{e}cursive *)
let rec  p x = function
   | 0 ->1
   | n -> let y =p x (n/2) in
          if n mod 2 = 0 then y * y else y * y * x;;
```

Preuve d'arrêt : Supposons qu'un appel de `p x n` donne lieu au sous appel récursif `p x (n/2)`. Alors n est non nul donc l'argument $n/2$ de p dans le sous appel est strictement inférieur à l'argument n de p dans l'appel principal. Complexité : Le nombre T_n de multiplications effectuées par un appel de `p x n` vérifie $T_n \leqslant 2 + T_{\lfloor n/2 \rfloor}$ donc (cf. le polycop de complexité) $T_n = O(\ln n)$.

4 Induction

4.1 Ensemble bien fondé

Nous allons formaliser la notion intuitive suivante. Si l'on dispose d'une quantité liée à la fonction récursive et à ses arguments telle que :
- cette quantité évolue dans un ensemble ordonné qui ne comporte pas de suite infinie strictement décroissante,
- à chaque appel de la fonction, la quantité en question décroît strictement,
- la fonction récursive achève son calcul pour les cas de base,
alors la fonction termine bien dans tous les cas.

Exemple(s).

```
let rec tri_rapide  = function
  |  []    -> []
  |  [e]   -> [e]
  |  e::r  -> let l1,l2 = partition  e r in
              (tri_rapide l1)@(e::(tri_rapide l2));;
```

avec `partition` est la fonction qui réalise la séparation en deux sous-listes en fonction de l'élément de tête (nous suivons une mise au point "de haut en bas") :

```
#let rec partition (e:int)  = function
__|_[] -> ([], [])
__| d::r -> let l1,l2 = partition e r in
___if (d < e) then (d::l1,l2) else (l1,d::l2);;
partition : int -> int list -> int list * int list = <fun>
#partition 18 [3;10;25;9;3;11;13;23;8];;
- : int list * int list = [3; 10; 9; 3; 11; 13; 8], [25; 23]
```

Ici, la définition de `tri_rapide` ne fait appel qu'à des listes de longueur strictement inférieure, cette quantité décroît strictement pour atteindre 1 ou 0 qui correspondent aux longueurs des deux cas de base prévus.

Donnons quelques définitions et propriétés afin de préciser les choses sur le plan mathématique. Dans la suite, E va désigner un ensemble non vide, $\preccurlyeq$ une relation d'ordre (partiel ou total) définie sur E et $\prec$ la relation d'ordre stricte associée ($\leqslant$ et $<$ continuent de désigner l'ordre usuel (et sa version stricte) sur les entiers, les réels...).

Définition 4.1.1 *$(E, \preccurlyeq)$ est dit* bien fondé[2] *s'il n'existe pas de suite infinie d'éléments de E strictement décroissante.*

Par exemple, $(\mathbb{N}, \leqslant)$ est bien fondé alors que $(\mathbb{Z}, \leqslant)$ ne l'est pas.
On a alors immédiatement la propriété (que nous utiliserons souvent par la suite sans le dire...) :

Proposition 4.1.2 *Si $(E, \preccurlyeq)$ est bien fondé et si F est une partie non vide de E alors $(F, \preccurlyeq)$ est bien fondé.*

Comme exemple moins élémentaire, on peut considérer un ordre très utile pour la suite : l'ordre *lexicographique*. Sur $\mathbb{N}^2$, il est défini par :

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

Il s'agit d'un ordre total qui peut être facilement généralisé[3] à $\mathbb{N}^p$. On a :

Proposition 4.1.3 *$\mathbb{N}^2$ muni de l'ordre lexicographique est bien fondé.*

[2]On trouve aussi la terminologie de *bien ordonné* pour un ensemble muni d'un ordre *total* bien fondé. Les preuves qui vont suivre ne dépendant pas du caractère partiel ou total de l'ordre, nous ne ferons pas de distinction et utiliserons le terme "bien fondé" dans les deux cas.

[3]À l'aide de l'ordre usuel sur les lettres de l'alphabet, cette construction fournit, sur les mots, l'ordre du dictionnaire.

