Conteneurs de première classe en Coq

Stéphane Lescuyer

Conteneurs de première classe en Coq

First-Class Containers in Coq

Stéphane Lescuyer

INRIA Saclay – Île-de-France – Projet ProVal,
91893 ORSAY CEDEX, France
stephane.lescuyer@inria.fr

RÉSUMÉ. *Nous présentons une bibliothèque Coq d'ensembles et de dictionnaires finis qui reproduit les fonctionnalités disponibles dans la bibliothèque existante* `FSets/FMaps` *mais où la généricité des structures est obtenue via des classes de types et non des modules. Cette architecture permet un usage simplifié de ces structures et facilite la programmation d'algorithmes complexes en Coq.*

ABSTRACT. *We present a Coq library for finite sets and maps which brings the same functionalities as the existing* `FSets/FMaps` *library, but uses type-classes instead of modules in order to ensure the genericity of the proposed data structures. This architecture facilitates the use of these data structures and more generally the implementation of complex algorithms in Coq.*

MOTS-CLÉS : *classes de types, ensembles finis, dictionnaires finis, Coq*

KEYWORDS: *typeclasses, finite sets, finite maps, Coq*

1. Introduction

Les assistants de preuve tels Coq [4] ou Isabelle/HOL [8] permettent d'exprimer et de prouver formellement des propriétés logiques complexes, mais aussi plus généralement de définir des types et des fonctions sur lesquelles il est ensuite possible de raisonner. Ce sont donc en particulier des langages de programmation et c'est une des caractéristiques qui les rend si polyvalents : on peut y définir des programmes, les spécifier formellement, et éventuellement les *extraire* [10] vers un langage de programmation usuel ou bien les exécuter directement au sein de l'assistant de preuve. Cette dernière fonctionnalité est à la base des preuves par *réflexion* [2] et fournit un moyen très puissant d'améliorer l'automatisation des preuves interactives.

En tant que langage de programmation, il est naturel de doter Coq de bibliothèques de structures de données génériques. En effet, les langages de programmation disposent habituellement de bibliothèques pour manipuler les structures qui reviennent couramment : séquences, ensembles finis, tables d'associations, etc. Par exemple, les programmeurs C++ peuvent s'appuyer sur la bibliothèque STL [13] (*Standard Template Library*), tandis que les programmeurs OCaml ont à leur disposition un certain nombre de modules (listes, files, ensembles et dictionnaires finis, tables de hachage, ...) dans la bibliothèque standard [1]. La généricité des structures proposées, i.e. le fait qu'on puisse les utiliser pour stocker des éléments de types quelconques, est obtenue de manière différente suivant les langages : polymorphisme dans les langages ML, *templates* en C++ ou encore *generics* en Java.

L'implémentation de certaines structures de données efficaces requiert des propriétés particulières sur les éléments, par exemple une fonction de comparaison. Ce type de généricité, appelé polymorphisme *ad hoc*, est rendu possible par l'usage de *foncteurs* en OCaml, et par celui de *classes de types* en Haskell [14]. Même s'ils peuvent être utilisés pour répondre au même problème, ces deux paradigmes diffèrent fondamentalement et ont chacun des avantages sur l'autre [15]. Depuis plusieurs années déjà, Coq dispose d'un système de modules similaire à celui d'OCaml [3] et P. Letouzey et J-C. Filliâtre l'ont mis à profit pour développer une bibliothèque très complète d'ensembles et de

dictionnaires finis [5]. Nous avons utilisé cette bibliothèque `FSets` de manière conséquente au sein d'un développement Coq [9] et avons rencontré un certain nombre de problèmes inhérents aux modules. Or, Coq a été récemment doté d'un système de classes de types fondé sur les enregistrements dépendants [12] et nous avons décidé de tirer profit de cette nouvelle fonctionnalité pour réimplanter la bibliothèque existante `FSets` à base de classes de types. C'est cette librairie que nous présentons et discutons dans cet article.

La section 2 présente rapidement le système de classes de types de Coq, ainsi que les problèmes ayant motivés ce travail. Nous introduisons ensuite en section 3 la notion de base de la librairie, à savoir les types ordonnés, avant de décrire comment les structures d'ensembles et de dictionnaires finis proprement dites ont été adaptées (section 4). Nous donnons ensuite quelques instantiations concrètes de ces structures, avant de comparer plus précisément notre librairie et la version modulaire en section 6.

2. Préliminaires et motivations

2.1. *Classes de types*

Nous présentons dans cette section le système de classes de types de Coq dans ses grandes lignes ; pour une présentation détaillée, le lecteur intéressé peut se reporter à [12].

Une classe de types permet d'empaqueter un certain nombre de définitions et de propriétés, à la manière d'un enregistrement[1]. Ces classes peuvent être paramétrées, ainsi on peut définir la classe des types disposant d'une égalité décidable :

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

1. et les classes sont d'ailleurs implantées par des enregistrements.

Cette classe `decidable` est paramétrée par un type `A` et contient une égalité booléenne sur ce type `A`, ainsi que les preuves adéquates sur cette égalité. Des objets de type `decidable` sont appelés des *instances* et sont définis de manière particulière à l'aide du mot-clé **Instance** :

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

Les champs peuvent aussi être initialisés directement ou prouvés interactivement. Les classes de types prennent tout leur intérêt par la conjonction de deux mécanismes :

– la possibilité de définir des objets paramétrés par des classes de types et de les utiliser sans instancier explicitement ces paramètres ;

– le mécanisme d'inférence automatique d'instances de classes de types à partir des instances déjà définies par l'utilisateur.

Par exemple, on peut prouver le lemme suivant pour tout type ayant une instance de `decidable` :

Lemma `decides_eq ‘{decidable A} :`
$\forall$`(`x y `: A),` `{`x `=` y`} + {`x $\neq$ y`}`.
Proof. .... **Qed**.

