Leçon 5 : Induction et Programmation Fonctionnelle

Types inductifs pour les données, fonctions récursives, principes d'induction

Pablo Donato

Logique et Fondements de l'Informatique

Année 2026

Rappel : Inductive pour les propositions

La semaine dernière, on a levé le voile sur les connecteurs logiques :

```
Inductive and (A B : Prop)
    : Prop :=
  | conj : A -> B -> A /\ B.
```

```
Inductive or (A B : Prop)
    : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B.
```

- Constructeurs = règles d'introduction en déduction naturelle
- Éliminateurs (`and_ind`, `or_ind`) dérivés automatiquement
- `destruct` = application de l'éliminateur

Rappel : Inductive pour les propositions

La semaine dernière, on a levé le voile sur les connecteurs logiques :

```
Inductive and (A B : Prop)
    : Prop :=
  | conj : A -> B -> A /\ B.
```

```
Inductive or (A B : Prop)
    : Prop :=
  | or_introl : A -> A \/ B
  | or_intror : B -> A \/ B.
```

- Constructeurs = règles d'introduction en déduction naturelle
- Éliminateurs (`and_ind`, `or_ind`) dérivés automatiquement
- `destruct` = application de l'éliminateur

Question

Et si on appliquait le même mécanisme à des **données** plutôt qu'à des *propositions* ?

Les entiers naturels à la Peano

Informellement :

- 0 est un entier naturel
- Si n est un entier naturel, son successeur $S(n)$ aussi
- Ce sont les *seuls* entiers naturels

Univers : `Set` (données calculables), pas `Prop` (preuves).

En Rocq :

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

Notation	Interne
0	`O`
1	`S O`
2	`S (S O)`
n	n applications de `S`

Les entiers naturels à la Peano

Informellement :

- 0 est un entier naturel
- Si n est un entier naturel, son successeur $S(n)$ aussi
- Ce sont les *seuls* entiers naturels

Univers : `Set` (données calculables), pas `Prop` (preuves).

En Rocq :

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

Notation	Interne
0	`O`
1	`S O`
2	`S (S O)`
n	n applications de `S`

Comparez avec `or`

`or` a deux constructeurs (`or_introl`, `or_intror`). `nat` a deux constructeurs (`O`, `S`). Le mécanisme est identique.

Calculer avec nat : le filtrage de motifs

Pour utiliser une valeur de type nat, on filtre par motif sur ses constructeurs, exactement comme on filtrait une preuve de `A \/ B` en Leçon 4.

```
Definition pred (n : nat) : nat :=
  match n with
  | O   => O        (* prédécesseur de O : convention *)
  | S m => m        (* prédécesseur de S m : c'est m  *)
  end.
```

Calculer avec nat : le filtrage de motifs

Pour utiliser une valeur de type nat, on filtre par motif sur ses constructeurs, exactement comme on filtrait une preuve de `A \/ B` en Leçon 4.

```
Definition pred (n : nat) : nat :=
  match n with
  | O   => O        (* prédécesseur de 0 : convention *)
  | S m => m        (* prédécesseur de S m : c'est m  *)
  end.
```

- Deux branches = deux constructeurs de nat
- Dans la branche `S m`, la variable `m : nat` est liée (comme `a : A` dans `or_introl a`)

```
Compute pred 3.   (* = 2 *)
Compute pred 0.   (* = 0 *)
```

Fixpoint : fonctions récursives

Certaines fonctions ont besoin de s'appeler elles-mêmes. Exemple : l'addition.

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O   => m
  | S p => S (add p m)
  end.
```

Fixpoint : fonctions récursives

Certaines fonctions ont besoin de s'appeler elles-mêmes. Exemple : l'addition.

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O   => m
  | S p => S (add p m)
  end.
```

Récursion structurelle

À chaque appel récursif, `p` est structurellement plus petit que `S p`. Rocq vérifie cela — tout `Fixpoint` termine.

Trace de `add 2 3` :

```
  add (S (S O)) 3
= S (add (S O) 3)
= S (S (add O 3))
= S (S 3)
= 5
```

Fixpoint : fonctions récursives

Certaines fonctions ont besoin de s'appeler elles-mêmes. Exemple : l'addition.

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O   => m
  | S p => S (add p m)
  end.
```

Trace de `add 2 3` **:**

