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 descendente
- Méthode ascendente
- 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 $x_1, \ldots, x_n \in \mathcal{Y}$ alors $f(x_1, \ldots, x_n)$ 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 $\{f \mid f : \mathcal{A}^n \to A\}$.

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 $x_1, \ldots, x_n \in \mathcal{Y}$, alors $f(x_1, \ldots, x_n) \in \mathcal{Y}$.

Remarque : Les fonctions/opérations sont partielles.

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

Plus petit ensemble inductif

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

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

Définitions inductives ascendentes

Définition : La clôture inductive ascendente de $\mathcal{X}$ sous F est définie comme $\mathcal{CA}(\mathcal{X}) = \bigcup_{i\geq 0} X_i$, où la séquence $\mathcal{X}_0, \mathcal{X}_1, \ldots$ est construite comme suit :

- $\mathcal{X}_0 = \mathcal{X}$
- $\mathcal{X}_{i+1} = \mathcal{X}_i \cup \{f(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in \mathcal{X}_i\}$

Équivalence

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

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

Types de définitions inductives

- Structures séquentielles : chaînes de caractères, ensembles, listes, suites de valeurs, fichiers, files, 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 $F = \{plus1 \mid plus1(n) = n + 1\}$.

La méthode descendente :

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

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

La méthode ascendente :

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

- $\mathcal{X}_0 = \{0\}$
- $\mathcal{X}_1 = \{0, 1\}$
- $\mathcal{X}_2 = \{0, 1, 2\}$
- $\ldots$
- $\mathcal{X}_i = \{0, 1, 2, \ldots, i\}$

Exemple : les mots sur un alphabet A

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

La méthode descendente :

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 ascendente :

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

- $\mathcal{X}_0 = \{\epsilon\}$
- $\mathcal{X}_1 = \{\epsilon\} \cup \{a \mid a \in A\}$
- $\mathcal{X}_2 = \{\epsilon\} \cup \{a \mid a \in A\} \cup \{a_1 a_2 \mid a_1, a_2 \in A\}$
- $\ldots$
- $\mathcal{X}_i = \{a_1 a_2 \ldots a_n \mid 0 \leq n \leq i, a_1, a_2, \ldots, a_n \in A\}$

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

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

La méthode descendente :

L'ensemble $liste(E)$ est le plus petit ensemble t.q.

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

La méthode ascendente :

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

- $\mathcal{X}_0 = \{nil\}$
- $\mathcal{X}_1 = \{nil\} \cup \{cons(e, nil) \mid e : E\}$
- $\mathcal{X}_2 = \begin{cases} \{nil\} \cup \{cons(e_1, nil) \mid e_1 : E\} \cup \\ \{cons(e_1, cons(e_2, nil)) \mid e_1 : E, e_2 : E\} \end{cases}$
- $\ldots$
- $\mathcal{X}_i = \{$liste $l \mid 0 \leq |l| \leq i$ et tout $e \in l$ vérifie $e : E\}$

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

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

La méthode descendente :

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

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

La méthode ascendente :

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

- $\mathcal{X}_0 = \{\texttt{"0"}, \texttt{"1"}\}$
- $\mathcal{X}_1 = \{\texttt{"0"}, \texttt{"1"}\} \cup \{\texttt{"0+0"}, \texttt{"0+1"}, \texttt{"1+0"}, \texttt{"1+1"}\} \cup \{\texttt{"0*0"}, \texttt{"0*1"}, \texttt{"1*0"}, \texttt{"1*1"}\}$
- $\mathcal{X}_2 = \mathcal{X}_1 \cup \{\texttt{"0+0+0"}, \texttt{"0+0+0+0"}, \texttt{"0+0*0"}, \texttt{"0*1+0"}, \texttt{"0+1*0"}, \ldots\}$
- $\ldots$
- $\mathcal{X}_i = \{$ expr $e \mid 0 \leq \texttt{nb\_operateurs}(e) \leq 2^i - 1\}$

Exemple : les arbres binaires sans étiquettes

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

La méthode descendente :

L'ensemble $Abin$ est le plus petit ensemble t.q.

- $nil \in Abin$
- Si $a_1, a_2 \in Abin$, alors $arb(a_1, a_2) \in Abin$.

La méthode ascendente :

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

- $\mathcal{X}_0 = \{nil\}$
- $\mathcal{X}_1 = \{nil, arb(nil, nil)\}$
- $\mathcal{X}_2 = \{nil, arb(nil, nil), arb(nil, arb(nil, nil)),$
  $arb(arb(nil, nil), nil) \cup arb(arb(nil, nil), arb(nil, nil))\}$
- $\ldots$
- $\mathcal{X}_i = \{$ arbre $a \mid 1 \leq$ `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)
```

Les entiers en OCAML

Cette définition n'est pas nécessaire car OCAML fournit un type prédéfinit 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.

Mieux encore : les mots sur n'importe quel alphabet

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

Les mots polymorphes en OCAML

```
# type alpha1 = A | B | C | D ;;
type alpha1 = 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
    |_    -> 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 prédéfinit 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 doné

```
# 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))
```

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 prédéfinit pour les listes polymorphes.

```
# [];;
- : '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>

# 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
     | _        -> 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 arb1 = N(N(Av,Av),Av);;
val arb1 : ab = N (N (Av, Av), Av)
```

```
# est_arbre_vide arb1;;
- : bool = false

# arbre_gauche arb1 ;;
- : ab = N (Av, Av)

# arbre_droite arb1 ;;
- : ab = Av
```