Ce lemme est paramétré par un type `A` et une instance de `decidable A` mais ces paramètres sont déclarés implicites. Lorsqu'on utilise ensuite ce lemme en l'appliquant à deux termes d'un même type `B`, une instance de `decidable B` est automatiquement cherchée à partir des définitions d'instances déjà réalisées. Par exemple, on peut écrire le terme `decides_eq true false` qui sera bien typé et va utiliser implicitement et automatiquement l'instance `bool_dec` définie auparavant.

L'inférence automatique d'instances est particulièrement intéressante lorsqu'on définit des familles d'instances, ou des instances paramétrées par d'autres instances. Par exemple, on peut définir une instance pour tout type produit `A` $\times$ `B` à partir d'instances pour les types `A` et `B`,

et le système pourra alors automatiquement inférer une instance pour tout produit de types décidables :

```
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))).
```

Pour finir, on peut construire des hiérarchies de classes, et un système de coercions automatiques garantit qu'une instance d'une classe peut être utilisée comme instance de ses sous-classes. Nous verrons un tel exemple dans la section 3.

2.2. *Motivations*

À la lumière des fonctionnalités offertes par le système de classes de types, nous revenons ici en détail sur les motivations qui nous ont conduits, pour notre développement de solveur SAT décrit en [9], à utiliser ce système à la place de l'implantation modulaire `FSets` déjà disponible.

Instantiation automatique. Dans notre développement, nous manipulons des ensembles de nombreux types différents, y compris des ensembles d'ensembles, et pour chaque nouveau type le foncteur qui crée un module d'ensembles finis doit être appliqué manuellement afin de créer le module adéquat. Ainsi, étant donné des modules de types ordonnés `Int`, `IntPair`, `BoolList` pour les entiers, les paires d'entiers et les listes de booléens, on devra écrire[2] :

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

2. `FSetList.Make` est un foncteur créant une structure d'ensemble fini à base de liste ordonnée.

pour pouvoir utiliser des ensembles sur ces trois types d'éléments. Cela peut sembler un moindre mal mais on est très vite amenés à instancier également les foncteurs de propriétés sur les types ordonnés et les ensembles finis, qui sont indispensables pour raisonner sur les structures ci-dessus.

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

Ce type de définitions, que tout utilisateur des `FSets` a rencontré, devient très rapidement pénible à lire et à maintenir et d'autre part les applications de ces foncteurs ne sont pas gratuites et il n'est pas rare de passer une poignée de secondes uniquement pour l'instantiation de tous ces objets. L'instantiation automatique et implicite des instances de classes de type apporte une solution à ce problème.

Surcharge. Comme le système de modules n'offre pas de surcharge réelle, on fait référence aux membres d'un module en les qualifiant du nom du module. Dans le cas ci-dessus, cela signifie que toute utilisation d'une fonction, d'un lemme ou d'un type apportés par ces modules (`IntSet, IntPairSet, IntSetProps,` ...) doit être proprement qualifiés. Cela rend très vite le code très verbeux et peu lisible, et les scripts de preuve très peu compacts. L'utilisateur finit souvent par donner des noms très courts à tous ces modules (`IS, IPS, ISP, ...`) et perd alors en lisibilité ce qu'il a gagné en verbosité. Par le biais des arguments de classe implicites, le système de classes de types dispense d'une telle qualification et permet une vraie surcharge des opérateurs.

Efficacité et modularité. Afin de garantir une bonne modularité dans notre développement, certaines parties du système doivent être paramétrées par des modules apportant des types et certaines structures de données sur ces types. Ainsi, il n'est pas rare en OCaml d'écrire des signatures de ce genre :

```
(* some abstract type *)
type t
(* finite sets of elements of type t *)
module TS = Set.S with type elt = t
(* finite maps indexed by elements of type t *)
module TM = Map.S with type key = t
...
```

En paramétrisant le développement de cette manière dans Coq, nous nous sommes heurtés à un problème d'efficacité des modules, à savoir que l'application de foncteurs paramétrés par de telles signatures prenait un temps déraisonnable. Ainsi, une application du foncteur final de notre système prenait de l'ordre de 15 secondes. Si cela semble être un problème d'implémentation[3] de l'application des foncteurs (à savoir, par substitution) plutôt qu'un problème inhérent aux modules, il n'en reste pas moins que c'est un problème pratique très limitant et que les classes de types ne souffrent pas de cette limitation.

Première classe. Pour approfondir le point précédent, les instances de classes de type de Coq sont de fait des valeurs de première classe, et donc le coût d'une instantiation est réduit au typage de l'argument (c'est juste le passage d'un argument à une fonction). Elles permettent donc d'envisager une instantiation interactive de procédures paramétrées par des classes de types. Par exemple, l'invocation d'une tactique réflexive peut commencer par générer une instance dépendant du contexte (comme une formule réifiée ou une table de hashconsing) et la passer à une procédure paramétrée. Ce type d'instantiation dynamique est inconcevable avec un foncteur puisque appliquer le foncteur à chaque invocation de la tactique serait rédhibitoire.

Parmi ces motivations, les deux premiers points sont inhérents aux modules et aux classes de types tandis que les deux derniers dépendent plutôt de l'implémentation. Bien que le troisième soit vraiment celui qui nous a décidé à changer de paradigme, les avantages soulignés par les deux premiers points sont déjà une raison suffisante en pratique.

3. en particulier, utiliser les ensembles finis à base d'arbres AVL plutôt que de listes ordonnées multipliait ce temps par quatre sans raison apparente.

3. Types ordonnés

Les structures d'ensembles et de dictionnaires finis, pour être implantées efficacement, requièrent que le type des éléments soit un type muni d'un ordre total décidable. Nous montrons dans cette section comment nous formalisons cette classe de types.

3.1. *OrderedType*

Un type ordonné est un type muni d'une égalité (une relation d'équivalence), d'un ordre strict (une relation transitive et irréflexive) et telle que ces relations sont décidables. Coq fournit déjà une classe `Equivalence` pour les relations d'équivalence, ainsi que les notations `x === y` (resp. `x =/= y`) pour les égalités (resp. diségalités) sur ces relations. Nous définissons la classe des ordres stricts modulo une relation d'équivalence. Cette classe est paramétrée par le type des éléments, la relation d'équivalence et la relation d'ordre :

```
Class StrictOrder {A} lt eq {Equivalence eq} := {
  StrictOrder_Transitive :
    ∀(x y z : A), lt x y → lt y z → lt x z ;
  StrictOrder_Irreflexive :
    ∀(x y : A), lt x y → x =/= y
}.
```

La bibliothèque `FSets` amène deux signatures alternatives pour les types ordonnés, respectivement dans les modules `OrderedType` (figure 1) et `OrderedTypeAlt` (figure 2).