```
  add (S (S O)) 3
= S (add (S O) 3)
= S (S (add O 3))
= S (S 3)
= 5
```

Récursion structurelle

À chaque appel récursif, `p` est structurellement plus petit que `S p`. Rocq vérifie cela — tout `Fixpoint` termine.

Pourquoi « `Fixpoint` » ? Le lien avec le combinateur Y

Une définition récursive est un point fixe

Définir `add` par `add n m := ... add ...` revient à chercher `add` tel que $\texttt{add} = F\ \texttt{add}$ (F capture le corps) — `add` est un **point fixe** de F.

Pourquoi « `Fixpoint` » ? Le lien avec le combinateur Y

Une définition récursive est un point fixe

Définir `add` par `add n m := ... add ...` revient à chercher `add` tel que $\texttt{add} = F\ \texttt{add}$ (F capture le corps) — `add` est un **point fixe** de F.

- En λ-calcul non typé, c'est le rôle du combinateur $\mathbf{Y}$: $\mathrm{Y}\,F =_\beta F\,(\mathrm{Y}\,F)$. Le mot-clé `Fixpoint` tire son nom de là.

Pourquoi « `Fixpoint` » ? Le lien avec le combinateur Y

Une définition récursive est un point fixe

Définir `add` par `add n m := ... add ...` revient à chercher `add` tel que $\texttt{add} = F\,\texttt{add}$ (F capture le corps) — `add` est un **point fixe** de F.

- En λ-calcul non typé, c'est le rôle du combinateur $\mathbf{Y}$: $\mathrm{Y}\,F =_\beta F\,(\mathrm{Y}\,F)$. Le mot-clé `Fixpoint` tire son nom de là.
- Mais **Y ne peut pas être un terme ordinaire dans Rocq** : son typage exigerait $\forall A,\ (A \to A) \to A$. Or $A \to A$ est habité par `fun x => x` : on habiterait `False` !

```
Axiom Y_type : forall A, (A -> A) -> A.
Check Y_type False (fun x => x).   (* : False *)
```

Pourquoi « `Fixpoint` » ? Le lien avec le combinateur Y

Une définition récursive est un point fixe

Définir `add` par `add n m := ... add ...` revient à chercher `add` tel que $\texttt{add} = F\,\texttt{add}$ (F capture le corps) — `add` est un **point fixe** de F.

- En λ-calcul non typé, c'est le rôle du combinateur $\mathbf{Y}$: $\mathrm{Y}\,F =_\beta F\,(\mathrm{Y}\,F)$. Le mot-clé `Fixpoint` tire son nom de là.
- Mais **Y ne peut pas être un terme ordinaire dans Rocq** : son typage exigerait $\forall A,\ (A \to A) \to A$. Or $A \to A$ est habité par `fun x => x` : on habiterait `False` !

```
Axiom Y_type : forall A, (A -> A) -> A.
Check Y_type False (fun x => x).   (* : False *)
```

`Fixpoint` = Y restreint au noyau

Primitive du noyau : construit un point fixe après vérification de la récursion structurelle, sans admettre l'axiome dangereux.

Curry-Howard : récursion = induction

Preuve par induction	**Fonction récursive (`Fixpoint`)**
Cas de base : montrer $P(0)$	Cas de base : calculer `f 0`
Cas inductif : déduire $P(p+1)$ en supposant $P(p)$	Cas récursif : calculer `f (S p)` à partir de `f p`
Hypothèse d'induction : $P(p)$	Appel récursif : `f p`
Conclusion : preuve de $\forall n.P(n)$	Valeur de retour : terme de type `T`

Curry-Howard : récursion = induction

Preuve par induction	**Fonction récursive (`Fixpoint`)**
Cas de base : montrer $P(0)$	Cas de base : calculer `f 0`
Cas inductif : déduire $P(p+1)$ en supposant $P(p)$	Cas récursif : calculer `f (S p)` à partir de `f p`
Hypothèse d'induction : $P(p)$	Appel récursif : `f p`
Conclusion : preuve de $\forall n.P(n)$	Valeur de retour : terme de type `T`

Le principe de Curry-Howard

Un `Fixpoint` sur `nat` **est** une preuve par induction sur `nat`. Seule différence : la conclusion est un type de *données* (e.g. `nat : Set`) et non une *proposition* (`forall n, P n : Prop`).

Les listes : un type inductif paramétré

Même structure que `nat`, mais avec un paramètre de type et une donnée dans le constructeur récursif :

```
Inductive list (A : Type) : Type :=
  | nil  : list A                        (* liste vide   *)
  | cons : A -> list A -> list A.        (* ajout en tête *)
```

Notations : `[]` pour `nil`, `x :: l` pour `cons x l`, `[1;2;3]` pour `1 :: 2 :: 3 :: []`.

Les listes : un type inductif paramétré

Même structure que `nat`, mais avec un paramètre de type et une donnée dans le constructeur récursif :

```
Inductive list (A : Type) : Type :=
  | nil  : list A                        (* liste vide   *)
  | cons : A -> list A -> list A.        (* ajout en tête *)
```

Notations : `[]` pour `nil`, `x :: l` pour `cons x l`, `[1;2;3]` pour `1 :: 2 :: 3 :: []`.

Exemple de fonction récursive — concaténation :

```
Fixpoint app {A : Type} (l1 l2 : list A) : list A :=
  match l1 with
  | []     => l2
  | h :: t => h :: (app t l2)
  end.
```

- Récursion sur `l1` — `t` est structurellement plus petit que `h :: t`
- Notation standard : `l1 ++ l2`

Le principe d'induction : nat_ind

Depuis la définition `Inductive nat`, Rocq dérive automatiquement l'éliminateur :

