

MDV : cours 4

Programmation certifiée avancée

Plan

- 1 L'envers du décor : preuve = λ -terme

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude
- 3 Le système de modules de Coq

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude
- 3 Le système de modules de Coq
- 4 Exercice : programmer une librairie d'ensembles finis

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude
- 3 Le système de modules de Coq
- 4 Exercice : programmer une librairie d'ensembles finis

L'envers du décor

Coq est basé sur un λ -calcul richement typé (+ types inductifs)

CCI = Calcul des Construction Inductives

Les λ -termes servent

- ▶ à représenter des programmes
- ▶ à représenter des preuves

Les types servent

- ▶ à donner un type aux programmes
- ▶ à écrire des formules logiques (énoncés de théorème)

Où sont les programmes, où sont les preuves ?

Pour garder un peu d'intuition, on peut se rattacher aux sortes **Set** et **Prop**.

Exemple : pour $h : T$

- ▶ si $T : \mathbf{Prop}$,
 - ▶ T est vue comme une propriété,
 - ▶ h comme une preuve,
- ▶ si $T : \mathbf{Set}$,
 - ▶ T est vue comme un type (une spécification),
 - ▶ h comme un programme.

Précision de vocabulaire

Dans une preuve interactive du genre :

```

P : Prop
H : P
=====
...

```

Ici, H n'est pas vraiment le nom d'une hypothèse mais plutôt le nom d'une *preuve* de P , dont on suppose l'existence.

Lorsqu'on énonce et prouve un théorème :

```

Theorem toto : 0 + 0 = 0.
Proof. auto. Qed.

```

Ici, `toto` n'est pas le nom que l'on donne à l'énoncé mais plutôt celui que l'on donne à la preuve (au λ -terme sous-jacent).

Extraction

Le mécanisme d'extraction transforme un λ -terme Coq en objet Caml (un type ou un terme).

Ce mécanisme d'extraction utilise la distinction **Set/Prop** :

- ▶ les objets de type **Set** sont conservés (contenu calculatoire)
- ▶ les objets de type **Prop** sont effacés (contenu logique)

...je simplifie un petit peu...

Quelques types inductifs de références

Nous allons maintenant énumérer plusieurs exemples de types inductifs (essentiellement des constructeurs de type) pour montrer

- ▶ que la plupart des constructeurs logiques ne sont pas des objets *primitifs*, mais forme une *sur-couche* sur le CCI,
- ▶ comment définir des spécifications de programmes riches.

Le type produit

```
Inductive prod (A : Set) (B : Set) : Set :=  
  | pair : A → B → A * B
```

Remarque : `prod A B` est noté `A * B`.

```
Definition p : prod nat bool := pair 1 true.
```

Extraction : `type ('a, 'b) prod = | Pair of 'a * 'b`

Le *et* logique

```

Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A → B → A ∧ B

```

Remarque : `and A B` est noté $A \wedge B$.

```

Section exemple.

```

```

  Variable h1 : 3 <= 6.

```

```

  Variable h2 : true<>false.

```

```

  Definition P : 3 <= 6 ∧ true<>false := conj h1 h2.

```

```

End exemple.

```

Extraction : \emptyset (pas de contenu calculatoire)

Le type somme

```
Inductive sum (A : Set) (B : Set) : Set :=
| inl : A → A + B
| inr : B → A + B
```

Remarques :

- ▶ `sum A B` est noté `A + B`
- ▶ le 1er argument de `inl` est implicite
- ▶ la notation `@` annule le mécanisme d'arguments implicites

```
Definition p : sum nat bool := @inl nat bool 1.
```

```
Definition p : sum nat bool := inl bool 1.
```

Extraction :

```
type ('a, 'b) sum =
| Inl of 'a
| Inr of 'b
```

Le *or* logique

```

Inductive or (A : Prop) (B : Prop) : Prop :=
  | or_introl : A  $\rightarrow$  A  $\vee$  B
  | or_intror : B  $\rightarrow$  A  $\vee$  B

```

Remarque : `or A B` est noté $A \vee B$.

