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.

Définitions inductives en informatique

- Syntaxe concrete
- Syntaxe abstraite
- Règles de typage
- Règles d'évaluation

Le principe

Une définition inductive est caracterisée par :

- Une ou plusieures assertions
- Un ensemble de règles d'inférence pour dériver ces assertions

Exemple :

- Assertion : "`X est naturel`" ou "`X nat`"
- Règles d'inférence :

  R1 : `0 est naturel`

  R2 : Si `n est naturel`, alors `succ(n) est naturel`.

Notation

Les règles d'inférence sont notées

$$\frac{\texttt{Hypothèse}_1 \ldots \texttt{Hypothèse}_n}{\texttt{Conclusion}} \text{ (Nom de la règle)}$$

- `Conclusion` est une assertion
- $\texttt{Hypothèse}_1 \ldots \texttt{Hypothèse}_n$ sont des assertions
- En général $n \geq 0$. Si $n = 0$ la règle est un axiome

Exemple (règle unaire)

Les entiers naturels

$$\frac{}{0 \texttt{ est naturel}} (Nat0) \qquad \frac{n \texttt{ est naturel}}{succ(n) \texttt{ est naturel}} (Nat+)$$

Exemple (règle binaire)

Les arbres binaires

$$\frac{}{vide \texttt{ est un arbre binaire}} \text{ (Abin-nil)}$$

$$\frac{A_1 \texttt{ est un arbre binaire} \qquad A_2 \texttt{ est un arbre binaire}}{node(A_1, A_2) \texttt{ est un arbre binaire}} \text{ (Abin-ind)}$$

Exemple

Les mots sur un alphabet A

$$\frac{}{\epsilon \texttt{ mot}} \qquad \frac{a \in A \quad n \texttt{ mot}}{a.n \texttt{ mot}}$$

Exemple (plusieurs axiomes, règles unaires et binaires)

Les expressions de la logique propositionnelle sur l'alphabet A

$$\frac{p \in A}{p \texttt{ expr}}$$

$$\frac{A_1 \texttt{ expr} \qquad A_2 \texttt{ expr}}{A_1 \vee A_2 \texttt{ expr}} \qquad \frac{A_1 \texttt{ expr} \qquad A_2 \texttt{ expr}}{A_1 \wedge A_2 \texttt{ expr}}$$

$$\frac{A_1 \texttt{ expr} \qquad A_2 \texttt{ expr}}{A_1 \rightarrow A_2 \texttt{ expr}} \qquad \frac{A \texttt{ expr}}{\neg A \texttt{ expr}}$$

Exemple (plusieures assertions)

Les forêts

$$\frac{}{vide \texttt{ arbre}} \qquad \frac{}{nil \texttt{ foret}}$$

$$\frac{A \texttt{ arbre} \qquad f \texttt{ foret}}{cons(A, f) \texttt{ foret}} \qquad \frac{f \texttt{ foret}}{node(f) \texttt{ arbre}}$$

Dérivation d'une assertion

Une assertion A est dérivable ssi

– A est un axiome

$$\frac{}{A}$$

– ou il y a une règle de la forme

$$\frac{A_1 \qquad A_n}{A}$$

telle que $A_1, \ldots, A_n$ sont dérivables

Ensemble inductif

Un ensemble inductif est le plus petit ensemble engendré par un système de règles d'inférence.

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

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
```