# Conteneurs de première classe en Coq

Stéphane Lescuyer

Proval
INRIA Saclay – Île-de-France




Coq est un langage de programmation.

$\longrightarrow$ on veut les structures de données de base dans la bibliothèque standard

Coq est un langage de programmation.

$\longrightarrow$ on veut les structures de données de base dans la bibliothèque standard

- ✓ types de base (entiers, booléens, chaînes, etc...)
- ✓ structures de données polymorphes (listes, streams, etc...)
- polymorphisme *ad-hoc* (AVL, listes triées, tables de *hash*...)

Coq est un langage de programmation.

→ on veut les structures de données de base dans la bibliothèque standard

- ✓ types de base (entiers, booléens, chaînes, etc...)
- ✓ structures de données polymorphes (listes, streams, etc...)
- polymorphisme *ad-hoc* (AVL, listes triées, tables de *hash*...)
  - ✓ modules et foncteurs (FSets/FMaps)
  - ? classes de types

Ensembles et dictionnaires finis à base de classes de types

# Plan

1. Préliminaires
   - Classes de types
   - Motivations
2. Types ordonnés
   - La classe `OrderedType`
   - Génération automatique
3. Ensembles finis
   - Interface et spécifications
   - Utilisation
4. Comparaison/Discussion
   - Performances, Partage, Paramétrisation...

# Classes de types (M. Sozeau)

Classe : paquet de définitions et de propriétés

```
Class decidable (A : Type) := {
  eq : A → A → bool;
  eq_dec : ∀xy, eq x y = true ↔ x = y
}.
```

# Classes de types (M. Sozeau)

**Classe : paquet de définitions et de propriétés**

```
Class decidable (A : Type) := {
  eq : A → A → bool;
  eq_dec : ∀xy, eq x y = true ↔ x = y
}.
```

**Instance : objet dont le type est une classe**

```
Definition bool_eq (x y : bool) := if x then y else negb y.
Property bool_eq_dec : ∀xy, bool_eq x y = true ↔ x = y.
Proof. .... Qed.
Instance bool_dec : decidable bool :=
  { eq := bool_eq; eq_dec := bool_eq_dec }.
```

# Classes de types (M. Sozeau)

## Paramétrisation implicite

```
Lemma decides_eq `{decidable A} : ∀(x y : A), {x=y}+{x≠y}.
Proof. .... Qed.
```

# Classes de types (M. Sozeau)

## Paramétrisation implicite

```
Lemma decides_eq `{decidable A} : ∀(x y : A), {x=y}+{x≠y}.
Proof. .... Qed.
```

## Inférence automatique d'instances

```
Instance prod_dec `{decidable A, decidable B} :
  decidable (A×B) := {
  eq := fun x y ⇒ eq (fst x) (fst y) && eq (snd x) (snd y);
  eq_dec := ...
}.
Check decides_eq (true, (false, true)) (false, (false, true)).
```

# Motivations

Par rapport aux modules/foncteurs :

1. Instantiation automatique
2. Surcharge
3. Efficacité et modularité
4. Instantiation à la volée

# Instantiation automatique

## Ensembles finis pour plusieurs types

```
Module IntSet := FSetList.Make Int.
Module IntPairSet := FSetList.Make IntPair.
Module BoolListSet := FSetList.Make BoolList.
```

# Instantiation automatique

## Ensembles finis pour plusieurs types

```
Module IntSet := FSetList.Make Int.
Module IntPairSet := FSetList.Make IntPair.
Module BoolListSet := FSetList.Make BoolList.

Module IntFacts := OrderedType.OrderedTypeFacts Int.
Module IntSetEqProps := FSetEqProperties.EqProperties IntSet.
Module IntSetProps := IntSetEqProps.MP.
Module IntSetFacts := IntSetEqProps.MP.Dec.FM.
Module IntPairFacts := OrderedTypFacts IntPair.
```

Il suffirait d'importer les modules une fois.

