MDV : cours 2

Types de données inductifs et prédicats inductifs

Les types inductifs en Coq

Type inductifs

Le système Coq est basé sur une logique appelée

Calcul des Constructions Inductives

Dans ce cours nous allons découvrir la partie inductive de l'outil.

Un même mot clé pour deux notions à priori distinctes :

```
Inductive id ... : ... Set := ....   (* structure de données *)
Inductive id ... : ... Prop := ....  (*        prédicat      *)
```

Plan

Plan

1. Structure de données inductives
   - Prouver

Plan

1. Structure de données inductives
   - Prouver
   - Programmer

Plan

1. Structure de données inductives
   - Prouver
   - Programmer
   - Exercices

Plan

Plan

Plan

1. Structure de données inductives
   - Prouver
   - Programmer
   - Exercices
2. Prédicats inductifs
3. D'autres tactiques utiles

Définition inductive

Une définition inductive définie un type de donnée en listant les *constructeurs* qui permettent de construire les objets de ce type.

Exemple : les entiers de Péano

```
Inductive nat : Set :=
   O : nat
 | S : nat → nat.
```

`O : nat` et `S : nat → nat` sont des *constructeurs*.
`nat` est le plus petit type contenant `O` et clos par le successeur `S`.

Exemples :

`O`	(0)
`S O`	(1)
`S (S O)`	(2)
`S (S (S O))`	(3)
...	

Lien avec les types de données Caml

Coq	Caml

<pre>Inductive nat : Set :=
   O : nat
 | S : nat → nat.
 	
Inductive couleur : Set :=
   Bleu : couleur
 | Blanc : couleur
 | Rouge : couleur.</pre>
	<pre>type nat =
	   O
	 | S of nat
	
	type couleur =
	   Bleu
	 | Blanc
	 | Rouge</pre>

Justification théorique

Les entiers naturels sont définis comme la plus petite partie $N \subseteq \{\mathsf{0}, \mathsf{S}\}^*$ vérifiant

$$N = \{\mathsf{0}\} \cup \{\mathsf{S}\, n \mid n \in N\}$$

Justification :

- $(\mathcal{P}(\{\mathsf{0}, \mathsf{S}\}^*), \subseteq, \bigcup)$ est un treillis complet,
- $F : \begin{array}{ccc} \mathcal{P}(\{\mathsf{0}, \mathsf{S}\}^*) & \longrightarrow & \mathcal{P}(\{\mathsf{0}, \mathsf{S}\}^*) \\ X & \longmapsto & \{\mathsf{0}\} \cup \{\mathsf{S}\, n \mid n \in X\} \end{array}$ est un opérateur monotone,
- il admet donc un plus petit point fixe (Th. de Knaster-Tarski)

$$N = F(N)$$

Un définition inductive correspond à un plus petit point fixe

Plan

Preuve par étude de cas

Pour démontrer un propriété sur un objet de type inductif il faut la démontrer dans tous les cas.

```
Require Export Arith. (** Charge le type [nat] des entiers de Péano *)

Lemma tout_entier_est_soit_0_soit_a_un_predecesseur :
 ∀ n : nat, n = 0 ∨ (∃ m : nat, n = S m).
Proof.
  intros.
  destruct n.
  left.
  reflexivity.
  right.
   exists n.
  reflexivity.
Qed.
```

Deux nouvelles tactiques : `destruct` et `reflexivity`.

destruct

Preuve par cas

```
 H : ... O ...
============================
 ... O ...
```

```
 n : nat
 H : ... n ...
============================
 ... n ...
```

```
 n : nat
 H : ... (S n) ...
============================
 ... (S n) ...
```

Commande : `destruct n.`

reflexivity

Pour prouver une égalité triviale entre deux termes identiques

```
============================
 t = t
```

Commande : `reflexivity.`

Remarques

- les deux termes peuvent être égaux à réduction (calcul) prêt,
- la tactique d'automatisation `auto` s'applique aussi dans ce cas.

Principe d'induction structurelle

```
Inductive t : Set :=
| C1 : ... → t
 ...
| Cn : ... → t.
```

Pour montrer une propriété `P` sur tous les objets de type `t`,

- il suffit de prouver `P` sur tous les termes de la forme `(C1 ...)`, . . ., `(Cn ...)`,
- en supposant que la propriété est vérifiée pour chacun des sous-termes de type `t`.