Section exemple.

```

Variable h1 : 3 <= 6.

```

```

Variable h2 : true<>false.

```

```

Definition P : 3 <= 6  $\vee$  true<>false := @or_introl _ _ h1.

```

End exemple.

Aucune extraction.

Les booléens logiques

```

Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A  $\rightarrow$  {A}+{B}
  | right : B  $\rightarrow$  {A}+{B}

```

Remarque : `sumbool A B` est noté $\{A\}+\{B\}$.

```

Definition eq_nat_dec :  $\forall$  n m : nat, {n=m}+{n<>m} := ...

```

```

Variable h : 1=1.

```

```

Definition essai : {1=1}+{1<>1} := @left _ _ h.

```

Extraction : `type sumbool = | Left | Right`

Exercice : un vrai type booléen logique

Proposez un type `richbool P` équivalent à $\{P\} + \{\neg P\}$.

Quantificateur existentiel

```
Inductive ex (A:Set) (P:A  $\rightarrow$  Prop) : Prop :=
| ex_intro :  $\forall$  x:A, P x  $\rightarrow$  ex A P.
```

Remarque : $\text{ex } A \text{ (fun } x \Rightarrow Q)$ est noté $\exists x, Q$.

```
Variable h : 1=1.
```

```
Definition essai :  $\exists$  x, x=1 := @ex_intro nat (fun x  $\Rightarrow$  x=1) 1 h.
```

Il s'agit d'un paire dont le deuxième élément a un type qui dépend du premier élément.

Sous-ensemble

```
Inductive sig (A : Set) (P : A  $\rightarrow$  Prop) : Set :=
  exist :  $\forall$  x : A, P x  $\rightarrow$  sig P
```

Remarques :

- ▶ sig A (**fun** x \Rightarrow Q) est noté { x:A | Q }
- ▶ la notation sous-ensemble est un peu trompeuse : un objet de type { x:A | Q }, n'est pas de type A !

```
Variable h : 1=1.
```

```
Definition essai : { x:nat | x=1 } := exist (fun x  $\Rightarrow$  x=1) 1 h.
```

Extraction :

```
type 'a sig0 = 'a
```

```
let essai = S 0
```

Exercice

Pour illustrer la différence entre `ex` et `sig`

- 1 prouver le lemme Coq suivant :

```
Lemma pred_ex :  $\forall n:\text{nat}, n <> 0 \rightarrow \exists p, n = S p.$ 
```

```
Proof.
```

```
...
Qed.
```

- 2 puis utiliser le même script de preuve pour prouver le lemme

```
Lemma pred_sig :  $\forall n:\text{nat}, n <> 0 \rightarrow \{ p:\text{nat} \mid n = S p \}.$ 
```

```
Proof.
```

```
...
Qed.
```

- 3 puis tester la commande `Extraction pred_sig`.

Exercice

Pour illustrer la différence entre `ex` et `sig`

- 1 prouver le lemme Coq suivant :

```
Lemma pred_ex :  $\forall n:\text{nat}, n <> 0 \rightarrow \exists p, n = S p.$ 
```

```
Proof.
```

```
...
Qed.
```

- 2 puis utiliser le même script de preuve pour prouver le lemme

```
Lemma pred_sig :  $\forall n:\text{nat}, n <> 0 \rightarrow \{ p:\text{nat} \mid n = S p \}.$ 
```

```
Proof.
```

```
...
Qed.
```

- 3 puis tester la commande `Extraction pred_sig.`

```
let pred = function
  | 0  $\rightarrow$  assert false (* absurd case *)
  | S n0  $\rightarrow$  n0
```

Programmation par preuve

Le langage de tactiques (`intros`, `auto`, ...) est en fait un langage de construction interactive de λ -terme.

- ▶ `intros` : abstraction `fun x \Rightarrow ...`
- ▶ `destruct` (ou `case`) : filtrage `match .. with ... end`
- ▶ `apply` : application d'une fonction.

