

Définitions inductives mathématiques et définitions récursives en OCAML

Définitions inductives en mathématique

- Pour manipuler plusieurs objets de même nature.
- Si le nombre d'objets est grand ou variable, alors les n -uplets ne sont pas commodes.
- Si le nombre d'objets n'est pas connu à l'écriture du programme, alors les n -uplets sont impossibles.

Définitions inductives : caractéristiques

Elles sont **constructives** ou **productives** car on se donne

- des **constantes** du domaine
- des **opérations** de construction

Certaines opérations utilisent des valeurs que l'on sait déjà construire pour produire de nouvelles valeurs.

Les méthodes

- Méthode descendante
- Méthode ascendante
- Équivalence

Définitions inductives (intuition)

Un ensemble \mathcal{Y} est défini récursivement si on se donne

1. un **ensemble de base** \mathcal{X} appartenant à l'ensemble E ;
2. des **opérations de constructions** $f \in F$ telles que si $x_1, \dots, x_n \in \mathcal{X}$ alors $f(x_1, \dots, x_n)$ est également un élément de \mathcal{Y} ;
3. un **principe de minimalité** qui dit que l'ensemble \mathcal{Y} est le plus petit ensemble qui vérifie les deux conditions précédentes. C'est-à-dire que si \mathcal{E} est un ensemble qui contient les valeurs de base \mathcal{X} et qui est stable par les opérations de constructions de F alors \mathcal{Y} est inclus dans \mathcal{E} .

Définitions inductives descendentes

Soit \mathcal{A} un ensemble, $\mathcal{X} \subseteq \mathcal{A}$ un sous-ensemble de \mathcal{A} et F un ensemble de fonctions/opérations $\{f \mid f : A^n \rightarrow A\}$.

1

Définition : Un ensemble \mathcal{Y} est **inductif** sur \mathcal{X} (sous F) ssi

1. $\mathcal{X} \subseteq \mathcal{Y}$
2. \mathcal{Y} est **clos** par F : pour toute fonction $f \in F$ d'arité n , si $x_1, \dots, x_n \in \mathcal{Y}$, alors $f(x_1, \dots, x_n) \in \mathcal{Y}$.

Remarque : Les fonctions/opérations sont **partielles**.

Remarque : \mathcal{A} est inductif sur \mathcal{X} .

Plus petit ensemble inductif

Définition : La **clôture inductive descendante** de \mathcal{X} sous F , notée $CD(\mathcal{X})$, est l'intersection de tous les ensembles inductifs sur \mathcal{X} .

Remarque : $CD(\mathcal{X})$ est clairement un ensemble inductif sur \mathcal{X} et en plus il est le plus petit. Si \mathcal{X} n'est pas vide, alors $CD(\mathcal{X})$ n'est pas vide.

Définitions inductives ascendantes

Définition : La **clôture inductive ascendante** de \mathcal{X} sous F est définie comme $CA(\mathcal{X}) = \bigcup_{i \geq 0} X_i$, où la séquence X_0, X_1, \dots est construite comme suit :

- $X_0 = \mathcal{X}$
- $X_{i+1} = X_i \cup \{f(x_1, \dots, x_n) \mid x_1, \dots, x_n \in X_i\}$

Équivalence

Lemme : $CD(\mathcal{X}) = CA(\mathcal{X})$.

Notation : $\hat{\mathcal{X}} = CD(\mathcal{X}) = CA(\mathcal{X})$.

Types de définitions inductives

- Structures séquentielles : chaînes de caractères, ensembles, listes, suites de valeurs, fichiers, piles, etc
- Structures arborescentes : expressions arithmétiques, expressions d'un langage, arbres binaires, arbres n -aires, formules propositionnelles, arbres généalogiques, etc

Exemple : les entiers naturels

$\mathcal{X} = \{0\}$ et $F = \{\text{plus1} \mid \text{plus1}(n) = n + 1\}$.

La **méthode descendante** :

L'ensemble \mathbb{N} des entiers naturels est le plus petit ensemble t.q.

- $0 \in \mathbb{N}$
- Si $n \in \mathbb{N}$, alors $n + 1 \in \mathbb{N}$.

2

La méthode ascendante :

L'ensemble \mathbb{N} est donné par $\bigcup_{i \geq 0} X_i$, où

- $X_0 = \{0\}$
- $X_1 = \{0, 1\}$
- $X_2 = \{0, 1, 2\}$
- ...
- $X_i = \{0, 1, 2, \dots, i\}$

Exemple : les mots sur un alphabet A

$\mathcal{X} = \{\epsilon\}$ et $F = \{\text{ajoute}_a \mid a \in A, \text{ajoute}_a(m) = am\}$.