PREUVE. Supposons l'existence d'une suite infinie $(a_n, b_n)_{n\in\mathbb{N}}$ de couples d'entiers, strictement décroissante. Tous les couples étant strictement inférieurs à (a_0, b_0), il existe un rang n_0 à partir duquel $a_n < a_0$ (en effet, l'ensemble des couples de $\mathbb{N}^2$ inférieurs strictement à (a_0, b_0) ayant la même première coordonnée que (a_0, b_0) est de cardinal fini b_0). De même, il existe un rang $n_1 \geqslant n_0$ à partir duquel $a_n < a_{n_0} < a_0$. En itérant cette construction, on obtient une suite infinie d'entiers, la suite des premières coordonnées a_{n_i}, qui est strictement décroissante dans $\mathbb{N}$ ce qui constitue une contradiction. □

On dispose dans un ensemble bien fondé de la propriété caractéristique suivante :

Proposition 4.1.4 *Un ensemble $(E, \preccurlyeq)$ est bien fondé si et seulement si toute partie non vide de E admet un élément minimal.*

Rappelons, avant de prouver cette proposition, que, si l'ordre est total, la notion d'élément minimal coïncide avec celle de minimum. Dans le cas d'un ordre partiel, un élément m d'une partie non vide $A \subset E$ est dit élément minimal de A si

$$\forall\, a \in A, \quad a \preccurlyeq m \;\Rightarrow\; m = a$$

Par exemple, dans $\mathbb{N} \setminus \{0, 1\}$ muni de la relation d'ordre partiel $|$ ("divise"), les éléments minimaux sont les nombres premiers.
Vous vérifierez de même que, si $F = \{a, b, c\}$ est un ensemble contenant trois éléments, les éléments minimaux de $E = \mathcal{P}(F) \setminus \{\emptyset\}$, l'ensemble des parties non vides de F, muni de la relation d'ordre partiel $\subset$ ("est inclus dans"), sont les trois singletons $\{a\}$, $\{b\}$ et $\{c\}$.
Revenons à la preuve de la proposition :
PREUVE. Soit E un ensemble non vide dans lequel toute partie non vide admet un élément minimal. Supposons l'existence de $(u_n)_{n\in\mathbb{N}}$ une suite infinie d'éléments de E telle que $\forall\, n \in \mathbb{N},\ u_{n+1} \prec u_n$. Considérons alors le support de la suite $S = \{u_n, n \in \mathbb{N}\}$ et m un élément minimal de S. On dispose donc de $n_0 \in \mathbb{N}$ tel que $m = u_{n_0}$. L'élément u_{n_0+1} apporte la contradiction : c'est un élément de S qui est, par hypothèse, strictement plus petit que m. Une telle suite $(u_n)_{n\in\mathbb{N}}$ n'existe donc pas et E est bien fondé.
Réciproquement, soit $(E, \preccurlyeq)$ un ensemble bien fondé et soit $A \subset E$ une partie non vide de E sans éléments minimaux. On dispose donc de la négation de la définition donnée précédemment, soit :

$$\forall\, m \in A, \quad \exists\, a \in A \quad \text{tel que } a \prec m$$

La partie A est nécessairement de cardinal infini et on construit alors aisément (par récurrence) une suite d'éléments de A strictement décroissante infinie, ce qui est contraire à l'hypothèse. □

On peut illustrer notre proposition dans $\mathbb{N}^2$ muni de l'ordre lexicographique. Le minimum d'une partie A non vide est le couple (n_0, m_0) défini par : $n_0 = \min\{n \in \mathbb{N} \mid \exists\, m \in \mathbb{N}, (n, m) \in A\}$ et $m_0 = \min\{m \in \mathbb{N} \mid (n_0, m) \in A\}$.

Théorème 4.1.5 (Terminaison)
Soit f une fonction récursive, $\mathcal{A}$ l'ensemble de ses arguments, φ une application de $\mathcal{A}$ dans un ensemble bien fondé $(E, \preccurlyeq)$ et $\mathcal{B}$ la partie de $\mathcal{A}$ constituée des éléments dont l'image par φ est minimale dans $\varphi < \mathcal{A} >$ (les cas de base). Si $f(b)$ termine pour tout $b \in \mathcal{B}$ et si dans la définition de $f(x)$ n'apparaissent, en nombre fini, que des appels à $f(y)$ tels que $\varphi(y) \prec \varphi(x)$ alors $f(x)$ termine pour tout x dans $\mathcal{A}$.

PREUVE. Faisons une démonstration par l'absurde.
Supposons non vide la partie $\mathcal{X}$ constituée des éléments de $\mathcal{A}$ sur lesquels f ne termine pas. Soit $F = \varphi < \mathcal{X} >$. En tant que partie non vide de E, on dispose de m_0 un élément minimal de F. Notons x_0 un antécédent par φ de m_0 dans $\mathcal{X}$.