Exemple de programmation par preuve

```
Variable A : Set.  
Variable a0 a1 : A.  
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
Definition F : nat → A :=  
  fun n ⇒  
    match n with  
    | 0 ⇒ a0  
    | S p ⇒ f p a1  
  end.
```

Exemple de programmation par preuve

Variable $A : \text{Set}$.

Variable $a0\ a1 : A$.

Variable $f : \text{nat} \rightarrow A \rightarrow A$.

Definition $F' : \text{nat} \rightarrow A$.

Definition $F : \text{nat} \rightarrow A :=$

```
fun n =>
  match n with
  | 0 => a0
  | S p => f p a1
end.
```

Exemple de programmation par preuve

```
Variable A : Set.  
Variable a0 a1 : A.  
Variable f : nat → A → A.
```

```
Definition F' : nat → A.  
  intros n.
```

```
Definition F : nat → A :=  
  fun n ⇒  
    match n with  
    | 0 ⇒ a0  
    | S p ⇒ f p a1  
  end.
```

Exemple de programmation par preuve

```
Variable A : Set.
Variable a0 a1 : A.
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
intros n.
destruct n as [ | p ].
```

Definition F : nat → A :=

```
fun n ⇒
  match n with
  | 0 ⇒ a0
  | S p ⇒ f p a1
end.
```

Exemple de programmation par preuve

```
Variable A : Set.
Variable a0 a1 : A.
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
  intros n.
  destruct n as [ | p ].
  apply a0.
```

Definition F : nat → A :=

```
  fun n ⇒
    match n with
    | 0 ⇒ a0
    | S p ⇒ f p a1
  end.
```

Exemple de programmation par preuve

```
Variable A : Set.  
Variable a0 a1 : A.  
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
  intros n.  
  destruct n as [ | p ].  
  apply a0.  
  apply f.
```

Definition F : nat → A :=

```
  fun n =>  
    match n with  
    | 0 => a0  
    | S p => f p a1  
  end.
```

Exemple de programmation par preuve

```
Variable A : Set.
Variable a0 a1 : A.
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
  intros n.
  destruct n as [ | p ].
  apply a0.
  apply f.
  apply p.
```

Definition F : nat → A :=

```
  fun n ⇒
    match n with
    | 0 ⇒ a0
    | S p ⇒ f p a1
  end.
```

Exemple de programmation par preuve

```
Variable A : Set.
Variable a0 a1 : A.
Variable f : nat → A → A.
```

Definition F' : nat → A.

```
  intros n.
  destruct n as [ | p ].
  apply a0.
  apply f.
  apply p.
  apply a1.
```

Qed.

Definition F : nat → A :=

```
  fun n ⇒
    match n with
    | 0 ⇒ a0
    | S p ⇒ f p a1
    end.
```

La commande **Print** F' affiche le même terme que F.

Remarques

- ▶ la programmation par preuve est parfois utile pour programmer des fonctions ayant des types très riches,
- ▶ la programmation standard étant dans ces cas, assez technique,
- ▶ attention à l'automatisation !
 - ▶ mieux vaut ne pas laisser l'automatisation contrôler la définition (et l'efficacité) du programme

Definition $f : \text{nat} \rightarrow \text{nat}.$

auto. (qu'est ce qui est programmé ? *)*

Qed.

- ▶ Le mot clé **Defined** est plus approprié que **Qed**.
 - ▶ **Qed** rend *opaque* la fonction.

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude**
- 3 Le système de modules de Coq
- 4 Exercice : programmer une librairie d'ensembles finis

Quelques lemmes utiles

Require Export Arith.

Lemma `le_n_plus_n_0` : $\forall n, n \leq 0 \rightarrow n + 0 = 0$.

Proof.

`intros n H; inversion H; auto.`

Qed.

Lemma `n_eq_Sm_plus_n_0` : $\forall m n, n = S m \rightarrow n + 0 = S m$.

Proof.

`intros n m H; rewrite H; auto.`

Qed.