# Surcharge

## Qualification explicite

```
Definition g (x y : nat) (s t : IntSet.t) :=
  IntSet.In x (IntSet.union s (IntSet.remove y t)).
```

# Surcharge

## Qualification explicite

```
Definition g (x y : nat) (s t : IntSet.t) :=
  IntSet.In x (IntSet.union s (IntSet.remove y t)).

Lemma ....
Proof.
 ...
 apply IntSetEqProps.add_add_2.
 ...
 rewrite BoolListSetFacts.mem_iff
 ...
Qed.
```

## Paramètres implicites : aucune qualification nécessaire.

# Types ordonnés dans les FSets

Deux signatures alternatives `OrderedType` et `OrderedTypeAlt` :

```
Parameter t : Type.
Parameter eq : t → t → Prop.
Parameter lt : t → t → Prop.
(* equivalence axioms for eq *)
...
Axiom lt_trans : ...
Axiom lt_not_eq :
  ∀xy, lt x y → ~ eq x y.
Parameter compare :
  ∀xy, Compare lt eq x y.

Inductive Compare (X : Type) (lt eq : X → X → Prop) (x y : X)
| LT : lt x y → Compare lt eq x y
| EQ : eq x y → Compare lt eq x y
| GT : lt y x → Compare lt eq x y.
```

# Types ordonnés dans les FSets

Deux signatures alternatives `OrderedType` et `OrderedTypeAlt` :

```
Parameter t : Type.
Parameter compare : t → t → comparison.
Parameter compare_sym : ∀xy,
  compare y x =  match compare x y with
  | Eq ⇒ Eq | Gt ⇒ Lt | Lt ⇒ Gt end.
Parameter compare_trans :
  ∀cxyz, compare x y = c →
    compare y z = c → compare x z = c.
```

```
Inductive comparison := Eq | Lt | Gt.
```

# La classe OrderedType

## Le meilleur des deux alternatives

```
Class OrderedType (A : Type) := {
  _eq : relation A;
  _lt : relation A;
  OT_Equivalence :> Equivalence _eq;
  OT_StrictOrder :> StrictOrder _lt _eq;
  compare : A → A → comparison;
  compare_dec : ∀xy, compare_spec _eq _lt x y (compare x y)
}.
```

# La classe OrderedType

## Le meilleur des deux alternatives

```
Class OrderedType (A : Type) := {
  _eq : relation A;
  _lt : relation A;
  OT_Equivalence :> Equivalence _eq;
  OT_StrictOrder :> StrictOrder _lt _eq;
  compare : A → A → comparison;
  compare_dec : ∀xy, compare_spec _eq _lt x y (compare x y)
}.

Inductive compare_spec {A} eq lt (x y : A) : comparison → Prop
| compare_spec_lt : lt x y → compare_spec eq lt x y Lt
| compare_spec_eq : eq x y → compare_spec eq lt x y Eq
| compare_spec_gt : lt y x → compare_spec eq lt x y Gt.
```

# Opérations génériques sur les types ordonnés

| Notation | Signification | Vue |
|---|---|---|
| `x === y` | `_eq x y` | |
| `x =/= y` | `_eq x y -> False` | |
| `x <<< y` | `_lt x y` | |
| `x >>> y` | `lt y x` | |
| `x =?= y` | `compare x y` | `compare_dec` |
| `x == y` | `x =?= y is Eq` | `eq_dec` |
| `x << y` | `x =?= y is Lt` | `lt_dec` |
| `x >> y` | `x =?= y is Gt` | `gt_dec` |

# Opérations génériques sur les types ordonnés

| Notation | Signification | Vue |
|---|---|---|
| `x === y` | `_eq x y` | |
| `x =/= y` | `_eq x y -> False` | |
| `x <<< y` | `_lt x y` | |
| `x >>> y` | `lt y x` | |
| `x =?= y` | `compare x y` | `compare_dec` |
| `x == y` | `x =?= y is Eq` | `eq_dec` |
| `x << y` | `x =?= y is Lt` | `lt_dec` |
| `x >> y` | `x =?= y is Gt` | `gt_dec` |

- bibliothèque de propriétés, morphismes, tactique `order`, ...

# Égalités spécifiques

Comment restreindre l'égalité ?

```
... OrderedType with Definition eq := ...
```

# Égalités spécifiques

Comment restreindre l'égalité ?