Comme f termine sur tous les arguments dont l'image par φ est un élément minimal de $\varphi < \mathcal{A} >$, m_0 n'est pas minimal dans $\varphi < \mathcal{A} >$. L'ensemble $M = \{m \in \varphi < \mathcal{A} > \ \mid m \prec m_0\}$ est donc non vide. D'autre part, puisque m_0 est minimal dans F, on a $M \subset \varphi < \mathcal{A} > \setminus F$ et f termine pour tous les arguments dont l'image par φ appartient à M.
Finalement, le calcul de $f(x_0)$, faisant directement appel à un nombre fini de $f(y)$ avec $\varphi(y) \prec \varphi(x_0) = m_0$, termine : ce qui constitue une contradiction avec $x_0 \in \mathcal{X}$. □

Proposition 4.1.6 (Induction sur un ensemble bien fondé)
Soit $(E, \preccurlyeq)$ un ensemble bien fondé et $\mathfrak{p}$ un prédicat sur les éléments de E. Si $\mathfrak{p}$ est vérifiée sur tous les éléments minimaux de E et si

$$\forall\, x \in E, \quad \Big([\forall\, y \prec x,\ \mathfrak{p}(y)] \Rightarrow \mathfrak{p}(x)\Big)$$

alors $\forall\ x \in E,\ \mathfrak{p}(x)$.

PREUVE. Effectuons une démonstration par l'absurde.
Supposons non vide la partie F constituée des éléments de E sur lesquels $\mathfrak{p}$ est faux. On dispose alors d'un élément minimal m_0 dans F. Donc $\forall\ y \prec m_0,\ y \notin F$ et donc $\mathfrak{p}(y)$ est vrai. De (1) on déduit $\mathfrak{p}(m_0)$: ce qui constitue la contradiction. □

Théorème 4.1.7 (Correction)

Soit f une fonction récursive, $\mathcal{A}$ l'ensemble de ses arguments, φ une application de $\mathcal{A}$ dans $(E, \preccurlyeq)$ un ensemble bien fondé, $\mathcal{B}$ la partie de $\mathcal{A}$ constituée des éléments dont l'image par φ est minimale dans $\varphi < \mathcal{A} >$ (les cas de base) et $\mathfrak{p}_f$ un prédicat sur les valeurs calculées par f.
Si $\mathfrak{p}_f(b)$ est vérifié pour tout b de $\mathcal{B}$ et si, en notant de nouveau (y) les arguments d'appel de f intervenant dans la définition de la fonction et x l'argument d'appel initial, on a :
- $\forall\ y,\ \varphi(y) \prec \varphi(x)$,
- $[\forall\ y,\ \mathfrak{p}_f(y)] \Rightarrow \mathfrak{p}_f(x)$,

alors $\mathfrak{p}_f$ est vérifié sur toutes les valeurs calculées par f.

PREUVE. Il suffit d'appliquer la proposition précédente au prédicat $\mathfrak{q}$ défini sur l'ensemble bien fondé $\varphi < \mathcal{A} >$ par : $\mathfrak{q}(e)$ est vrai si $\forall\ x \in \mathcal{A}$ tel que $\varphi(x) = e$ on a $\mathfrak{p}_f(x)$ vrai.

□

On notera la similitude des théorèmes de terminaison et de correction (substituez "$f(x)$ termine" par "$\mathfrak{p}_f(x)$"). De plus, *comme le test de $\mathfrak{p}_f(x)$ n'a de sens que si le calcul de $f(x)$ termine*, on mène en général de front les preuves de terminaison et de correction[4].

4.2 preuve de programmes récursif

99% du temps une injection dans $\mathbb{N}$ suffit à prouver un prédicat sur un programme (notamment sa terminaison). En pratique, on associe à la fonction récursive un entier qui décroit strictement à chaque appel de celle-ci.

Exemple(s). •

```
let rec fib(n) =
   if n<2 then n else fib(n-1)+fib(n-2);;
```

Cette formulation est *terriblement* inefficace ; en effet, notons τ_n le nombre d'appels à `fib` requis pour évaluer F_n : $\tau_0 = \tau_1 = 1$, et $\tau_n = 1 + \tau_{n-1} + \tau_{n-2}$ pour $n \geqslant 2$. On en déduit $\tau_n \geqslant F_n$, avec F_n le $n^{\text{ième}}$nombre de Fibonacci or $F_n \sim_{n\to\infty} \frac{\varphi^n}{\sqrt{5}}$ où $\varphi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or. Le coût est donc *exponentiel.*

Remarque(s). On peut facilement donner une formulation itérative, puisque la suite est définie par une récurrence du deuxième ordre :

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

donc $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$. Voici une réalisation en Caml :

```
let fib(n) =
   let f(x,y)=(x+y,x) and p2(x,y)=y and u=ref(1,0) in
      for k=1 to n do u:=f(!u) done; p2(!u);;
```

Le coût de la version itérative est donc linéaire. On peut faire encore mieux : puisque, en fait, il s'agit de calculer A^n, où A est la matrice $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, on aura un coût logarithmique en utilisant l'algorithme d'exponentiation rapide.