`OrderedType` apporte un type `t`, une relation d'équivalence `eq` et un ordre strict `lt` sur `t`, ainsi que les preuves adéquates. La décidabilité est donnée par la fonction `compare` qui est complètement spécifiée par son type de retour, défini par l'inductif `Compare` : la fonction `compare` permet donc de comparer deux éléments et retourne également une preuve de la relation de comparaison entre ces deux éléments. Cette formalisation est assez pratique à utiliser, en particulier lorsqu'on raisonne par cas sur la comparaison entre deux éléments, les hypothèses correspondant à chaque branche sont ajoutées naturellement. Un inconvénient

```
Inductive Compare {A} lt eq x y :=
| 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.

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.
```

Figure 1 – Module `OrderedType` existant

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

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.
```

Figure 2 – Module `OrderedTypeAlt` existant

possible est que la fonction `compare` est informative et non purement calculatoire, or elle est utilisée intensément dans les calculs et cela peut être source d'inefficacité. Pour assurer une séparation calculs/preuves, `OrderedTypeAlt` est au contraire centré sur une fonction de comparaison dont le type de retour `comparison` est le type à trois valeurs `Eq | Lt | Gt`. La spécification de cette fonction par les propriétés de symétrie et de transitivité est en revanche assez pénible à utiliser dans les phases de raisonnement.

Afin de garder le meilleur de ces deux alternatives, nous choisissons une fonction de comparaison purement calculatoire, et nous la spécifions séparément à l'aide de l'inductif suivant :

```
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.
```

Contrairement à `Compare`, cet inductif n'est pas le type de retour de la fonction de comparaison, mais relie chaque valeur de comparaison à l'hypothèse adéquate. Il suffit alors de prouver que tous les résultats de la fonction de comparaison vérifient cette relation pour l'avoir spécifiée correctement. Nous sommes alors en mesure de donner la classe `OrderedType` des types ordonnés :

```
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)
}.
```

Cette classe est paramétrée par le type `A` des éléments et contient les relations d'égalité et d'ordre strict. Des sous-classes `Equivalence` et `StrictOrder` (introduites par `:>`) sont utilisées pour spécifier ces relations. Enfin, la fonction de comparaison et sa spécification sont introduites comme expliqué ci-dessus. Cette spécification est aussi simple à utiliser que l'originale malgré le type non informatif de la fonction `compare` : en effet, dans un contexte où `compare a b` apparaît, il suffit d'invoquer `destruct (compare_dec a b)` pour raisonner par cas sur cette comparaison : `compare a b` est alors remplacé dans chaque branche par sa valeur, et l'hypothèse correspondante est ajoutée au

contexte. L'inductif `compare_spec` est similaire aux "vues" réflexives de l'extension SSReflect [7].[4]

Une fois les classes définies, de nombreux lemmes utiles (comme le fait que l'ordre est un morphisme pour l'égalité) et des notations sont établis et peuvent être utilisés pour n'importe quel type ordonné. La table suivante récapitule les notations disponibles et les "vues" correspondantes :

Notation	Signification	Vue
`x =?= y`	`compare x y`	`compare_dec`
`x == y`	`true` ssi `x =?= y` renvoie `Eq`	`eq_dec`
`x << y`	`true` ssi `x =?= y` renvoie `Lt`	`lt_dec`
`x >> y`	`true` ssi `x =?= y` renvoie `Gt`	`gt_dec`

3.2. *Égalités particulières*

Lorsqu'on écrit du code paramétré par un type ordonné, il est très fréquent de requérir qu'un certain type soit ordonné en contraignant la relation d'égalité à être une égalité particulière, et très souvent l'égalité de Leibniz. Le système de modules permet d'exprimer une telle contrainte via la restriction de signature `OrderedType with Definition eq := ...`. Malheureusement, ce type de contraintes n'est pas exprimable avec les classes de types ; pour ce faire, il faut que la partie que l'on veut contraindre soit un paramètre de la classe de type et non un membre. Pour traiter ces cas particuliers, on introduit une classe particulière `SpecificOrderedType` paramétrée par la relation d'équivalence souhaitée, et on montre que tout type de cette classe est aussi un `OrderedType` :

```
Class SpecificOrderedType (A : Type)
  (eqA : relation A) `(Equivalence A eqA) := {
  SOT_lt : relation A ;
  SOT_StrictOrder : StrictOrder SOT_lt eqA ;
  SOT_compare : A → A → comparison ;
```

4. cette discussion se rapporte à la version 8.2 de Coq ; la prochaine version devrait introduire une signature "mixte" tirant profit des classes de types et de `compare_spec`.

```
  SOT_compare_spec :
    ∀xy, compare_spec eqA SOT_lt x y (SOT_compare x y)
}.
Instance SOT_as_OT `{SpecificOrderedType A} :
  OrderedType A := { ... }.
```

On ajoute également une notation `UsualOrderedType` pour dénoter le cas particulier fréquent de l'égalité de Leibniz. Ces types ordonnés à égalité spécifique seront utiles pour définir les conteneurs dans la section 4.

3.3. *Génération d'instances*

Après que les classes et les propriétés génériques ont été définies, nous déclarons des instances d'`OrderedType` pour tous les types de bases et les constructeurs de type usuels. Lorsque c'est possible, nous déclarons des instances d'`UsualOrderedType`, y compris pour les constructeurs de type[5]. La librairie fournit des instances pour : les entiers Peano, les entiers binaires (strictement positifs, naturels et relatifs), les rationnels, les booléens, les listes, les produits, les sommes et les options. Ainsi, les fonctions génériques sur les types ordonnés peuvent être utilisées sur toutes les combinaisons de ces constructeurs et types de bases sans aucune intervention manuelle, grâce à l'inférence automatique des classes de types :