Principe d'induction structurelle

Une propriété de la forme `∀ n:nat, P n` peut être démontrée à l'aide du principe d'induction associé au type `nat` :

```
∀ P : nat → Prop,
  P 0  →  (∀ n : nat, P n → P (S n))  →  ∀ n : nat, P n
```

Remarques :

- Ce principe correspond au raisonnement par récurrence,
- Notez la quantification d'ordre supérieur « pour tout prédicat `P` ».

> En Coq, chaque définition inductive génère un principe d'induction associé

Justification théorique

Rappel :

$$\begin{array}{rrcl} F: & \mathcal{P}(\{\mathsf{0},\mathsf{S}\}^*) & \longrightarrow & \mathcal{P}(\{\mathsf{0},\mathsf{S}\}^*) \\ & X & \longmapsto & \{\mathsf{0}\} \cup \{\mathsf{S}\ n \mid n \in X\} \end{array}$$

Le théorème de Knaster-Tarski nous dit plus précisément que

N est le plus petit ensemble X vérifiant $F(X) \subseteq X$

Conséquence : si une partie $P \subseteq (\{\mathsf{0},\mathsf{S}\}^*)$ vérifie

- $\mathsf{0} \in P$
- pour tout $n \in (\{\mathsf{0},\mathsf{S}\}^*)$, $n \in P$ implique $\mathsf{S}\ n \in P$

alors $F(P) \subseteq P$ et donc $N \subseteq P$: tous les éléments de N vérifient la propriété représentée par P.

Exemple

```
Section induction.
  Variable double : nat → nat.
  Hypothesis double_0 : double 0 = 0.
  Hypothesis double_S : ∀ n, double (S n) = S (S (double n)).

  Variable pair : nat → Prop.
  Hypothesis pair_0 : pair 0.
  Hypothesis pair_S : ∀ n, pair n → pair (S (S n)).

  Lemma double_pair : ∀ n, pair (double n).
  Proof.
    induction n.
    rewrite double_0.
    apply pair_0.
    rewrite double_S.
    apply pair_S.
    assumption.
  Qed.
End induction.
```

Les sections Coq

Une section permet d'ajouter temporairement des déclarations locales au contexte.

```
Section foo.
  Variable a : A.       (* Soit a de type A *)
  Hypothesis H : P a. (* Supposons que a vérifie P *)
  Lemma PimpQ : Q a.  (* Montrons que a vérifie Q *)
  Proof. ... Qed.
End foo.
```

A la fin de la section, les déclarations sont *déchargées*

```
Check PimpQ.          (* Vérifions le type de PimpQ hors de la section *)
 > PimpQ : ∀ a : A, P a → Q a
```

induction

Preuve par induction

```
  ============================
   P 0
```

```
  n : nat
  Hn : P n
  ============================
  P (S n)
```

```
  ============================
   ∀ n, P n
```

Commande : `induction n.`

rewrite

Réécrire une égalité dans le but courant

```
H : t1 = t2
H0 : ... t1 ...
============================
... t1 ...
```

```
H : t1 = t2
H0 : ... t1 ...
============================
... t2 ...
```

Commande : `rewrite H.`

Variantes

- `rewrite <- H.` pour remplacer `t2` par `t1`,
- `rewrite H in H0.` pour réécrire dans l'hypothèse `H0`,
- `pattern t1 at i; rewrite H.` pour seulement réécrire la `i`-ème occurrence de `t1` dans le but.

D'autres exemples : les enregistrements

```
Record rational : Set := rat { num : nat; denum:nat }.

Definition demi := rat 1 2.

Eval compute in demi.(num). (* évaluation d'un expression *)
 > 1
Eval compute in demi.(denum).
 > 2
Print rational.               (* affiche une définition *)
 > Inductive rational : Set :=  rat : nat → nat → rational
```

Un enregistrement est encodé avec un type inductif possédant un seul constructeur.

D'autres exemples : les listes polymorphes

```
Inductive list (A:Set) : Set :=
  nil : list A
| cons : A → list A → list A.
```

`A:Set` est un *paramètre* de l'inductif.

`cons` et `nil` ont un paramètre supplémentaire : `A:Set`.

```
Definition l1 := cons bool true (cons bool false (nil bool)).
Definition l2 := cons nat 1 (cons nat 2 (nil nat)).
```

Cet argument peut être inféré par Coq.

```
Definition l1' := cons _ true (cons _ false (nil _)).
Definition l2' := cons _ 1 (cons _ 2 (nil _)).
```

La notion d'argument implicite permet de cacher ce type d'argument

```
Implicit Arguments nil [A].
Implicit Arguments cons [A].
Definition l1'' := cons true (cons false nil).
Definition l2'' := cons 1 (cons 2 nil).
```

Exercice

Donner le principe d'induction associé au type `list`.