`... OrderedType with Definition eq := ...`

```
Class SpecificOrderedType A eqA `(Equivalence eqA) := {
  SOT_lt : relation A;
  SOT_StrictOrder : StrictOrder SOT_lt eqA;
  SOT_compare : A → A → comparison;
  SOT_compare_spec : ∀xy,
    compare_spec eqA SOT_lt x y (SOT_compare x y)
}.
Instance SOT_as_OT `{SpecificOrderedType A}
  : OrderedType A := { ... }.
```

# Égalités spécifiques

## Comment restreindre l'égalité ?

```
... OrderedType with Definition eq := ...
```

```
Class SpecificOrderedType A eqA `(Equivalence eqA) := {
  SOT_lt : relation A;
  SOT_StrictOrder : StrictOrder SOT_lt eqA;
  SOT_compare : A → A → comparison;
  SOT_compare_spec : ∀xy,
    compare_spec eqA SOT_lt x y (SOT_compare x y)
}.
Instance SOT_as_OT `{SpecificOrderedType A}
  : OrderedType A := { ... }.
```

## Cas de l'égalité de Leibniz

```
Notation "'UsualOrderedType' A" :=
  (SpecificOrderedType A (@Logic.eq A) eq_equivalence).
```

# Génération automatique

## Instances de base et paramétriques

- `nat`, `positive`, `N`, `Z`, `Q`, `bool`, `unit`, ...
- `A + B`
- `A * B`
- `option A`
- `list A`

# Génération automatique

## Instances de base et paramétriques

- `nat`, `positive`, `N`, `Z`, `Q`, `bool`, `unit`, ...
- `A + B`
- `A * B`
- `option A`
- `list A`

```
Require Import OrderedTypeEx.

Goal ∀(x y : ((nat × bool) + (list Z × Q))),
  x == y = true ↔ x === y.
Eval vm_compute in (5 ::3 ::nil, true) =?= (nil, false).
```

# Génération pour inductifs

```
Inductive t := C1 : nat → bool → t | C2 : list Z → Q → t.
```

# Génération pour inductifs

```
Inductive t := C1 : nat → bool → t | C2 : list Z → Q → t.
```

Une commande vernaculaire `Generate OrderedType` :

- génère égalité, ordre et fonction de comparaison
- utilise des tactiques adéquates pour prouver les propriétés
- enregistre les instances
- fonctionne sur inductifs récursifs et mutuellement récursifs

# Génération pour inductifs

```
Inductive t := C1 : nat → bool → t | C2 : list Z → Q → t.
```

Une commande vernaculaire `Generate OrderedType` :

- génère égalité, ordre et fonction de comparaison
- utilise des tactiques adéquates pour prouver les propriétés
- enregistre les instances
- fonctionne sur inductifs récursifs et mutuellement récursifs

```
Inductive t := C1 : nat → bool → t | C2 : list Z → t → t.
Generate OrderedType t.
Check (∀(x y : t), x == y → x =?= y = Eq).
```

# La classe FSet : interface calculatoire

```
Class FSet `{H : OrderedType A} := {
  set : Type;
  In : A → set → Prop;
  empty : set;
  mem : A → set → bool;
  add : A → set → set;
  ...
  FSet_OT :> SpecificOrderedType _ (Equal_pw set A In) _
}.
Global Opaque set In empty mem add ... .
```

# La classe FSet : interface calculatoire

```
Class FSet `{H : OrderedType A} := {
  set : Type;
  In : A → set → Prop;
  empty : set;
  mem : A → set → bool;
  add : A → set → set;
  ...
  FSet_OT :> SpecificOrderedType _ (Equal_pw set A In) _
}.
Global Opaque set In empty mem add ... .
```

Un ensemble est aussi un type ordonné, pour l'égalité point à point

```
Definition Equal_pw (set elt : Type) (In : elt → set → Prop)
  (s s' : set) : Prop := ∀a : elt, In a s ↔ In a s'.
```

# Opérations et notations

L'interface calculatoire suffit pour définir un certain nombre de prédicats et de notations génériques sur les ensembles :

| | |
|---|---|
| `s [=] t` | `Equal s t` |
| `s [<=] t` | `Subset s t` |
| | `Empty s` |
| | `Exists P s` |
| | `Forall P s` |

| | |
|---|---|
| `v ∈ s` | `In v s` |
| `{}` | `empty` |
| `{v}` | `singleton v` |
| `{v ; s}` | `add v s` |
| `{s ~ v}` | `remove v s` |
| `v in s` | `mem v s` |
| `s ++ t` | `union s t` |
| `s \ t` | `diff s t` |