```
Goal ∀(x y : ((nat × bool) + (list Z × Q))), x === y.
```

Un type comme ci-dessus sera fréquemment défini directement comme un inductif à deux branches

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

Le système des classes de types ne peut pas inférer d'instances automatiquement pour des types inductifs, mais nous avons implanté en

5. par exemple, si `A` et `B` sont des types ordonnés pour l'égalité de Leibniz, alors leur produit ou leur somme également.

OCaml une commande vernaculaire qui peut traiter ce genre de cas automatiquement. Cette commande `Generate OrderedType` prend un type inductif en argument, génère relations d'équivalence et d'ordre strict, fonction de comparaison et toutes les preuves nécessaires avant de déclarer l'instance de type ordonné correspondant. Elle utilise pour ce faire toutes les instances déjà disponibles dans l'environnement. Les constructeurs sont ordonnés arbitrairement, et les paramètres d'un même constructeur sont ordonnés lexicographiquement[6]. La commande fonctionne avec les inductifs (mutuellement) récursifs et les paramètres uniformes, ce qui en fait un ajout très utile à la bibliothèque. Par exemple, les commandes suivantes montrent son utilisation pour comparer des chaînes de caractères :

```
Generate OrderedType ascii.
Generate OrderedType string. (* string utilise ascii *)
Eval vm_compute in (“longue” =?= “petite”).
(* ce calcul renvoie Lt *)
```

4. Ensembles et dictionnaires finis

Les types ordonnés décrits dans la section 3 sont une classe de types sur lesquels il est possible d'implémenter un certain nombre de structures efficaces de conteneurs de données. Le but de la bibliothèque est de fournir ce type de structure, nous définissons maintenant leur interface.

4.1. *Interfaces et spécifications*

La classe des ensembles finis contenant des éléments d'un type ordonné `A` est définie de la manière suivante :

```
Class FSet ‘{H : OrderedType A} := {
  set : Type ;
  In : A → set → Prop ;
```

6. mais la commande est typiquement utilisée dans les cas où n'importe quel ordre bien défini convient, à l'instar de `Pervasives.compare` en OCaml.

```
  empty : set ;
  mem : A → set → bool ;
  add : A → set → set ;
  ...
  FSet_OrderedType :>
    SpecificOrderedType _ (Equal_pw set A In) _
}.
```

Implicit Arguments `set [[H] [FSet]].`

Cette classe paramétrée[7] par le type `A` et une instance d'`OrderedType A` apporte le type `set` des ensembles finis de `A` et les différents opérateurs disponibles sur ces ensembles. Le champ `In` est le prédicat d'appartenance à l'ensemble et est la seule partie propositionnelle de cette classe : tous les opérateurs vont être spécifiés en fonction de ce prédicat. Le champ `FSet_OrderedType` nécessite des explications : il garantit que le type `set` des ensembles d'éléments de `A` est lui-même un type ordonné, qui plus est un type ordonné pour une égalité bien particulière, en introduisant une sous-classe `SpecificOrderedType` comme expliqué en section 3.2. Cette égalité est l'égalité point à point pour l'opérateur d'appartenance `In`, définie ainsi pour tout type de conteneur `ctr` et d'éléments `elt` :

Definition `Equal_pw (ctr elt : Type)`
`(In : elt → ctr → Prop) (`s s' `: ctr) : Prop :=`
$\forall a$ `: elt, In` a $s \leftrightarrow$ `In` a s'.

Ces définitions permettent de considérer des ensembles comme des types ordonnés, en écrivant par exemple `s === empty`, et assure que cette égalité est convertible avec l'égalité point à point, qui est celle utilisée dans la librairie `FSets` originalement. Étant donné une instance de `FSet` pour un type ordonné `A`, on peut alors manipuler des ensembles facilement :

Definition `add_all (`x y z `: A) (`s `: set A) :=`
`add` x `(add` y `(add` z s`)).`

7. ce choix, loin d'être bénin, est discuté en section 6.

Lors des manipulations de preuve, il n'est pas souhaitable que l'instance de `FSet` utilisée soit dévoilée à l'utilisateur, i.e. que les projections `set`, `add`, etc, soient réduites. Pour cela, nous rendons opaques les différentes projections de la classe `FSet`[8] :

Global Opaque `set In empty mem ... .`

La classe `FSet` ne contient que l'interface calculatoire de la structure d'ensembles finis et non sa spécification. Le choix a été fait de séparer interface et spécifications pour des raisons pragmatiques : les définitions de fonctions et d'algorithmes peuvent se contenter de l'interface calculatoire, qui reste ainsi relativement petite, tandis que seules les phases de preuve nécessiteront les spécifications. Avant de définir ces spécifications, nous pouvons déja définir un certain nombre de prédicats et de notations génériques sur les ensembles finis, parmi lesquelles `Equal s t` pour l'égalité point à point et `Subset s t` pour l'inclusion. Les notations disponibles sont énumérées dans la table 3.

`s [=] t`	`Equal s t`
`s [<=] t`	`Subset s t`
`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`

Figure 3 – Notations disponibles pour les ensembles finis

L'ensemble des spécifications des opérations fournies par la classe `FSet` pourrait être empaqueté dans une grande classe `FSetSpecs` paramétrée par `FSet`, mais nous choisissons plutôt de spécifier chaque

8. cela ne bloque pas le calcul avec `compute` ou `vm_compute`, mais les δ-conversions réalisées par certaines tactiques.

opération dans une classe à part. Ainsi, les spécifications des opérations `empty` et `add` sont décrites par les classes suivantes :