Definition `nat_dec` : $\forall n m : \text{nat}, \{n=m\} + \{n < m\} := \text{eq_nat_dec}$.

Lemma `le_n_Sm_diff` : $\forall m n, n \leq S m \rightarrow n < S m \rightarrow n \leq m$.

Proof.

`intros n m H Heq; inversion H; intuition.`

Qed.

Lemma `plus_n_Sm` : $\forall m n, n + S m = S (n + m)$.

Proof.

`induction n; simpl; auto.`

Qed.

Lemma `plus_Sp_Sm` : $\forall m p n, n + p = m \rightarrow n + S p = S m$.

Proof.

`intros n m p H; rewrite <- H; apply plus_n_Sm.`

Qed.

Hint Resolve `le_n_plus_n_0 n_eq_Sm_plus_n_0 le_n_Sm_diff plus_n_Sm plus_Sp_Sm`.

La soustraction entière

```

Fixpoint minus (m n:nat) {struct m} : nat :=
  match m with
  | 0 => 0
  | S p =>
    match nat_dec n (S p) with
    | left _ => 0
    | right _ => S (minus p n)
    end
  end.

```

Theorem minus_OK : $\forall m n, n \leq m \rightarrow n + (\text{minus } m \ n) = m$.

Proof.

```

induction m; intros; simpl; auto.
destruct nat_dec; auto.

```

Qed.

Une spécification plus riche : fonction partielle

```
minus1 :  $\forall (m\ n:\text{nat}), n \leq m \rightarrow \text{nat}$ 
```

```
n : nat
m : nat
h : n <= m
```

```
=====
```

```
... (minus1 n m h) ...
```

`minus1` prend trois arguments : deux entiers et une preuve

Nous restreignons ainsi le domaine de définition : fonction partielle.

Une spécification plus riche : fonction partielle

```

Fixpoint minus1 (m n:nat) {struct m} : n <= m → nat :=
  match m return n<=m → nat with
  | 0 ⇒ fun H: n <= 0 ⇒ 0
  | S p ⇒ fun H: n <= S p ⇒
    match nat_dec n (S p) with
    | left _ ⇒ 0
    | right heq ⇒
      S (minus1 p n      ...      )
    end
  end
end.

```

Contexte :

```

le_n_Sm_diff : ∀ m n, n <= S m → n <> S m → n <= m
p : nat
n : nat
H : n <= S p
heq : n <> S p

```

Nous cherchons une preuve de $n <= p$.

Une spécification plus riche : fonction partielle

```

Fixpoint minus1 (m n:nat) {struct m} : n <= m → nat :=
  match m return n<=m → nat with
  | 0 ⇒ fun H: n <= 0 ⇒ 0
  | S p ⇒ fun H: n <= S p ⇒
    match nat_dec n (S p) with
    | left _ ⇒ 0
    | right heq ⇒
      S (minus1 p n      ...      )
    end
  end
end.

```

Contexte :

```

le_n_Sm_diff : ∀ m n, n <= S m → n <> S m → n <= m
p : nat
n : nat
H : n <= S p
heq : n <> S p

```

Nous cherchons une preuve de $n <= p$.

```
le_n_Sm_diff p n : n <= S p → n <> S p → n <= p
```

Une spécification plus riche : fonction partielle

```

Fixpoint minus1 (m n:nat) {struct m} : n <= m → nat :=
  match m return n<=m → nat with
  | 0 ⇒ fun H: n <= 0 ⇒ 0
  | S p ⇒ fun H: n <= S p ⇒
    match nat_dec n (S p) with
    | left _ ⇒ 0
    | right heq ⇒
      S (minus1 p n      ...      )
  end
end.