La **méthode descendante** :

Les mots sur un alphabet A, dénoté par A^* , est le plus petit ensemble

t.q.

- $\epsilon \in A^*$
- Si $m \in A^*$, alors pour toute lettre $a \in A$ on a $am \in A^*$.

La **méthode ascendante** :

L'ensemble A^* est donné par $\bigcup_{i \geq 0} X_i$, où

- $X_0 = \{\epsilon\}$
- $X_1 = \{\epsilon\} \cup \{a \mid a \in A\}$
- $X_2 = \{\epsilon\} \cup \{a \mid a \in A\} \cup \{a_1 a_2 \mid a_1, a_2 \in A\}$
- ...
- $X_i = \{a_1 a_2 \dots a_n \mid 0 \leq n \leq i, a_1, a_2, \dots, a_n \in A\}$

Exemple : les listes d'éléments d'un type donné

$\mathcal{X} = \{\text{nil}\}$ et $F = \{\text{ajoute}_e \mid \text{ajoute}_e(l) = \text{cons}(e, l) \text{ et } e : E\}$.

La **méthode descendante** :

L'ensemble $\text{liste}(E)$ est le plus petit ensemble t.q.

- $\text{nil} \in \text{liste}(E)$
- Si $l \in \text{liste}(E)$, alors pour tout élément $e : E$ on a $\text{cons}(e, l) \in \text{liste}(E)$.

La **méthode ascendante** :

L'ensemble $\text{liste}(E)$ est donné par $\bigcup_{i \geq 0} X_i$, où

- $X_0 = \{\text{nil}\}$
- $X_1 = \{\text{nil}\} \cup \{\text{cons}(e, \text{nil}) \mid e : E\}$
- $X_2 = \{\text{nil}\} \cup \{\text{cons}(e_1, \text{nil}) \mid e_1 : E\} \cup \{\text{cons}(e_1, \text{cons}(e_2, \text{nil})) \mid e_1, e_2 : E\}$
- ...
- $X_i = \{\text{liste } l \mid 0 \leq |l| \leq i \text{ et tout } e \in l \text{ vérifie } e : E\}$

Exemple : l'ensemble d'expressions avec + et * sur $\{0, 1\}$

$\mathcal{X} = \{^*0^*, ^*1^*\}$ et $F = \{\text{add}(n, m) = n^{**} + ^*m\} \cup \{\text{mul} \mid \text{mul}(n, m) = n^{**} * ^*m\}$.

3

La méthode descendante :

L'ensemble $Exp_{+,*}^{0,1}$ est le plus petit ensemble t.q.

- $\{^*0^*, ^*1^*\} \subseteq Exp_{+,*}^{0,1}$
- Si $n, m \in Exp_{+,*}^{0,1}$ alors $^{*n+m^*}, ^*n*m^* \in Exp_{+,*}^{0,1}$.

La **méthode ascendante** :

L'ensemble $Exp_{+,*}^{0,1}$ est donné par $\bigcup_{i \geq 0} X_i$, où

- $X_0 = \{^*0^*, ^*1^*\}$
- $X_1 = \{^*0^*, ^*1^*\} \cup \{^*0+0^*, ^*0+1^*, ^*1+0^*, ^*1+1^*\} \cup \{^*0*0^*, ^*0*1^*, ^*1*0^*, ^*1*1^*\}$
- $X_2 = X_1 \cup \{^*0+0+0^*, ^*0+0+0+0^*, ^*0+0*0^*, ^*0*1+0^*, ^*0*1*0^*, \dots\}$
- ...
- $X_i = \{\text{expr } e \mid 0 \leq \text{nb_opérateurs}(e) \leq 2^i - 1\}$

Exemple : les arbres binaires sans étiquettes

$\mathcal{X} = \{\text{nil}\}$ et $F = \{\text{arb}(a_1, a_2)\}$.

La **méthode descendante** :

L'ensemble A_{bin} est le plus petit ensemble t.q.

- $\text{nil} \in A_{bin}$
- Si $a_1, a_2 \in A_{bin}$, alors $\text{arb}(a_1, a_2) \in A_{bin}$.

La **méthode ascendante** :

L'ensemble $A_{bin}(E)$ est donné par $\bigcup_{i \geq 0} X_i$, où

- $X_0 = \{\text{nil}\}$
- $X_1 = \{\text{nil}, \text{arb}(\text{nil}, \text{nil})\}$
- $X_2 = \{\text{nil}, \text{arb}(\text{nil}, \text{nil}), \text{arb}(\text{nil}, \text{arb}(\text{nil}, \text{nil})), \text{arb}(\text{arb}(\text{nil}, \text{nil}), \text{nil}) \cup \text{arb}(\text{arb}(\text{nil}, \text{nil}), \text{arb}(\text{nil}, \text{nil}))\}$
- ...
- $X_i = \{\text{arbre } a \mid 1 \leq \text{nb_nil}(a) \leq 2^i\}$

Définitions récursives en OCAML

- Définition d'un **type récursif** par l'utilisateur.
- Définition d'une **fonction récursive** travaillant sur un type récursif.