Voyons à présent d'autres exemples.
Le calcul des coefficients binomiaux peut se faire à l'aide de la formule du triangle de Pascal :

$$\forall\ (n,p) \in \mathbb{N}^2, \quad 1 \leqslant p < n, \qquad \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

[4]Souvent même, les prédicats considérés ne mentionnent pas explicitement la terminaison du calcul et ne font que la sous-entendre (car elle reste, bien sûr, indispensable).

Ce qui donne la fonction récursive suivante définie sur $\mathbb{N}^2$:

```
let rec binome = function
    | (n,0) -> 1
    | (n,p) -> if (p>n)
               then 0
               else binome ((n-1),p) + binome ((n-1),(p-1));;
```

Cette méthode est assez inefficace car des calculs de binomiaux identiques sont effectués de multiples fois. Cependant cet algorithme, appelé avec des entiers *naturels*, termine bien. En effet :
- les arguments de `binome` varient dans $\mathcal{A} = \mathbb{N}^2$. On peut choisir à nouveau $E = \mathcal{A}$ et $\preccurlyeq$ l'ordre lexicographique,
- les deux appels à `binome` intervenant dans sa définition le sont avec les couples $(n-1,p) \prec (n,p)$ et $(n-1,p-1) \prec (n,p)$,
- l'élément minimum $(0,0)$ de $(\mathbb{N}^2, \preccurlyeq)$ est traité dans le cas `(n,0)`.
Attention, l'instruction `(n,0) -> 1` est indispensable. Elle garantit que p (et donc n) reste bien dans $\mathbb{N}$. Si l'on remplaçait ce motif par le simple `(0,0)` alors le calcul de `binome (0,1)` ne terminerait pas. En effet, on parviendrait à des entiers négatifs et $(\mathbb{Z}^2, \preccurlyeq)$, on le sait, n'est pas bien fondé.
Sur cet exemple, on peut également constater que les choix de E et φ ne sont pas nécessairement uniques. En effet, on pourrait aussi conclure en considérant $E = \mathbb{N}$ et $\varphi(n,p) = n+p$ qui décroît elle aussi strictement à chaque appel.
Voyons à présent le cas d'une fonction chère aux informaticiens : la fonction d'Ackermann[5]. Elle est définie comme suit :

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

C'est une fonction à croissance très rapide comme on le montrera en exercice. On peut par exemple calculer :

```
#ackermann 1 1;;
- : int = 3
#ackermann 2 3;;
- : int = 9
#ackermann 3 7;;
- : int = 1021
```

mais vous pouvez attendre longtemps pour calculer et voir s'afficher l'énorme `ackermann 4 4`!
Cette fois-ci, nous ne programmons pas une fonction banale et la preuve de terminaison prend tout son intérêt. Nous allons encore utiliser l'ordre lexicographique sur $\mathbb{N}^2$. On sait que le calcul de `ackermann n p` termine si `n=0` (à nouveau les motifs `0 p` et `n 0` garantissent le maintien dans $\mathbb{N}^2$). Sinon il fait directement appel à (`ackermann (n-1) 1`) et $(n-1,1) \prec (n,p)$ ou à (`ackermann n (p-1)`) et $(n,p-1) \prec (n,p)$ ou à (`ackermann (n-1) X`) et $(n-1,X) \prec (n,p)$ (où `X = ackermann n (p-1)`). D'après le théorème de terminaison, la fonction termine donc bien pour tout couple d'arguments entiers naturels.
En outre, on remarque que, pour cet exemple, les applications à valeurs dans $(\mathbb{N}, \leqslant)$ que sont $\varphi(n,p) = n+p$ ou $\varphi(n,p) = \max(n,p)$... ne permettent pas de conclure.
Attention Caml évalue ses arguments avant de les appeler :
Voici un cas d'école : considérons la fonction dite "fonction de Morris" définie par :

```
let rec morris = fun
    | 0 _ -> 1
    |_m n  -> morris (m-1) (morris m n);;_
```

Une preuve trop rapide de terminaison de cette fonction conduirait à écrire :
- on considère l'ordre lexicographique sur $\mathbb{N}^2$,
- le calcul de `morris m n` fait appel au calcul de `morris (m-1) X` et $(m-1,X) \prec (m,n)$ pour tout `X`,
- enfin, la fonction termine lorsque son premier argument est nul.