# Opérations et notations

L'interface calculatoire suffit pour définir un certain nombre de prédicats et de notations génériques sur les ensembles :

| | |
|---|---|
| `s [=] t` | `Equal s t` |
| `s [<=] t` | `Subset s t` |
| | `Empty s` |
| | `Exists P s` |
| | `Forall P s` |

| | |
|---|---|
| `v ∈ s` | `In v s` |
| `{}` | `empty` |
| `{v}` | `singleton v` |
| `{v ; s}` | `add v s` |
| `{s ~ v}` | `remove v s` |
| `v in s` | `mem v s` |
| `s ++ t` | `union s t` |
| `s \ t` | `diff s t` |

```
Goal ∀(x : A) (s : set A), Empty s → {x ; s} [=] {x}.
```

# Spécifications

Pour chaque opération, ses spécifications sont regroupées dans une classe.

# Spécifications

Pour chaque opération, ses spécifications sont regroupées dans une classe.

```
Class FSetSpecs_empty `(FSet A) := {
  empty_1 : Empty empty
}.
Class FSetSpecs_add `(FSet A) := {
  add_1 : ∀s x y, x === y → In y (add x s);
  add_2 : ∀s x y, In y s → In y (add x s);
  add_3 : ∀s x y, x =/= y → In y (add x s) → In y s
}.
...
```

# Spécifications

Pour chaque opération, ses spécifications sont regroupées dans une classe.

```
Class FSetSpecs_empty `(FSet A) := {
  empty_1 : Empty empty
}.
Class FSetSpecs_add `(FSet A) := {
  add_1 : ∀s x y, x === y → In y (add x s);
  add_2 : ∀s x y, In y s → In y (add x s);
  add_3 : ∀s x y, x =/= y → In y (add x s) → In y s
}.
...
```

- plus rapide
- donne un `SearchAbout` plus pertinent
- permet des spécifications partielles !

# Différents niveaux de spécification

- Aucune spécification : `FSet`

  $\Rightarrow$ un oracle pour une autre partie du système par exemple

# Différents niveaux de spécification

- Aucune spécification : `FSet`
  $\Rightarrow$ un oracle pour une autre partie du système par exemple
- Quelques opérations : `FSet` + `FSetSpecs_xxx` adéquats
  $\Rightarrow$ structures n'implémentant pas certaines opérations

# Différents niveaux de spécification

- Aucune spécification : `FSet`
  $\Rightarrow$ un oracle pour une autre partie du système par exemple
- Quelques opérations : `FSet` + `FSetSpecs_xxx` adéquats
  $\Rightarrow$ structures n'implémentant pas certaines opérations
- Spécifications complètes : `FSet` + `FSetSpecs`

```
Class FSetSpecs `(F : FSet A) := {
  FFSetSpecs_In :> FSetSpecs_In F;
  FFSetSpecs_mem :> FSetSpecs_mem F;
  FFSetSpecs_add :> FSetSpecs_add F;
  ...
}.
```

# Bibliothèque de propriétés

Les modules de propriétés de FSets/FMaps sont adaptés :

- SetFacts
- SetProperties
- SetEqProperties
- SetDecide (tactique `fsetdec` d'A. Bohannon)
- MapFacts

Les noms sont conservés ; certaines vues ajoutées.

```
Inductive choose_spec : option elt → Prop :=
| choose_spec_Some : ∀x (Hin : In x s), choose_spec (Some x)
| choose_Spec_None : ∀(Hempty : Empty s), choose_spec None.
Property choose_dec : choose_spec (choose s).
```

# Implémentations concrètes

## Ensembles finis

- listes triées (`SetList`)
- arbres AVL (`SetAVL`)
- arbres de Patricia (`SetPatricia`) sans preuves

## Dictionnaires finis

- listes d'associations triées (`MapList`)
- arbres AVL (`MapAVL`)
- arbres de préfixe pour entiers (`MapPositive`)
- arbres de Patricia (`MapPatricia`) sans preuves

# Utilisation pratique (Sets)

En chargeant le module `Sets` :
types ordonnés, instances, AVLs, propriétés

```
Require Import Sets.
Fixpoint fill n s :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill n0 {n0 ; s}
  end.
Eval vm_compute in mem 6 (fill 7 {42}).
```