```
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
}.
```

Nous faisons ce choix pour deux raisons. Tout d'abord, il est très fréquent en pratique d'interroger le système sur les lemmes disponibles à propos d'un certain identificateur, par exemple `add`, à l'aide de la commande **SearchAbout** `add` par exemple. Si toutes les spécifications (une cinquantaine) sont regroupées dans une seule classe, cette commande va malheureusement afficher les objets énormes que sont le constructeur de la classe et les principes d'élimination de cette classe, ce qui la rend inutilisable. En regroupant les spécifications par fonction, seuls les lemmes pertinents sont affichés par la commande. L'autre raison est que ce choix permet en général de n'avoir des preuves qui ne dépendent que du strict nécessaire : ainsi, si une structure d'ensemble particulière ne fournit pas certaines opérations d'ensembles mais que celles-ci ne sont pas utilisées, un développement peut tout de même s'appuyer sur l'interface générique puisque seules les spécifications de certaines opérations sont manquantes. A l'extrême on peut imaginer un développement sur lequel aucune preuve n'est nécessaire (qui est utilisée comme oracle pour une autre partie du système) et qui ne demanderait aucune spécification mais se contenterait d'utiliser l'interface `FSet`. Pour les développements qui nécessiteraient au contraire l'interface complète de toutes les opérations, on définit une classe qui est une superclasse de toutes les spécifications :

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

```
  ...
}.
```

Conjointement, cette classe de spécifications et la classe `FSet` correspondent exactement à l'interface `FSetInterface.S` de la librairie existante.

Dictionnaires finis. Nous ne présentons ici que l'interface des ensembles finis pour des raisons de concision, mais l'interface des dictionnaires finis est adaptée de manière similaire à partir de celle de la bibliothèque existante. En particulier, les mêmes choix sont faits quant à la séparation interface/spécifications et la modélisation des spécifications.

4.2. *Les bibliothèques de propriétés*

La librairie `FSets` contient plusieurs modules de résultats généraux sur les structures d'ensembles finis : `FSetFacts, FSetDecide, FSetProperties, FSetEqProperties`. L'adaptation de ces modules aux interfaces à base de classes de type présentées précédemment représentait un bon test pour étalonner la facilité d'usage de cette présentation. Tous ces modules ont été adaptés sans problème majeur, le point le plus délicat étant certainement `FSetDecide` et sa tactique `fsetdec` contribuée par A. Bohannon. Il faut noter que, alors que la tactique originale s'intéressait à un seul type d'ensemble à la fois, la tactique de notre librairie traite toutes les hypothèses se référant à des ensembles finis dans le contexte et cela peut conduire à quelques incompatibilités. Tous les lemmes et toutes les propriétés ont gardé le même nom que dans la librairie originale, ce qui minimise la quantité de travail nécessaire pour porter un code de la version modulaire à la version présentée ici (cf. Section 6.3).

Un certain nombre de propriétés ont été ajoutées pour faciliter le raisonnement par cas sur des fonctions comme `mem`, `choose` ou `min_elt`, à l'aide de vues inductives de leur spécification. Par exemple, la spécification de la fonction `choose` est disponible de cette façon :

```
Inductive choose_spec (s : set elt) :
  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 : ∀s, choose_spec (choose s).
```

et s'utilise facilement en raisonnant par cas sur le résultat de `choose_dec`.

5. Applications

5.1. *Listes et arbres AVL*

La bibliothèque existante `FSets` propose deux types d'implantations d'ensembles et de dictionnaires finis, les unes fondées sur des listes triées et les autres sur des arbres de recherche équilibrés (AVL) [6].

Nous avons transposé les ensembles et les dictionnaires à base de listes, ainsi que ceux à base d'AVL. Détaillons par exemple le cas des ensembles à base de listes. En pratique, le développement des listes triées est le même dans la version modulaire et dans notre version à classes de types et ils ne diffèrent que marginalement[9]. Ainsi, alors que le code original des listes est un foncteur paramétré par un module de signature `OrderedType`, le code des listes triées à base de classes de type est en fait paramétré par rapport à une instance de la classe `OrderedType`. Cela se fait en utilisant le mécanisme de sections de Coq et la commande `Context` qui permet d'introduire des variables d'instance dans une section :

9. il est alors naturel de s'interroger sur la problématique de duplication de code entre les deux développements ; nous discutons ce point dans la section 6.

Version modulaire	Version à base de classes

Module `Make (X : OrderedType)`
 `  <: S` **with Module** `E := X.`
 `  Module E := X.`
 `  Definition elt := X.t.`
 `  ...`
 End `Make.`
	Section `Make.`
	 `  Context ‘{OrderedType elt}.`
	 `  ...`
	 End `Make.`

Dans le contexte `‘{OrderedType elt}`, `elt` est un type frais et disposant d'un ordre décidable. Les définitions de la section peuvent donc utiliser `elt` comme un type ordonné, et elles sont automatiquement généralisées pour tout type ordonné à la fermeture de la section.

Une fois que les définitions des listes triées et des différentes opérations, ainsi que les preuves adéquates, ont été réalisées, il ne reste plus qu'à définir les instances correspondant aux classes présentées dans la section 4.1. On peut empaqueter toutes ces définitions dans un module particulier `SetList`, qui n'a pas à être importé par l'utilisateur, puisque seules les instances implémentant l'interface sont nécessaires. Dans le cas des listes triées, qui fournissent une structure d'ensembles pour n'importe quel type ordonné, on peut définir une instance générique paramétrée par un type ordonné :

```
Instance SetList_FSet ‘{Helt : OrderedType elt} :
  FSet := {
  set := SetList.set elt ;
  In := @SetList.In elt Helt ;
  empty := ...
}.
```

Cette définition déclare bien une famille d'ensembles finis, ou autrement dit une manière d'obtenir une structure d'ensemble fini sur `elt` pour tout type ordonné `elt`. De même, on déclare une famille de spécifications pour chacune de ces structures indexées par un type ordonné `elt` :

```
Instance SetList_FSetSpecs ‘{Helt : OrderedType elt} :
  FSetSpecs SetList_FSet := {
```