Exemple : les entiers

```
# type entier = Z | S of entier;;
type entier = Z | S of entier
# Z;;
- : entier = Z
# S(Z);;
- : entier = S Z
# S(S(Z));;
- : entier = S (S Z)
```

4

Les entiers en OCAML

Cette définition n'est pas nécessaire car OCAML fournit un type prédéfini pour les entiers.

```
# 0;;
- : int = 0
# 1;;
- : int = 1
# 2;;
- : int = 2
```

Exemple : les mots sur un alphabet A

```
# type alpha1 = A | B | C | D ;;
type alpha1 = A | B | C | D

# type mots_alpha1 = MotV1 | AjL1 of alpha1 * mots_alpha1;;
type mots_alpha1 = MotV1 | AjL1 of alpha1 * mots_alpha1

# AjL1(A,AjL1(B,MotV1));;
- : mots_alpha1 = AjL1 (A, AjL1 (B, MotV1))

# type alpha2 = E | F | G | H | I ;;
type alpha2 = E | F | G | H | I

# type mots_alpha2 = MotV2 | AjL2 of alpha2 * mots_alpha2 ;;
type mots_alpha2 = MotV2 | AjL2 of alpha2 * mots_alpha2

# AjL2(E,AjL2(F,MotV2));;
- : mots_alpha2 = AjL2 (E, AjL2 (F, MotV2))
```

On remarque que...

- Les deux définitions `mots_alpha1` et `mots_alpha2` sont similaires.
- Un **autres type** représentant les mots sur un **autre alphabet** serait aussi similaire.

5

Mieux encore : les mots sur n'importe quel alphabet

On fera abstraction de l'alphabet pour donner une définition plus **généraliste** (**polymorphe**).

Les mots polymorphes en OCAML

```
# type alpha = A | B | C | D ;;
type alpha = A | B | C | D
# type alpha2 = E | F | G | H | I ;;
type alpha2 = E | F | G | H | I

# type 'a mots = MotV | AjL of 'a * 'a mots ;;
type 'a mots = MotV | AjL of 'a * 'a mots

# AjL(A,AjL(B,MotV));;
- : alpha1 mots = AjL(A, AjL(B, MotV))
# AjL(E,AjL(F,MotV));;
- : alpha2 mots = AjL(E, AjL(F, MotV))

(* opération de construction *)
# let cons_mot(e,s) = AjL(e,s);;
val cons_mot : 'a * 'a mots -> 'a mots = <fun>

(* opérations d'accès *)
# let premier_car(m) = match m with
  MotV -> failwith "erreur"
| AjL(e,s) -> e ;;
val premier_car : 'a mots -> 'a = <fun>

# let reste_mot(m) = match m with
  MotV -> failwith "erreur"
| AjL(e,s) -> s ;;
val reste_mot : 'a mots -> 'a mots = <fun>

(* opération de test *)
# let est_mot_vide(m) = match m with
  MotV -> true
```

6

```
|_ -> false;;
val est_mot_vide : 'a mots -> bool = <fun>
```

Les chaînes de caractères en OCAML

Si l'alphabet en question contient tous les symboles informatiques *possibles*, alors cette définition n'est pas nécessaire car OCAML fournit un type prédéfini pour les chaînes de caractères sur cet alphabet.

```
# "AB";;
- : string = "AB"
# "EF";;
- : string = "EF"
# "$$$###";;
- : string = "$$$###"
# "\E\E\E";;
- : string = "\200\201\202"

(* comme premier_car *)
# String.get "abc" 0 ;;
- : char = 'a'

(* comme rste_mot *)
# String.sub "abc" 1 2 ;;
- : string = "bc"
```

Exemple : les listes d'éléments d'un type donné

```
# type liste_ent = LeV | ConsE of int * liste_ent ;;
type liste_ent = LeV | ConsE of int * liste_ent

# ConsE(3,ConsE(4,LeV));;
- : liste_ent = ConsE (3, ConsE (4, LeV))

# type liste_flot = LfV | ConsF of float * liste_flot;;
type liste_flot = LfV | ConsF of float * liste_flot

# ConsF(3.3,ConsF(5.1,LfV));;
- : liste_flot = ConsF (3.3, ConsF (5.1, LfV))
```