```

Contexte :

```

le_n_Sm_diff : ∀ m n, n <= S m → n <> S m → n <= m
p : nat
n : nat
H : n <= S p
heq : n <> S p

```

Nous cherchons une preuve de $n <= p$.

```

le_n_Sm_diff p n : n <= S p → n <> S p → n <= p
le_n_Sm_diff p n H : n <> S p → n <= p

```

Une spécification plus riche : fonction partielle

```

Fixpoint minus1 (m n:nat) {struct m} : n <= m → nat :=
  match m return n<=m → nat with
  | 0 ⇒ fun H: n <= 0 ⇒ 0
  | S p ⇒ fun H: n <= S p ⇒
    match nat_dec n (S p) with
    | left _ ⇒ 0
    | right heq ⇒
      S (minus1 p n      ...      )
    end
  end
end.

```

Contexte :

```

le_n_Sm_diff : ∀ m n, n <= S m → n <> S m → n <= m
p : nat
n : nat
H : n <= S p
heq : n <> S p

```

Nous cherchons une preuve de $n \leq p$.

```

le_n_Sm_diff p n : n <= S p → n <> S p → n <= p
le_n_Sm_diff p n H : n <> S p → n <= p
le_n_Sm_diff p n H heq : n <= p

```

Une spécification plus riche : fonction partielle

```

Fixpoint minus1 (m n:nat) {struct m} : n <= m → nat :=
  match m return n<=m → nat with
  | 0 ⇒ fun H: n <= 0 ⇒ 0
  | S p ⇒ fun H: n <= S p ⇒
    match nat_dec n (S p) with
    | left _ ⇒ 0
    | right heq ⇒
      S (minus1 p n (le_n_Sm_diff p n H heq))
    end
  end.

```

Contexte :

```

le_n_Sm_diff : ∀ m n, n <= S m → n <> S m → n <= m
p : nat
n : nat
H : n <= S p
heq : n <> S p

```

Nous cherchons une preuve de $n \leq p$.

```

le_n_Sm_diff p n : n <= S p → n <> S p → n <= p
le_n_Sm_diff p n H : n <> S p → n <= p
le_n_Sm_diff p n H heq : n <= p

```

Une spécification plus riche : fonction partielle

Extraction `minus1`.

```
let rec minus1 m n =
  match m with
  | 0 → 0
  | S p →
    (match nat_dec n (S p) with
     | Left → 0
     | Right → S (minus1 p n))
```

même preuve de correction que précédemment

Theorem `minus1_OK` : $\forall m n (H:n \leq m), n + (\text{minus1 } m \ n \ H) = m$.

Proof.

```
induction m; intros; simpl; auto.
destruct nat_dec; auto.
```

Qed.

Une spécification très riche

$$\text{minus2} : \forall m n, n \leq m \rightarrow \{ p:\text{nat} \mid n + p = m \}$$

`minus2` prend trois arguments (deux entiers et une preuve) et renvoie un entier et une preuve.

Il y a plusieurs façons de programmer ce type de fonctions.

Programmation par preuve

Definition `minus2` : $\forall m n, n \leq m \rightarrow \{ p:\text{nat} \mid n + p = m \}$.

Proof.

```
induction m; intros.  
exists 0; auto.  
destruct (nat_dec n (S m)).  
exists 0; auto.  
destruct (IHm n) as [p Hp].  
auto.  
exists (S p); auto.
```

Defined.

Programmation par preuve et terme

Definition `type_minus m n := n <= m → { p:nat | n + p = m }.`

Definition `minus3 : ∀ m n, type_minus m n.`

Proof.

```

refine (
fix f (m n:nat) {struct m} : type_minus m n :=
  match m return type_minus m n with
    | 0 ⇒ fun H ⇒ exist _ 0 _
    | S p ⇒ fun H ⇒
      match nat_dec n (S p) with
        | left Heq ⇒ exist _ 0 _
        | right Heq ⇒ let (x,Hx) := (f p n _) in exist _ (S x) _
      end
    end
  ).
auto.
auto.
auto.
auto.
Qed.