```
  FFSetSpecs_In := ... ;
  FFSetSpecs_mem := ... ;
  ...
}.
```

Une fois ces instances définies dans un fichier (ou plus généralement un module), il suffit d'importer ce module et on peut alors utiliser des ensembles finis sur n'importe quel type ordonné.

5.2. *Utilisation*

La simplicité d'utilisation de notre bibliothèque est un de ses principaux intérêts. Pour travailler avec des ensembles finis, il suffit d'importer le module `Sets` qui exporte les fonctionnalités suivantes :

– la notion de type ordonné, ainsi que la bibliothèque de résultats et d'instances associés ;

– les instances génériques des ensembles finis à base d'arbres AVL ;

– les interfaces, spécifications, notations et propriétés relatives aux ensembles finis.

Une première remarque est que ce sont donc les AVL qui sont chargés par défaut lorsqu'on importe ce fichier. Cela est justifié par le fait que les AVL sont plus efficaces en général que les listes triées, et il n'y a aucune pénalité en terme de temps de chargement par rapport à l'utilisation des listes triées. Cela diffère de la bibliothèque existante où l'application des foncteurs d'arbres AVL est beaucoup plus longue que celle des foncteurs de listes triées. Un utilisateur qui souhaiterait malgré tout utiliser les listes triées peut se contenter de charger les instances adéquates ; il peut également spécifier manuellement quelle instance utiliser si les circonstances l'imposent[10]. Un module `Maps` existe également qui charge de manière similaire toute l'infrastructure nécessaire pour utiliser les dictionnaires finis ; il charge en particulier les dictionnaires à base d'AVL.

10. le gain en verbosité par rapport aux modules peut alors disparaître, mais cette instantiation explicite peut être évitée le plus souvent.

Une fois `Sets` chargé, on peut utiliser toutes les définitions et notations génériques sur les ensembles finis, en les utilisant sur des types ordonnés. Si de plus les instances d'`OrderedType` peuvent être inférées automatiquement comme décrit à la section 3.3, ce qui est le cas pour la plupart des types habituels, alors l'utilisation d'ensembles finis devient tout à fait transparente pour l'utilisateur, à l'instar de structures polymorphes comme les listes. L'exemple suivant montre le calcul d'un ensemble d'entiers :

```
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}).
(* ce calcul renvoie 'true' *)
```

Des ensembles finis de différents types peuvent cohabiter sans problème dans les mêmes fonctions, et en particulier grâce au membre `FSet_OrderedType` de la classe `FSet` (cf. 4.1), on peut manipuler des ensembles d'ensembles :

```
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 {})).
(* ce calcul renvoie 3 *)
```

L'utilisation des lemmes de la bibliothèque se fait de manière tout aussi facile lors des preuves, par exemple pour appliquer la première partie de la spécification de l'opération `add`, il suffit d'appliquer le lemme et les arguments implicites sont correctement inférés :

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

Pour conclure cette section, voilà un exemple d'utilisation des dictionnaires finis et de quelques notations associées. Le type des dictionnaires

associant des valeurs de type `elt` à des clés `key` est `Map[key, elt]`. La notation `s[k ← v]` dénote l'ajout (ou la mise à jour) d'une association dans le dictionnaire `s`, `[]` le dictionnaire vide et `s[k]` la valeur associée à `k` dans `s` si elle existe.

```
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].
(* ce calcul renvoie 'Some 5' *)
```

La bibliothèque est disponible à l'adresse suivante : `http://www.lri.fr/~lescuyer/Containers.fr.html`.

6. Discussion

Dans cette section, nous revenons sur la comparaison entre notre bibliothèque et celle existante, et discutons un certain nombre de choix et de limites dans l'implantation actuelle.

6.1. *Performances*

Afin de comparer les performances de notre bibliothèque par rapport à l'implantation à base de modules, la bibliothèque comprend un fichier `BenchMarks.v` qui teste les fonctions de base des ensembles finis. Le test consiste en la création d'un ensemble d'entiers à partir d'une séquence générée pseudo-aléatoirement, puis de tests d'appartenance dans l'ensemble obtenu. Ce processus est répété pour les ensembles à base de classes de types et ceux à base de modules. Le résultat est satisfaisant puisque lorsque les fonctions de comparaison des éléments

sont les mêmes[11], les deux alternatives calculent exactement à la même vitesse.

Pour comprendre pourquoi le simple fait que les performances sont conservées est satisfaisant, il faut remarquer que le côté très pratique et concis de l'utilisation des classes de types se fait aux dépens de la taille des termes. En effet, le caractère implicite des arguments de classes de types ne doit pas faire oublier que ces arguments sont bien présents, et que les instances correspondantes doivent être passées et réduites lors des calculs. Par exemple, la simple expression `{1 ; {}}` (ou `add 1 empty`) qui dénote l'ajout de l'entier 1 dans l'ensemble vide correspond en fait à l'expression absconse suivante :

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

alors que l'expression correspondante dans les modules serait simplement :