7

Mieux encore : les listes polymorphes

```
# type 'a liste = LV | Cons of 'a * 'a liste ;;
type 'a liste = LV | Cons of 'a * 'a liste

# Cons(3, Cons(5,LV));;
- : int liste = Cons (3, Cons (5, LV))

# Cons(3, Cons(5.4, LV));;
This expression has type float but is here used with type int

# Cons(true, Cons(false, LV));;
- : bool liste = Cons (true, Cons (false, LV))

(* opération de construction *)
# let cons_liste(e,s) = Cons(e,s);;
val cons_liste : 'a * 'a liste -> 'a liste = <fun>

(* opérations d'accès *)
# let premier_liste(l) = match l with
  LV -> failwith "erreur"
| Cons(e,s) -> e ;;
val premier_liste : 'a liste -> 'a = <fun>

# let reste_liste(l) = match l with
  LV -> failwith "erreur"
| Cons(e,s) -> s ;;
val reste_liste : 'a liste -> 'a liste = <fun>

(* opération de test *)
# let est_liste_vide(l) = match l with
  LV -> true
| _ -> false;;
val est_liste_vide : 'a liste -> bool = <fun>
```

Les listes polymorphes en OCAML

Cette définition n'est pas nécessaire car OCAML fournit un type prédéfini pour les listes **polymorphes**.

8

```

# [];;
- : 'a list = []

# 1::[];;
- : int list = [1]

# 1:: 2 :: 3 ::[];;
- : int list = [1; 2; 3]

(* D'autres listes *)
# "juste" :: "quelques" :: "mots" :: [];;
- : string list = ["juste"; "quelques"; "mots"]

# [true; false; true];;
- : bool list = [true; false; true]

# [ (3.2,4.5); (3.1,5.5) ];;
- : (float * float) list = [(3.2, 4.5); (3.1, 5.5)]

Les listes de listes ...
# [ []; [1]; [1; 2] ];;
- : int list list = [[]; [1]; [1; 2]]

# [ [ []; [1]; [1; 2] ]; [ [ []; [1]; [1; 2] ] ];;
- : int list list list =
  [[[]; [1]; [1; 2]]; [[[]; [1]; [1; 2]]]

Exemple : l'ensemble d'expressions avec + et * sur {0,1}
# type expr = Z | U |
  Plus of expr * expr |
  Mul of expr * expr;;
type expr = Z | U | Plus of expr * expr | Mul of expr * expr

(* opérations de construction *)
# let cons_expr_plus(a,b) = Plus(a,b);;
val cons_expr_plus : expr * expr -> expr = <fun>

```

9

```

# let cons_expr_mul(a,b) = Plus(a,b);;
val cons_expr_mul : expr * expr -> expr = <fun>

```

```

(* opérations d'accès *)
# let expr_gauche(e) = match e with
  Plus(a,b) -> a
| Mul(a,b) -> a
| _ -> failwith "erreur";;
val expr_gauche : expr -> expr = <fun>

# let expr_droite(e) = match e with
  Plus(a,b) -> b
| Mul(a,b) -> b
| _ -> failwith "erreur";;
val expr_droite : expr -> expr = <fun>

```

```

(* opérations de test *)
# let est_Z(e) = match e with
  Z -> true
| _ -> false;;
val est_Z : expr -> bool = <fun>

# let est_U(e) = match e with
  U -> true
| _ -> false;;
val est_U : expr -> bool = <fun>

```

```

# let est_P(e) = match e with
  Plus(_,_) -> true
| _ -> false;;
val est_P : expr -> bool = <fun>

# let est_M(e) = match e with
  Mul(_,_) -> true

```

10

```

| _ -> false;;
val est_M : expr -> bool = <fun>

```

Exemple : les arbres binaires sans étiquettes

```

# type ab = Av | N of ab * ab;;
type ab = Av | N of ab * ab

(* opération de construction *)
# let cons_ab(x,y) = N(x,y);;
val cons_ab : ab * ab -> ab = <fun>

(* opérations d'accès *)
# let arbre_gauche a = match a with
  N(x,y) -> x
| _ -> failwith "erreur";;
val arbre_gauche : ab -> ab = <fun>

# let arbre_droite a = match a with
  N(x,y) -> y
| _ -> failwith "erreur";;
val arbre_droite : ab -> ab = <fun>

(* opération de test *)
# let est_arbre_vide a = match a with
  Av -> true | _ -> false;;
val est_arbre_vide : ab -> bool = <fun>

# let arbl = N(N(Av,Av),Av);;
val arbl : ab = N (N (Av, Av), Av)

# est_arbre_vide arbl;;
- : bool = false

# arbre_gauche arbl ;;
- : ab = N (Av, Av)

# arbre_droite arbl ;;
- : ab = Av

```

11