Mais `X = morris m n`, ce qui entraîne que, par exemple, l'exécution en Caml de `morris 1 0` ne termine pas! En effet, ce calcul conduit à l'exécution de `morris 0 (morris 1 0)` et donc à nouveau au calcul de `morris 1 0` et ainsi de suite. En effet, comme dans la plupart des langages de programmation, en Caml, les arguments sont évalués avant d'être passés à la fonction (on parle "d'appel par valeurs").

[5]La fonction d'Ackermann constitue un exemple de fonction récursive "non primitive", cette notion étant fondamentale en théorie de la calculabilité.

4.3 Preuves par induction structurelle

Cette fois-ci, c'est sur les *types de données* que nous allons faire des preuves.
Nous allons pour cela nous intéresser à une catégorie d'ensembles (bien fondés) qui vont modéliser beaucoup de types de données : les ensembles définis inductivement[6].
Succinctement, un ensemble inductif $\mathcal{T}$ est défini par la donnée d'éléments de base et de constructeurs (permettant à partir d'éléments de $\mathcal{T}$ d'en obtenir d'autres). Par exemple, l'ensemble des arbres binaires est un ensemble inductif. C'est, en effet, la donnée d'éléments de base (les feuilles) et d'un constructeur (`Noeud`) qui permet d'obtenir tous les arbres.
Avant de donner la définition générale, rappelons que l'on appelle *arité* d'une application *le nombre d'arguments sur lesquels elle opère.*

Définition 4.3.1 *Soit E un ensemble, $\mathcal{B}$ une partie de E (les éléments de base) et $\mathcal{C}$ un ensemble d'applications f (les constructeurs) définies de $E^{a(f)}$ dans E (où $a(f)$ désigne l'arité de f).*
On appelle ensemble défini inductivement à l'aide de E, $\mathcal{B}$ et $\mathcal{C}$ *le plus petit sous-ensemble $\mathcal{T}$ de E (au sens de l'inclusion) tel que :*
- (a) - $\mathcal{B} \subset \mathcal{T}$,
- (b) - $\forall\ f \in \mathcal{C}$, $\forall\ (t_1, \ldots, t_{a(f)}) \in \mathcal{T}^{a(f)}$, $f(t_1, \ldots, t_{a(f)}) \in \mathcal{T}$.

Les éléments de $\mathcal{T}$ sont habituellement appelés des *termes*. Les expressions arithmétiques constituent un premier exemple d'ensemble inductif. Les éléments de base sont les lettres (les variables) et les entiers (les constantes). Les constructeurs sont les trois applications '`+`' et '`*`' d'arité 2 et '`sin`' d'arité 1. Vous remarquerez *l'analogie* entre *ensemble inductif* et *type récursif* en Caml.
Les listes forment un autre exemple d'ensemble inductif, un peu moins simple qu'il n'y paraît. La structure de liste d'éléments d'un ensemble $\mathcal{A}$ est usuellement notée ainsi :

$$(1) \qquad Liste = nil \quad + \quad \acute{e}l\acute{e}ment \times Liste$$

ce qui signifie : une liste est soit la liste vide (classiquement notée *nil*), soit un élément de $\mathcal{A}$ suivi d'une liste. Vous remarquez que le $\times$ de (1) n'est pas exactement un constructeur d'arité 2 au sens de la définition d'un ensemble inductif, puisque ses deux arguments n'ont pas le même type. Il convient donc de définir autant de constructeurs que d'éléments de $\mathcal{A}$. On dispose donc d'un élément de base *nil* et potentiellement d'une infinité de constructeurs d'arité 1 les $conse_a$, pour $a \in \mathcal{A}$. L'objet Caml `[1;2;4]` est ainsi l'élément $conse_1(conse_2(conse_4(nil)))$ de l'ensemble inductif des listes d'entiers.
Les arbres binaires *étiquetés* sont également formalisés à l'aide d'une infinité de constructeurs.
Enfin, l'ensemble inductif qui vous est sans doute le plus familier est l'ensemble des entiers naturels $\mathbb{N}$. Pour vous en convaincre, reprenons sa définition à l'aide des axiomes de Peano. Les voici, présentés selon nos besoins :

- 0 est un entier naturel,
- il existe une application *succ* (pour "successeur") injective de $\mathbb{N}$ dans $\mathbb{N}$,
- $0 \notin succ < \mathbb{N} >$,
- toute partie M de $\mathbb{N}$ contenant 0 et stable par *succ* est égale à $\mathbb{N}$.