```

Programmation par terme

Definition `type_minus m n := n <= m → { p:nat | n + p = m }.`

```

Fixpoint minus4 (m n:nat) {struct m} : type_minus m n :=
  match m return type_minus m n with
  | 0 ⇒ fun H ⇒ exist (fun p ⇒ n + p = 0) 0 (le_n_plus_n_0 n H)
  | S p ⇒ fun H ⇒
    match nat_dec n (S p) with
    | left Heq ⇒
      exist (fun x ⇒ n + x = S p) 0 (n_eq_Sm_plus_n_0 p n Heq)
    | right Heq ⇒
      let (x,Hx) := minus4 p n (le_n_Sm_diff p n H Heq) in
      exist (fun x ⇒ n + x = S p) (S x) (plus_Sp_Sm p x n Hx)
    end
  end.

```

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude
- 3 Le système de modules de Coq**
- 4 Exercice : programmer une librairie d'ensembles finis

Le système de modules de Coq

Le système de module de Coq est similaire à celui de Caml.
Trois grandes notions :

- ▶ les *modules*
- ▶ les *signatures* (interfaces)
- ▶ les *foncteurs* (modules paramétrés)

Modules

Un module regroupe une collection d'objets Coq.

```

Module TestZero.
  Definition t := nat.
  Definition P (n:nat) := n <> 0.
  Definition test (n:nat) :=
    match n with
    | 0 => false
    | _ => true
    end.

  Lemma test_computes_P :  $\forall n, P\ n \leftrightarrow test\ n = true.$ 
  Proof. ...(* preuve *)... Qed.
End TestZero.
```

Signatures

Chaque module admet un type : *une signature*.

Une signature (parmi d'autres) du module `TestZero` est

```
Module Type DECPROP.  
  Parameter t : Set.  
  Parameter P : t → Prop.  
  Parameter test : t → bool.  
  Parameter test_computes_P : ∀ n, P n ↔ test n = true.  
End DECPROP.
```

Signatures

Une autre possibilité

```
Module Type DECPROP'.
  Parameter t : Set.
End DECPROP.
```

Tout module déclaré avec cette signature pourra uniquement accéder à son élément `t`. Les autres éléments sont cachés.

Une autre possibilité : signature spécialisée

- ▶ `DECPROP with Definition t:=nat` est une signature correcte pour tous les modules de type `DECPROP` pour lesquels `t` est égal à `nat`.

Foncteurs

Un foncteur est une fonction qui prend des modules en argument et produit un module en résultat.

```

Module ListDec (T:DECPROP) <: DECPROP.
  Definition t := list T.t.
  Definition P (l:t):=  $\forall x, \text{In } x \text{ l} \rightarrow T.P \ x$ .
  Fixpoint test (l:t) : bool :=
    match l with []  $\Rightarrow$  true | x::q  $\Rightarrow$  (T.test x)&&(test q) end.

  Lemma test_computes_P :  $\forall l, P \ l \leftrightarrow \text{test } l = \text{true}$ .
  Proof. ...(* proof script omitted *)... Qed.
End ListDec.

```

Ce foncteur prend en argument un module respectant la signature `DECPROP` et retourne un module de signature `DECPROP` opérant sur les listes.

Foncteurs

Les dépendances entre le module produit et les modules arguments peuvent être spécifiés avec le mot-clé **with**.

```
Module ListDec (T:DECPROP) <: DECPROP
  with Definition t := list T.t
  with Definition P (l:t):=  $\forall x, \text{In } x \text{ l} \rightarrow T.P \ x.$ 
```

donne la nature exact du module construit.

On peut alors construire de nouveaux modules

```
Module ListListTestZero := ListDec(ListDec(TestZero)).
```

Plan

- 1 L'envers du décor : preuve = λ -terme
- 2 Spécification riche : un cas d'étude
- 3 Le système de modules de Coq
- 4 Exercice : programmer une librairie d'ensembles finis

Une première interface

```

Inductive comp : Set :=
  | Lt : comp
  | Eq : comp
  | Gt : comp.

```

```

Module Type OrderedType.

```

```

Parameter t : Set.
Parameter compare : t → t → comp.

```

```

Parameter compare_eq_prop1 : ∀ x y, compare x y = Eq → x = y.
Parameter compare_eq_prop2 : ∀ x y, x = y → compare x y = Eq.