Donner le principe d'induction associé au type `tree` défini par

```
Inductive tree (A:Set) : Set :=
  leaf : tree A
| node : A → tree A → tree A → tree A.
Implicit Arguments leaf [A].
Implicit Arguments node [A].
```

Généralisation : induction bien fondée

Soit A un ensemble, une relation $\prec \subseteq A \times A$ est dite *bien fondée* si il n'existe pas de suite infinie $a_1, \ldots, a_n, \ldots$ d'éléments de A telle que

$$\cdots \prec a_n \prec \cdots \prec a_1$$

Principe d'induction bien fondé

Soit Q une propriété sur A, si

$$\forall x \in A,\ (\forall y \in A,\ y \prec x \Rightarrow\ Q(y)) \implies Q(x)$$

alors

$$\forall x \in A,\ Q(x))$$

Questions

- quelle est la relation bien fondée sous-jacente au principe d'induction structurelle ?
- quelle est la relation bien fondée sous-jacente au principe de récurrence classique ?
- quelle est la relation bien fondée sous-jacente au principe de récurrence forte ?

Plan

Programmation par cas

La données de type inductif se manipulent généralement à l'aide d'un filtrage.

```
Definition is_zero (n:nat) :=
  match n with
    O ⇒ true
  | S p ⇒ false
  end.
```

La tactique `simpl` permet de simplifier un terme comme `is_zero (S n)`.
Le filtrage doit toujours être exhaustif :

En Coq, les fonctions sont totales.

Programmation par cas

Quand le type a seulement deux constructeurs, on peut utiliser la syntaxe `if .. then ... else`.

```
Definition b2i (x:bool) : nat :=
  if x then 0 else 1.
```

```
Definition b2i (x:bool) : nat :=
  match x with
    true ⇒ 0
  | false ⇒ 1
  end.
```

Quand le type a seulement un constructeur, on peut utiliser la syntaxe `let ... := ... in ...`.

```
Inductive triplet (A:Set) : Set :=
  trip : A → A → A → triplet A.
```

```
Definition fst3
  (A:Set) (t:triplet A) : A :=
  let (x,y,z) := t in x.
```

```
Definition fst3
   (A:Set) (t:triplet A) : A :=
  match t with trip x y z ⇒ x end.
```

Programmation récursive

La récursion n'est autorisée que sur des sous-termes de l'argument principal : récurrence structurelle.

```
Fixpoint double (n : nat) : nat :=
  match n with
  | O ⇒ O
  | S p ⇒ S (S (double p))
  end.
```

En Coq, toutes les fonctions terminent.

L'argument principal est éventuellement précisé avec `{struct n}`.

Plan

Exercices

http://www.irisa.fr/lande/pichardie/M2/MDV/

Compléter le fichier `arithmetique.v`

1. Prouver
   ```
   Lemma plus_commutative : ∀ n m : nat, plus n m = plus m n.
   ```
2. Prouver
   ```
   Lemma plus_associative : ∀ a b c : nat, plus a (plus b c) = plus (plus a b) c.
   ```
3. Définir une version récursive terminale (*tail-recursive*) de `plus`.
   Montrer son équivalence avec `plus`.

Compléter le fichier `append.v`

Plan

Prédicats inductifs

Nous avons vu comment définir des ensembles (types) par point fixe, regardons maintenant comment définir des sous-ensembles (propriétés).

> Une propriété inductive est définie par une liste de constructeurs.
> Les constructeurs définissent les règles qui permettent de prouver la propriété.

Exemples

Être pair :

```
Inductive pair : nat → Prop :=
 | pair_O : pair O
 | pair_S : ∀ n, pair n → pair (S (S n)).
```

Un prédicat inductif est associé à un principe d'induction

```
pair_ind
     : ∀ P : nat → Prop,
       P O →
       (∀ n : nat, pair n → P n → P (S (S n))) →
       ∀ n : nat, pair n → P n
```