```
nat_ind :
  forall (P : nat -> Prop),
  P 0 ->
  (forall p : nat, P p -> P (S p)) ->
  forall n : nat, P n
```

Le principe d'induction : nat_ind

Depuis la définition `Inductive nat`, Rocq dérive automatiquement l'éliminateur :

```
nat_ind :
  forall (P : nat -> Prop),
  P O ->
  (forall p : nat, P p -> P (S p)) ->
  forall n : nat, P n
```

1. Si $P(0)$
2. Et si, pour tout p, $P(p)$ implique $P(\mathtt{S}\,p)$
3. Alors $P(n)$ pour tout n

$$\frac{P(0) \qquad \forall p,\ P(p) \Rightarrow P(\mathtt{S}\,p)}{\forall n,\ P(n)}$$

Le principe d'induction : nat_ind

Depuis la définition `Inductive nat`, Rocq dérive automatiquement l'éliminateur :

```
nat_ind :
  forall (P : nat -> Prop),
  P O ->
  (forall p : nat, P p -> P (S p)) ->
  forall n : nat, P n
```

1. Si $P(0)$
2. Et si, pour tout p, $P(p)$ implique $P(\mathtt{S}\,p)$
3. Alors $P(n)$ pour tout n

$$\frac{P(0) \qquad \forall p,\ P(p) \Rightarrow P(\mathtt{S}\,p)}{\forall n,\ P(n)}$$

Vérificationnisme (Leçon 4)

Comme `and_ind` et `or_ind`, `nat_ind` est généré par Rocq à partir des *constructeurs* de `nat` — on ne l'écrit pas à la main.

La tactique `induction` : une application de `nat_ind`

Observation clé :

- `0 + n = n` est vrai *par définition* de `add` ⇒ `reflexivity` suffit.
- `n + 0 = n` n'est pas réductible sans connaître la forme de `n` ⇒ il faut raisonner par cas sur `n`.

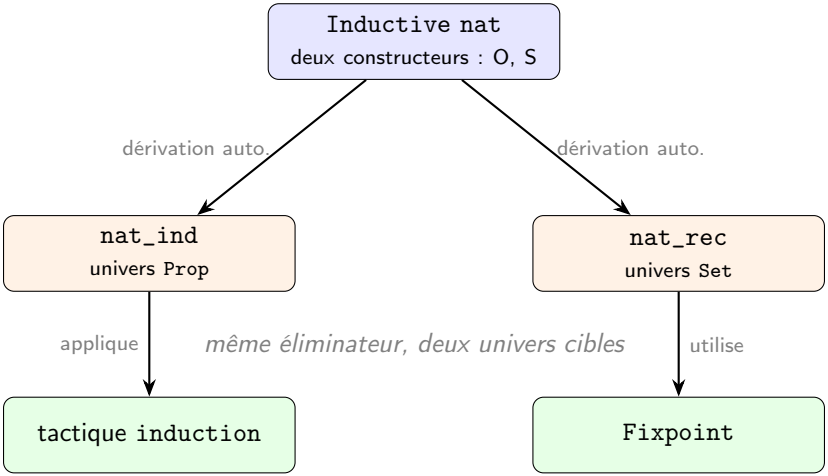
La tactique `induction` : une application de `nat_ind`

Observation clé :

- `0 + n = n` est vrai *par définition* de `add` ⇒ `reflexivity` suffit.
- `n + 0 = n` n'est pas réductible sans connaître la forme de `n` ⇒ il faut raisonner par cas sur `n`.

```
Lemma add_0_r : forall n : nat, n + 0 = n.
Proof.
  intro n.
  induction n as [| p IH].
  -             (* but : 0 + 0 = 0       *) reflexivity.
  -             (* but : S p + 0 = S p   *)
                (* IH  : p + 0 = p       *)
    simpl.      (* S p + 0 ~> S (p + 0)  *)
    rewrite IH. (* S (p + 0) ~> S p      *)
    reflexivity.
Qed.
```

La connexion complète



Synthèse et ouverture

	Propositions (L4)	**Données (L5)**
Mécanisme	`Inductive ...: Prop`	`Inductive ...: Set/Type`
Constructeurs	règles d'introduction	constructeurs de données
Élimination	`match`	`match`
Éliminateur dérivé	`and_ind`, `or_ind`	`nat_rec`, `list_rec`, `nat_ind`, `list_ind`
Tactique de preuve	`destruct`	`induction`

Synthèse et ouverture

	Propositions (L4)	**Données (L5)**
Mécanisme	`Inductive ...: Prop`	`Inductive ...: Set/Type`
Constructeurs	règles d'introduction	constructeurs de données
Élimination	`match`	`match`
Éliminateur dérivé	`and_ind`, `or_ind`	`nat_rec`, `list_rec`, `nat_ind`, `list_ind`
Tactique de preuve	`destruct`	`induction`

Leçon 6 — Vérification formelle

La semaine prochaine, on prouvera que `rev (rev l) = l`. On verra ce que signifie faire de la *vérification formelle* : prouver qu'un programme fait ce qu'on attend de lui.