```

```

Definition lt (x y:t) : Prop := compare x y = Lt.
Parameter lt_trans : ∀ x y z, lt x y → lt y z → lt x z.
Parameter lt_not_eq : ∀ x y, lt x y → x <> y.

```

```

Hint Resolve lt_trans lt_not_eq compare_eq_prop1 compare_eq_prop2.

```

```

End OrderedType.

```

Nous spécifions ainsi les type sur lesquels nous savons faire des test d'égalité et de comparaison.

Une deuxième interface

Module Type FSet.

```
Parameter elt : Set.
Parameter t : Set.
Parameter empty : t.
Parameter mem : elt → t → bool.
Parameter elements : t → list elt.
Parameter add : elt → t → t.
Parameter remove : elt → t → t.
Parameter union : t → t → t.
Parameter inter : t → t → t.
```

Definition In_set : elt → t → Prop := fun x s ⇒ In x (elements s).

```
Parameter empty_prop : ∀ x, ¬ In_set x empty.
Parameter mem_prop : ∀ x s, In_set x s ↔ mem x s = true.
Parameter add_prop : ∀ a s x, In_set x (add a s) ↔ (x=a ∨ In_set x s).
Parameter remove_prop : ∀ a s x,
  In_set x (remove a s) ↔ (x<>a ∧ In_set x s).
Parameter union_prop : ∀ s1 s2 x,
  In_set x (union s1 s2) ↔ (In_set x s1 ∨ In_set x s2).
Parameter inter_prop : ∀ s1 s2 x,
  In_set x (inter s1 s2) ↔ (In_set x s1 ∧ In_set x s2).
```

End FSet.

Objectif

cf fichier `sort.v`

Programmer un foncteur de type

```
Module Make (O:OrderedType) : FSet with Definition elt := O.t.  
  ...  
End Make.
```

en utilisant des listes triés.

Les listes triés

Definition `Inf x l : Prop := $\forall y, \text{In } y \text{ l} \rightarrow 0.\text{lt } x \text{ y}$.`

Inductive `sorted : list elt \rightarrow Prop :=`
`| sorted_nil : sorted nil`
`| sorted_cons : $\forall a \text{ l}, \text{Inf } a \text{ l} \rightarrow \text{sorted } l \rightarrow \text{sorted } (a::l)$.`

Record `sorted_list : Set := sl {`
`content :> list elt;`
`content_sorted : sorted content`
`}`.

Definition `t := sorted_list.`

Remarques :

- ▶ les objets de type `sorted_list` sont des couples contenant une liste et une preuve,
- ▶ la notation `content :> list elt` permet d'omettre `content` quand le contexte est suffisamment clair

Exemple :

`$\forall (x:\text{elt}) (l:t), \text{In } x \text{ l}$` au lieu de `$\forall (x:\text{elt}) (l:t), \text{In } x \text{ l}.\text{(content)}$`

Un exemple avant de se lancer

Definition `elements (l:t) : list elt := l.(content).`

Definition `In_set : elt → t → Prop := fun x s ⇒ In x (elements s).`

Fixpoint `mem_aux (x:elt) (l:list elt) {struct l} : bool :=`
`match l with`
`| nil ⇒ false`
`| y::q ⇒`
`match 0.compare y x with`
`| Eq ⇒ true`
`| _ ⇒ mem_aux x q`
`end`
`end.`

Lemma `mem_aux_prop1 : ∀ x l, mem_aux x l = true → In x l.`

Proof. ... **Qed.**

Lemma `mem_aux_prop2 : ∀ x l, sorted l → In x l → mem_aux x l = true.`

Proof. ... **Qed.**

Definition `mem (x:elt) (l:t) : bool := mem_aux x l.`

Lemma `mem_prop : ∀ x s, In_set x s ↔ mem x s = true.`

Proof.

```
intros; unfold mem, In_set, elements.
destruct s as [l l_sorted]; simpl; split.
apply mem_aux_prop2; auto.
apply mem_aux_prop1.
```

Qed.