Exemples de preuve

```
Lemma succ_non_pair : ∀ n, pair (S n) → ¬ pair n.
Proof.
  induction n; intros.
  inversion H.
  intros H1.
  inversion_subst H.
  elim IHn; assumption.
Qed.
```

```
Lemma double_pair : ∀ n, pair (double n).
Proof.
  induction n; simpl; constructor; assumption.
Qed.
```

```
Lemma pair_plus : ∀ n m, pair n → pair m → pair (plus n m).
Proof.
  induction 2; simpl.
  assumption.
  constructor.
  assumption.
Qed.
```

inversion

Raisonnement par cas sur une hypothèse inductive

```
H : pair n
H0 : ... n ...
============================
... n ...
```

```
H : pair n
H0 : ... n ...
H1 : O = n
============================
... O ...
```

```
H : pair n
H0 : ... n ...
n0 : nat
H1 : pair n0
H2 : S (S n0) = n
============================
... (S (S n0)) ...
```

Commande : `inversion H.`

Variante

- `inversion_subst` permet de nettoyer les égalités générées (`Require Export Tac`)

constructor

Appliquer l'une des règles d'un prédicat inductif

```
  n : nat
============================
  pair (S (S n))
```

```
  n : nat
============================
  pair n
```

Commande : `constructor 2` (`pair_S` est le 2ème constructeur)

Variante

- `constructor` utilise la première règle applicable (mais pas forcément la bonne si plusieurs règles s'appliquent)

D'autres exemples de prédicats inductifs

```
Inductive In : A → list A → Prop :=
 | In0 : ∀ a q, In a (cons a q)
 | In1 : ∀ a x q, In a q → In a (cons x q)
```

Lorsque qu'un argument est constant, mieux vaut le mettre en paramètre.

```
Inductive In (a:A) : list A → Prop :=
 | In0 : ∀ q, In a (cons a q)
 | In1 : ∀ x q, In a q → In a (cons x q)
```

D'autres exemples de prédicats inductifs

```
Inductive le (n:nat) : nat → Prop :=
 | le_n : le n n
 | le_S : ∀ m, le n m → le n (S m).
```

Variante

```
Inductive le' : nat → nat → Prop :=
 | le_O : ∀ n, le' O n
 | le_SS : ∀ n m, le' n m → le' (S n) (S m).
```

Exercice (maison) : démontrer l'équivalence entre les deux prédicats.

Exercices

1. Définir un prédicat inductif `last : A → list A → Prop` pour exprimer qu'un élément est le dernier élément d'une liste.
2. Montrer
   ```
   Lemma last_rev : ∀ a l, last a l → ∃ q, rev _ l = cons a q.
   ```
   de deux façons :
   - par induction sur `l`,
   - en utilisant le principe d'induction de `last` (tactique `induction 1`).
3. Définir une fonction `get_last : list A → option A` renvoyant le dernier élément d'une liste, ou `None` s'il n'existe pas (`Print option`). Démontrer la correction de `get_last` par rapport au prédicat `last`.

Plan

`subst`

Cas particulier de réécriture : `t1` est une variable

```
  x : A
  H : x = t2
  H0 : ... x ...                     H0 : ... t2 ...
============================      ============================
  ... x ...                          ... t2 ...
```

Commande : `subst x.`

Remarques

- supprime complètement `x` du contexte courant,
- `subst` (sans arguments) itère `subst x` pour tous les identificateurs `x` du contexte.

`discriminate`

Lorsque qu'une égalité en hypothèse est impossible à cause de constructeurs distincts

```
  H : S n = O
============================
  ...
```

Commande : `discriminate H.`

Variante

- `discriminate` (sans argument) recherche un hypothèse qui s'applique,
- `discriminate` permet aussi de prouver un but du type `S n <> O`.

`injection`

Pour utiliser l’injectivité des constructeurs

```
 H : S n1 = S n2
============================
 ...
```

```
 H : S n1 = S n2
============================
 n1 = n2 → ...
```

Commande : `injection H.`

Remarque

- technique d’utilisation standard : `injection H; intros; subst.`

`case_eq`

Preuve par cas en « mémorisant » l'égalité

```
                              H : ... n ...
                              ============================
                              n = 0 → ... 0 ...

  n : nat                     n : nat
  H : ... n ...               H : ... n ...
============================  ============================
 ... n ...                    ∀ p : nat, n = S p → ... (S p) ...
```

Commande : `case_eq n.`

Remarque

- ce n'est pas une tactique disponible par défaut (faire `Require Export Tac.` pour l'avoir).

`assert`

Pour instancier une propriété quantifiée

```
 n : nat
 H : ∀ x, P x
============================
 ...
```

```
 n : nat
 H : ∀ x, P x
 Hn : P n
============================
 ...
```

Commande : `assert (Hn:=H n).`

Remarque

- utile pour pouvoir utiliser ensuite **`rewrite`** ou **`inversion`** sur `Hn`.