```
NatSet.add 1 NatSet.empty
```

Les applications de foncteurs sont donc remplacées par des applications d'arguments supplémentaires dans tous les opérateurs.

---

11. certaines fonctions de comparaison, pour les entiers relatifs notamment, n'étant pas purement calculatoires dans la librairie standard, leur performance pouvait être jusqu'à cinq fois moindre que celles purement calculatoires de notre formalisation. Cela ne dénote pas d'une différence significative entre modules et classes de types, mais souligne l'intérêt d'avoir une interface qui encourage à écrire des fonctions de comparaison purement calculatoires.

Si les performances des calculs ne souffrent pas de cette complexité cachée, ce n'est malheureusement pas le cas du temps mis à typer ces objets lors de la compilation d'un fichier ou simplement lors des preuves interactives où ils sont manipulés. Nous revenons sur ce point *infra* en section 6.4.

Nous avons en outre adapté notre développement d'un solveur SAT entièrement réflexif [9] à notre bibliothèque, et n'avons vu aucun changement sur les nombreux test de performances de la tactique dont nous disposions. Comme cette procédure fait un usage très important des structures d'ensemble fini, cela confirme bien que la vitesse des calculs n'est pas affectée par rapport à la bibliothèque existante.

6.2. *Mise à jour de code existant*

La mise à jour de notre solveur SAT (et également d'autres parties d'un système plus large) représentait un bon étalon pour juger de la facilité à adapter du code existant, fondé sur `FSets/FMaps`, à notre bibliothèque alternative. La base de code est en effet d'environ 30 000 lignes de Coq et utilise différents types d'ensembles finis, y compris des ensembles d'ensembles.

La conclusion de cette expérience était très positive puisque la mise à jour du code existant s'est passée sans aucun problème. De manière générale, les noms de lemmes et d'opérateurs ayant bien été conservés entre la bibliothèque existante et la nôtre, les modifications à apporter au code sont quasiment automatiques :

– pour toutes les définitions de modules vérifiant la signature `OrderedType`, définir l'instance `OrderedType` correspondante[12] ou utiliser les foncteurs décrits dans la section suivante ;

– remplacer les occurrences des types d'ensembles comme `NatSet.t` par `set nat` ;

– remplacer `destruct compare` par `destruct compare_dec` dans les preuves ;

12. et ce, seulement si elle ne peut pas déjà être inférée automatiquement par le système, ou bien encore générée par la commande `Generate OrderedType`.

– "déqualifier" toutes les références à des objets appartenant à des modules d'ensembles finis ou de propriétés, par exemple remplacer toute référence à `NatSet.add`, au lemme `NatSet.add_3` ou à la tactique `NatSetDec.fsetdec` par `add`, `add_3` et `fsetdec`.

Ces modifications ne posent pas de problème et rendent le développement plus concis et plus lisible, et ne devraient pas être un frein à l'adaptation d'une base de code existante à notre bibliothèque.

6.3. *Partage de code*

Lors de la présentation de l'interface en section 4 et des implémentations concrètes en section 5, nous avons souligné le fait que les bibliothèques de propriétés génériques ainsi que les constructions de listes et d'arbres AVL étaient exactement les mêmes que dans la bibliothèques existante, et qu'il était seulement nécessaire de les adapter légèrement aux classes de types. Il est donc naturel de s'interroger sur la duplication de code ainsi produite entre notre travail et la bibliothèque déjà disponible : il ne serait pas souhaitable que le code restât ainsi dupliqué. Cela forcerait une maintenance pénible des modifications faites dans une des deux versions.

Afin de partager un maximum de code, on peut se contenter d'écrire uniquement les versions basées sur les classes de types, puis d'en déduire à moindres frais les versions modulaires. Nous montrons cette construction sur l'exemple des types ordonnés. Étant donné les définitions de la signature `OrderedType` et de la classe de type `OrderedType` données en section 3, on peut construire le foncteur suivant qui transforme toute instance de classe `OrderedType` en un module de signature `OrderedType` :

```
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 eq_refl : ∀(x : t), eq x x :=
    reflexivity.
  Definition eq_sym : ∀(x y : t), eq x y → eq y x :=
    symmetry.
  ...
  Definition compare : ∀x y, Compare lt eq x y.
  Proof. ... Qed.
End OT_to_FOT.
```

La signature `S` permet juste d'empaqueter un type ordonné avec son instance dans un module. Le foncteur proprement dit est ensuite paramétré par un module de cette signature `S`, i.e. par le type ordonné, et crée un module de signature `OrderedType` pour le type et les relations données. L'instance et les définitions pour un certain type ordonné `t` peuvent donc être définies une seule fois, et l'utilisateur de la bibliothèque modulaire peut obtenir un module adéquat via ce foncteur. Il est intéressant de noter que l'on peut également réaliser la construction réciproque, c'est-à-dire un foncteur paramétré par un module `OrderedType` et qui retourne un module contenant simplement l'instance correspondante. Cette construction a évidemment beaucoup moins d'intérêt puisqu'elle requiert de l'utilisateur qu'il définisse chaque instance explicitement via l'application de ce foncteur. Le foncteur `OT_to_FOT`, au contraire, n'est pas plus contraignant à utiliser que le système à base de modules existant.

Le partage réalisé de cette manière peut être généralisé aux autres parties du système, ainsi on pourrait définir un foncteur qui retourne un module d'ensemble fini pour un type `A` à partir d'une instance de `FSet A`. On n'aurait ainsi qu'à dupliquer les interfaces des différentes parties du système, tout en partageant les implantations concrètes. Notre bibliothèque contient un module `Bridge` qui contient de tels foncteurs pour les types ordonnés.

6.4. *Paramétrisation de l'interface*

Dans la section 4.1, nous avons choisi de paramétrer la classe `FSet` par un type d'élément ordonné. Nous aurions également pu écrire la classe `FSet` sans ce paramètre, de la manière suivante :

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

Cette classe s'interprète légèrement différemment de celle définie dans la section 4.1 : elle n'est plus paramétrée par un type ordonné, mais chacun de ses membres l'est. Une instance de cette classe fournit donc des structures d'ensembles finis pour n'importe quel type ordonné et non pas pour un seul. Par exemple, les listes triées et les arbres AVL de la section 5.1 sont des instances potentielles de cette classe, tandis que des structures spécifiques comme des arbres de Patricia [11] ne le sont pas. L'avantage d'une telle formalisation est de pouvoir utiliser différentes instances au sein même de la classe, par exemple on peut y ajouter une opération `map` :

```
  map : ∀'{OrderedType A, OrderedType B},
          (A → B) → set A → set B
```

ce qui n'est pas possible dans l'autre cas, où même avec des modules. Il nous semble que ce type d'avantage est moins important que la capacité de pouvoir définir des structures spécifiques à certains types, en particulier les entiers, et nous avons donc préféré la formalisation où `FSet` est paramétrée.

Ce choix a malheureusement une conséquence sur la taille des termes créés à l'utilisation de la librairie. Nous avons illustré dans la section 6.1 comment les arguments implicites de classes de types venaient grossir les termes de manière cachée. Cet effet est amplifié par la paramétrisation de la classe `FSet` : en effet, tous les opérateurs de la classes `FSet` se retrouvent également paramétrés de la même manière et le type or-

donné se retrouve passé deux fois en argument. Par exemple, si `F` est une instance générique de `FSet` et `nat_OT` est de type `OrderedType nat`, l'expression `add 5 {}` est en réalité