L'usage veut que l'on note n l'entier $succ(succ(\ldots succ(succ(0))\ldots))$ (où *succ* est appliquée n fois). Une fois l'addition définie, *succ* devient bien sûr l'application qui à n associe $n+1$.
L'ensemble des entiers naturels est donc un ensemble inductif avec 0 pour seul élément de base et *succ* pour seul constructeur. Remarquez qu'en modifiant la fonction *succ* en $succ_2 : n \longmapsto n+2$, on définit inductivement $2\mathbb{N}$...
On peut caractériser un ensemble inductif en le construisant "par étapes" :

Lemme 1. *Avec les notations précédentes, on définit la suite $(\mathcal{T}_i)_{i\in\mathbb{N}}$ de parties de E par :*
- $\mathcal{T}_0 = \mathcal{B}$,
- $\mathcal{T}_{n+1} = \mathcal{T}_n \ \cup \ \bigcup_{f\in\mathcal{C}} \{f(t_1, \ldots, t_{a(f)}) \mid (t_1, \ldots, t_{a(f)}) \in {\mathcal{T}_n}^{a(f)}\}$.

On a alors $\forall\ t \in \mathcal{T}$, $\exists\ n \in \mathbb{N}$, $t \in \mathcal{T}_n$.

Preuve. On note $\mathcal{X} = \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$. Le lemme devient alors : $\mathcal{T} \subset \mathcal{X}$.

Pour montrer ce résultat, $\mathcal{T}$ étant minimal au sens de l'inclusion, il suffit de montrer que $\mathcal{X}$ vérifie les propriétés (a) et (b).
- On a par construction $\mathcal{B} = \mathcal{T}_0 \subset \mathcal{X}$.

[6] expression que nous abrégerons par la suite en "ensembles inductifs".

- Soit $f \in \mathcal{C}$ et $a(f)$ éléments de $\mathcal{X}$ quelconques $t_1, t_2, \ldots, t_{a(f)}$.
Pour tout $i \in [\![1, a(f)]\!]$, $\exists\ n_i \in \mathbb{N}$ tel que $t_i \in \mathcal{T}_{n_i}$. La suite $(\mathcal{T}_n)_{n\in\mathbb{N}}$ est croissante au sens de l'inclusion (c'est-à-dire $\forall\ n \in \mathbb{N}$, $\mathcal{T}_n \subset \mathcal{T}_{n+1}$) donc, en notant $N = \max\{n_i, i \in [\![1, a(f)]\!]\}$, on a $\forall\ i \in [\![1, a(f)]\!]$, $t_i \in \mathcal{T}_N$. Ainsi, $f(t_1, \ldots, t_{a(f)})$ appartient par construction à $\mathcal{T}_{N+1}$ et donc à $\mathcal{X}$.
Avec les notations précédentes, on a $\mathcal{X} \in \mathcal{F}$, soit $\mathcal{T} \subset \mathcal{X}$. □

Le lemme précédent permet de donner la définition suivante :

Définition 4.3.2 *Pour $t \in \mathcal{T}$, l'entier $n_0 = \min\{n \in \mathbb{N} \mid t \in \mathcal{T}_n\}$ est appelé la* complexité *du terme t.*

Dans les ensembles inductifs que nous avons vus, cette complexité s'interprète comme :
- la longueur d'une liste (c'est en effet le nombre de *conses*),
- la hauteur d'un arbre binaire,
- la *profondeur* d'une expression arithmétique (c'est-à-dire la hauteur de sa représentation arborescente),
- la valeur d'un entier (c'est le nombre de *succ*).

Par construction, $\mathcal{T}_n$ est la partie constituée des éléments de $\mathcal{T}$ de complexité inférieure ou égale à n. On peut à présent établir la propriété augurée par le lemme, c'est-à-dire l'égalité entre un ensemble inductif et la réunion de ses termes classés suivant leur complexité.

Proposition 4.3.3 *Avec les notations précédentes, on a $\mathcal{T} = \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$.*

Preuve. En notant à nouveau $\mathcal{X} = \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$, on sait grâce au lemme que $\mathcal{T} \subset \mathcal{X}$. Pour établir l'inclusion inverse, on va montrer, par récurrence sur n, que $\forall\ n \in \mathbb{N}$, $\mathcal{T}_n \subset \mathcal{T}$.
- On a bien $\mathcal{B} = \mathcal{T}_0 \subset \mathcal{T}$.