# Utilisation pratique (Sets)

En chargeant le module `Sets` :
types ordonnés, instances, AVLs, propriétés

```
Require Import Sets.
Fixpoint fill n s :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill n0 {n0 ; s}
  end.
Eval vm_compute in mem 6 (fill 7 {42}).

Definition map_fill (s : set nat) : set (set nat) :=
  fold (fun n S ⇒ {fill n {} ; S}) s {}.
Eval vm_compute in cardinal (map_fill (fill 3 {})).
```

# Utilisation pratique (Sets)

En chargeant le module `Sets` :
types ordonnés, instances, AVLs, propriétés

```
Require Import Sets.
Fixpoint fill n s :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill n0 {n0 ; s}
  end.
Eval vm_compute in mem 6 (fill 7 {42}).

Definition map_fill (s : set nat) : set (set nat) :=
  fold (fun n S ⇒ {fill n {} ; S}) s {}.
Eval vm_compute in cardinal (map_fill (fill 3 {})).

Goal ∀(x : option nat) s, In x {x ; s}.
Proof. intro ; apply add_1 ; reflexivity. Qed.
```

# Utilisation pratique (Maps)

En chargeant le module `Maps` :
types ordonnés, instances, AVLs, propriétés

```
Require Import Maps

Fixpoint fill (s : Map[nat,nat]) (n : nat) :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill s[n0 <- S n0] n0
  end.
Eval vm_compute in (fill [] 7)[4]. (* renvoit 'Some 5' *)
```

# Utilisation pratique (Maps)

En chargeant le module `Maps` :
types ordonnés, instances, AVLs, propriétés

```
Require Import Maps

Fixpoint fill (s : Map[nat,nat]) (n : nat) :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill s[n0 <- S n0] n0
  end.
Eval vm_compute in (fill [] 7)[4]. (* renvoit 'Some 5' *)
```

Disponible pour Coq v8.2 à :
`http://www.lri.fr/~lescuyer/Containers.fr.html`

# Performances

Un module `BenchMark` pour comparer les versions module et typeclasses

$\Rightarrow$ les temps de calcul sont les mêmes !

# Performances

Un module `BenchMark` pour comparer les versions module et typeclasses

$\Rightarrow$ les temps de calcul sont les mêmes ! Satisfaisant malgré les paramètres de classes de types :

```
@add nat (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)
  (@SetAVLInstance.SetAVL_FSet nat
     (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT))
  1
  (@empty nat
     (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)
     (@SetAVLInstance.SetAVL_FSet nat
        (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)))
```

# Performances

Un module `BenchMark` pour comparer les versions module et typeclasses

$\Rightarrow$ les temps de calcul sont les mêmes ! Satisfaisant malgré les paramètres de classes de types :

```
@add nat (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)
  (@SetAVLInstance.SetAVL_FSet nat
     (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT))
  1
  (@empty nat
     (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)
     (@SetAVLInstance.SetAVL_FSet nat
        (@SOT_as_OT nat (@eq nat) (@eq_equivalence nat) nat_OT)))

 IntSet.add 1 IntSet.empty
```

# Mise à jour de code existant

La migration des modules vers les TC est très simple :

- réalisée sans souci sur un SAT solveur, sur CC(X) (20kloc)

# Mise à jour de code existant

La migration des modules vers les TC est très simple :

- réalisée sans souci sur un SAT solveur, sur CC(X) (20kloc)
- en pratique :
  - ✓ remplacer les modules `OrderedType` par les instances correspondantes (si besoin)
  - ✓ remplacer `BoolSet.t` par `set bool`
  - ✓ remplacer `destruct (compare x y)` par `destruct (compare_dec x y)`
  - ✓ "déqualifier" tous les objets qualifiés (opérations, lemmes, ...)

⇒ changement simple, code plus concis et plus lisible

# Partage de code

Comment éviter de tout maintenir en double ?

# Partage de code

Comment éviter de tout maintenir en double ?