```
@add F nat nat_OT 5 (@empty F nat nat_OT)
```

dans le cas non paramétré tandis qu'elle devient

```
@add nat nat_OT (F nat nat_OT) 5
  (@empty nat nat_OT (F nat nat_OT))
```

dans le cas paramétré. La différence peut sembler minime mais non avons mesuré précisément son effet sur un développement utilisant intensément les ensembles finis et la taille totale des preuves et définitions du fichier a augmenté de près de 40%, ainsi que le temps de compilation dudit fichier. Cette augmentation de la complexité des termes et du temps de typage est un des seuls désavantages de l'usage des classes de type, et il est regrettable qu'il se trouve ici amplifié par la paramétrisation (souhaitable) de la classe `FSet`[13]. En pratique, le temps gagné sur l'instantiation des foncteurs dépassait tout de même le temps perdu à cause de la taille des termes.

6.5. *Classes de types et modules*

Le travail présenté ici ne se veut pas une critique générale des modules par rapport aux classes de types, et encore moins une critique de la bibliothèque existante. Comme démontré dans [15], les modules et les classes de types ne sont pas interchangeables et chacun peut se prévaloir d'avantages sur l'autre. En particulier, les modules permettent un très bon contrôle de l'espace de nommage, ce que ne permettent pas les classes de types. Les modules sont très bien adaptés pour découper des systèmes en parties de taille réduite et aux interfaces bien définies ; les foncteurs permettent de remplacer facilement une de ces boîtes par une autre ayant la même interface et ceci peut se révéler d'une grande

13. on peut noter que les arguments qui se retrouvent dupliqués sont typés dans le même contexte, et donc qu'une forme de hash-consing ou de memoization dans le typage de Coq annulerait les effets négatifs provoqués par ces arguments.

aide pour tester plusieurs alternatives au sein d'un grand système par exemple. Nous utilisons fortement le système de modules à cet effet dans [9] et il répond parfaitement à nos attentes.

En revanche, nous pensons que les structures de données génériques, comme les ensembles et les dictionnaires finis, ne sont pas adaptés à un design modulaire car il est fréquent d'avoir besoin conjointement de nombreuses instances de ces structures, ce qui soulève les problèmes rappelés en section 2.2. Il nous semble donc intéressant de profiter de l'introduction des classes de types dans le système Coq pour proposer une présentation alternative de ces structures qui tire partie des classes de types.

6.6. *Travail futur*

La bibliothèque que nous proposons n'est pas encore complète vis-à-vis de la bibliothèque existante. En particulier, elle ne dispose d'aucun support pour les types à égalité décidable mais sans ordre : la bibliothèque `FSets/FMaps` supporte ces types particuliers et fournit des structures moins efficaces que pour les types ordonnés, les ensembles et dictionnaires *faibles*. Nous n'avons pas implémenté ces structures faibles car nous nous placions dans l'optique d'une bibliothèque pour un langage de programmation, et qu'il nous semble que dans la grande majorité des cas, les types que l'on souhaite manipuler dans ces structures peuvent être ordonnés totalement, au moins arbitrairement. Il serait néanmoins intéressant de les ajouter afin d'offrir les mêmes fonctionnalités que ce qui existe actuellement. On peut noter que indépendamment des structures de conteneurs, le support apporté par la bibliothèque pour les types ordonnés peut s'avérer utile dans de nombreuses occasions.

La bibliothèque pourrait également être enrichie de nouvelles structures :

– de nouvelles implémentations des classes existantes, par exemple des ensembles de Patricia[14] pour les entiers binaires, ou encore des

14. ils sont disponibles dans la bibliothèque comme une instance de `FSet` mais sans spécifications.

arbres de préfixe pour les séquences ;

– de nouvelles structures de données, bâties sur les mêmes principes, comme des graphes, des tableaux persistants, etc.

7. Conclusion

Nous avons présenté une bibliothèque Coq d'ensembles et de dictionnaires finis qui reproduit la plupart des fonctionnalités de la bibliothèque existante `FSets/FMaps` mais qui est implantée à l'aide du nouveau système de classes de types. Grâce à l'usage de ces classes de types, cette bibliothèque rend l'utilisation de ces structures beaucoup plus facile et conduit à un développement plus concis. Elle évite également certains problèmes d'efficacité dus au système de modules et à l'instantiation des foncteurs. Les développements existants fondés sur l'ancienne bibliothèque peuvent être facilement adaptés à cette nouvelle alternative. Nous sommes convaincus qu'une bibliothèque comme celle-ci permet d'améliorer Coq en tant que langage de programmation, puisqu'elle apporte des structures de données efficaces et génériques d'un emploi très aisé. En tant qu'un des premiers développements conséquents réalisé à l'aide des classes de types dans Coq, sa mise au point a également permis de faire progresser cette nouvelle fonctionnalité. Il nous semble très prometteur de continuer à améliorer cette bibliothèque en proposant d'autres structures de données utilisées fréquemment en programmation fonctionnelle.

Références

[1] The Objective Caml language - Standard Library. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/index_modules.html`.

[2] S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.

[3] J. Chrząszcz. Implementation of modules in the Coq system. In *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.

[4] The Coq Proof Assistant. `http://coq.inria.fr/`.

[5] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, pages 370–384, Barcelona, April 2004.

[6] G. M. Adel'son-Vel'skii, Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 1962.

[7] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[9] S. Lescuyer and S. Conchon. Improving coq propositional reasoning using a lazy cnf conversion scheme. In S. Ghilardi and R. Sebastiani, editors, *FroCos*, pages 287–303, 2009.

[10] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*. Springer-Verlag, 2003.

[11] C. Okasaki and A. Gill. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*, pages 77–86, September 1998.

[12] M. Sozeau and N. Oury. First-Class Type Classes. In C. M. Otmane Ait Mohamed and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 278–293, 2008.

[13] A. Stepanov and M. Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1995.

[14] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM.

[15] S. Wehr and Manuel. Ml modules and haskell type classes : A constructive comparison. *Journal for Functional Programming*, 2006.