- Si l'on suppose que $\mathcal{T}_n \subset \mathcal{T}$ alors, comme $\mathcal{T}$ vérifie la propriété (b), $\forall\ f \in \mathcal{C}$, $\forall\ (t_1, \ldots, t_{a(f)}) \in {\mathcal{T}_n}^{a(f)}$ on a $f(t_1, \ldots, t_{a(f)}) \in \mathcal{T}$. Ainsi
$\left(\mathcal{T}_n\ \cup\ \bigcup_{f\in\mathcal{C}} \{f(t_1, \ldots, t_{a(f)}) \mid (t_1, \ldots, t_{a(f)}) \in {\mathcal{T}_n}^{a(f)}\}\right) \subset \mathcal{T}$ soit $\mathcal{T}_{n+1} \subset \mathcal{T}$.
On a donc bien l'égalité $\mathcal{X} = \mathcal{T}$ □

La représentation arborescente d'un terme d'un ensemble inductif permet de visualiser facilement la notion de *sous-terme* d'un terme. Intuitivement cela revient à considérer le terme représenté par les branches issues d'un noeud de la représentation arborescente du terme initial.
On peut donner une définition inductive de cette notion :

Définition 4.3.4 *Soit $\mathcal{T}$ un ensemble défini inductivement à partir de $(E, \mathcal{B}, \mathcal{C})$ et t un terme de $\mathcal{T}$. L'ensemble $S(t)$ des sous-termes de t est défini inductivement par :*
- si $t \in \mathcal{B}$, alors $S(t) = \{t\}$,
- si $t = f(t_1, \ldots, t_{a(f)})$, $S(t) = \{t\}\ \cup\ \bigcup_{i=1}^{a(f)} S(t_i)$.
Pour $(t, t') \in \mathcal{T}^2$, on note "$t'$ est un sous-terme de t" par $t' \preccurlyeq t$.

La notation $\preccurlyeq$ n'est pas fortuite :

Proposition 4.3.5 *La relation $\preccurlyeq$ est un ordre bien fondé sur $\mathcal{T}$.*

Preuve. Il s'agit clairement d'une relation d'ordre partiel[7]. Nous allons montrer qu'il n'existe pas de suite infinie d'éléments de $\mathcal{T}$ strictement décroissante.
Il suffit de remarquer que si $t' \prec t$ (*i.e.* si t' est un sous-terme strict de t) alors la complexité de t' est strictement inférieure à celle de t. L'existence d'une suite infinie strictement décroissante de termes conduirait donc à l'existence d'une suite d'entiers naturels strictement décroissante, ce qui est exclu. □

On remarque que l'ensemble des éléments minimaux de $\mathcal{T}$ est exactement $\mathcal{B}$.
Il résulte de la proposition précédente que, sur un ensemble de termes défini inductivement, on dispose des méthodes de preuve en vigueur dans les ensembles bien fondés.

[7] Pour l'antisymétrie, il suffit de montrer que $t' \preccurlyeq t$ et complexité(t) = complexité(t') entraîne $t = t'$.

Théorème 4.3.6 (Induction structurelle)

Soit $\mathcal{T}$ un ensemble défini inductivement à partir de $(E, \mathcal{B}, \mathcal{C})$ et $\mathfrak{p}$ un prédicat sur $\mathcal{T}$; si on a :
- $\forall\ b \in \mathcal{B}$, $\mathfrak{p}(b)$ est vrai,
- $\forall\ f \in \mathcal{C}$, $\forall\ (t_1, \dots, t_{a(f)}) \in \mathcal{T}^{a(f)}$:

$$\Big(\mathfrak{p}(t_1) \text{ et } \mathfrak{p}(t_2) \text{ et } \dots \text{ et } \mathfrak{p}(t_{a(f)})\Big) \quad \Rightarrow \quad \mathfrak{p}(f(t_1, \dots, t_{a(f)}))$$

alors $\forall\ t \in \mathcal{T}$, $\mathfrak{p}(t)$ est vrai.
Cette méthode de preuve est usuellement appelée preuve par induction structurelle.

Preuve. Sachant qu'un ensemble inductif est bien fondé, on peut directement utiliser le théorème d'induction. Ou bien, on peut refaire ici une preuve :
On considère $\mathcal{V} = \{t \in \mathcal{T} \mid \mathfrak{p}(t) \text{est vrai}\}$. On a, d'après la première hypothèse, $\mathcal{B} \subset \mathcal{V}$ et d'après la seconde il est évident que $\mathcal{V}$ vérifie la propriété (b) de la définition d'un ensemble inductif et donc $\mathcal{V} \in \mathcal{F}$ soit $\mathcal{T} \subset \mathcal{V}$. L'inclusion inverse est triviale. □

Sur l'ensemble inductif $\mathbb{N}$, le théorème d'induction structurelle devient :
- si $\mathfrak{p}(0)$ est vrai
- et si, pour tout n, $\mathfrak{p}(n) \Rightarrow \mathfrak{p}(succ(n))$
alors $\forall\ n \in \mathbb{N}$, $\mathfrak{p}(n)$.