1. réaliser le développement à base de classes de types
2. écrire les signatures de modules correspondantes
3. implémenter des foncteurs transformant les instances en modules

```
Module Type S.
  Parameter t : Type.
  Instance Ht : OrderedType t.
End S.
Module OT_to_FOT (Import X : S) <: OrderedType.
  Definition t := t.
  Definition eq : t → t → Prop := _eq.
  Definition lt : t → t → Prop := _lt.
  ...
  Definition compare : ∀x y, Compare lt eq x y := ...
End OT_to_FOT.
```

# Paramétrisation et taille des termes

On a écrit la classe FSet de la manière suivante :

```
Class FSet `{H : OrderedType A} := {
  set : Type;
  In : A → set → Prop;
  empty : set;
  mem : A → set → bool;
  add : A → set → set;
  ...
}.
```

# Paramétrisation et taille des termes

On aurait pu écrire la classe FSet sans paramètre :

```
Class FSet := {
  set : ∀A {OrderedType A}, Type ;
  In : ∀'{OrderedType A}, A → set A → Prop ;
  empty : ∀'{OrderedType A}, set A ;
  mem : ∀'{OrderedType A}, A → set → bool ;
  add : ∀'{OrderedType A}, A → set → set
  ...
}.
```

# Paramétrisation et taille des termes

On aurait pu écrire la classe FSet sans paramètre :

```
Class FSet := {
  set : ∀A {OrderedType A}, Type ;
  In : ∀'{OrderedType A}, A → set A → Prop ;
  empty : ∀'{OrderedType A}, set A ;
  mem : ∀'{OrderedType A}, A → set → bool ;
  add : ∀'{OrderedType A}, A → set → set
  ...
  map : ∀'{OrderedType A, OrderedType B},
          (A → B) → set A → set B
}.
```

Contrepartie : seules les structures "génériques" sont prises en compte.

# Paramétrisation et taille des termes

```
(* Sans paramètre *)
AVL : FSet
nat_OT : OrderedType nat
s : set nat
============================
add 5 s
```

---

```
(* Avec paramètre *)
AVL : ∀‘{H : OrderedType A}, FSet
nat_OT : OrderedType nat
s : set nat
============================
add 5 s
```

# Paramétrisation et taille des termes

```
(* Sans paramètre *)
AVL : FSet
nat_OT : OrderedType nat
s : set AVL nat nat_OT
============================
add AVL nat nat_OT 5 s
```

---

```
(* Avec paramètre *)
AVL : ∀‘{H : OrderedType A}, FSet
nat_OT : OrderedType nat
s : set nat
============================
add 5 s
```

# Paramétrisation et taille des termes

```
(* Sans paramètre *)
AVL : FSet
nat_OT : OrderedType nat
s : set AVL nat nat_OT
========================
add AVL nat nat_OT 5 s
```

---

```
(* Avec paramètre *)
AVL : ∀‘{H : OrderedType A}, FSet A H
nat_OT : OrderedType nat
s : set nat nat_OT (AVL nat nat_OT)
===================================
add nat nat_OT (AVL nat nat_OT) 5 s
```

# Paramétrisation et taille des termes

```
(* Sans paramètre *)
AVL : FSet
nat_OT : OrderedType nat
s : set AVL nat nat_OT
========================
add AVL nat nat_OT 5 s
```

---

```
(* Avec paramètre *)
AVL : ∀‘{H : OrderedType A}, FSet A H
nat_OT : OrderedType nat
s : set nat nat_OT (AVL nat nat_OT)
==========================
add nat nat_OT (AVL nat nat_OT) 5 s
```

- peut augmenter de 40% la taille des termes, et le typechecking !
- pourrait être résolu par *hash-consing*

# Pistes pour travail futur

La bibliothèque pourrait être améliorée de diverses façons :

- ajouter les notions d'ensembles/dictionnaires faibles
- nouvelles implémentations
  - preuves pour les ensembles de Patricia
  - *tries*

```
Instance Trie '(FMap A) : FMap (list A) := ...
```

- nouvelles classes
  - graphes
  - tableaux persistants
  - ...

# Conclusion

- une bibliothèque Coq de conteneurs fondée sur les classes de types
- ensembles/dictionnaires de première classe
- rend l'utilisation de structures de données plus simples
- assouplit considérablement l'architecture de gros développements

- les types ordonnés sont utiles en soi
- premier gros développement à base de classes de types
- pas parfait : impact de certaines décisions
- pas une critique des modules !