On reconnaît la classique propriété de récurrence.
Reprenons l'ensemble inductif $\mathcal{AB}$ des arbres binaires :

```
type arbre_binaire = Feuille of int
                   | Noeud of  arbre_binaire * arbre_binaire;;
```

Pour $t \in \mathcal{AB}$, on note n_t (resp. f_t) le nombre de constructeurs `Noeud` (resp. `Feuille`) intervenant dans sa structure.
Montrons par induction structurelle la propriété suivante : "$f_t = n_t + 1$".
- Les éléments de base de l'ensemble inductif sont les arbres réduits à une feuille pour lesquels la propriété est clairement vérifiée.
- Soit $t =$ `Noeud`(t_1, t_2) un arbre binaire de hauteur supérieure ou égale à 1. On a $n_t = n_{t_1} + n_{t_2} + 1$ et $f_t = f_{t_1} + f_{t_2}$.
Par hypothèse d'induction, $f_{t_1} = n_{t_1} + 1$ et $f_{t_2} = n_{t_2} + 1$ d'où $f_t = n_{t_1} + n_{t_2} + 2 = n_t + 1$.
La propriété est donc vérifiée par induction structurelle sur $\mathcal{AB}$.
Nous avons dit que l'ensemble inductif des listes contient potentiellement une infinité de constructeurs. Sur les listes d'entiers, par exemple, on remplace, par commodité, dans les preuves par induction structurelle, la partie "$\forall\ conse_n \in \mathcal{C}$, $\mathfrak{p}(conse_n(\dots))$" par "$\forall\ n \in \mathbb{N}$, $\mathfrak{p}(conse\ n\ \ \dots)$" (ou encore, en Caml, $\mathfrak{p}$(`n::...`)).
Il en est de même avec les arbres étiquetés et plus généralement avec les ensembles inductifs dont les constructeurs sont indexés par un ensemble infini.
Vous comparerez avec profit les preuves effectuées sur des fonctions récursives maniant des listes et les preuves que l'on rédige à présent en considérant le caractère inductif de l'ensemble des listes.
Pour finir, signalons qu'une fonction est particulièrement simple à définir sur un ensemble inductif. Il suffit de la définir sur les éléments de base puis inductivement sur les nouveaux éléments construits à partir d'éléments déjà connus. C'est à nouveau une conséquence du théorème d'induction structurelle :

Proposition 4.3.7 (Définition et unicité) *Soit $\mathcal{T}$ un ensemble défini inductivement à partir de $(E, \mathcal{B}, \mathcal{C})$. Soient g et φ deux applications de $\mathcal{T}$ dans un ensemble quelconque. Si :*
- $\forall\ b \in \mathcal{B}$, $\varphi(b) = g(b)$,
- $\forall\ f \in \mathcal{C}$, $\forall\ (t_1, \dots, t_{a(f)}) \in \mathcal{T}^{a(f)}$ en notant $t = f(t_1, \dots, t_{a(f)})$:

$$\Big(g(t_1) = \varphi(t_1) \text{ et } \dots \text{ et } g(t_{a(f)}) = \varphi(t_{a(f)})\Big) \quad \Rightarrow \quad g(t) = \varphi(t)$$

alors $g = \varphi$.
Autrement dit, les définitions de φ sur $\mathcal{B}$ et relativement aux constructeurs de $\mathcal{C}$ suffisent à caractériser de façon unique φ sur $\mathcal{T}$.

Preuve. Si g et φ sont toutes les deux considérées comme connues, le prédicat $g(t) = \varphi(t)$ se montre aussitôt sur $\mathcal{T}$ par induction structurelle.
Si l'on considère que g est une fonction donnée et que φ est une fonction à définir sur $\mathcal{T}$, le prédicat "$\varphi(t)$ est défini" est lui aussi immédiatement prouvé sur $\mathcal{T}$ par induction structurelle. □

Avec cette proposition, on dispose donc à la fois d'une propriété permettant de définir commodément une application sur un ensemble inductif et d'une caractérisation de l'égalité entre deux applications définies sur un ensemble inductif. Toutes les fonctions récursives vues en Sup sur les listes d'entiers ont été définies sur ce principe (définition sur `[]` puis sur `a::r`, $\forall\, a \in \mathbb{N}$) : c'est le filtrage de Caml ; nous avons ici la confirmation de la non ambiguïté de ces définitions.