FaCiLe en Coq : vérification formelle des listes d'intervalles

Amélie Ledein, Catherine Dubois

FaCiLe en Coq : vérification formelle des listes d'intervalles

Amélie Ledein[1], Catherine Dubois[1,2]

[1] ENSIIE, Évry, France
[2] Samovar, Évry, France
{amelie.ledein, catherine.dubois}@ensiie.fr

Abstract

Lors du développement d'un solveur de contraintes, la représentation des domaines des variables est un choix de conception important car ce dernier a une forte influence sur l'efficacité du solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à stocker une séquence d'intervalles aussi larges que possible. La bibliothèque OCaml de programmation par contraintes sur les domaines finis, nommée FaCiLe (Barnier, Brisset, 1997), ainsi que de nombreux solveurs, implantent les domaines avec une telle représentation. Nous présentons dans cet article la formalisation en Coq du module de FaCiLe dédié à la création et la manipulation des domaines. Dans ce travail, l'effort a porté non seulement sur la preuve mais surtout sur la spécification des fonctions de ce module. Plus généralement, nous proposons une nouvelle implantation Coq des ensembles finis d'entiers. Notre objectif est d'utiliser le module Coq correspondant dans le solveur de contraintes formellement vérifié développé par Carlier et al, afin d'obtenir de meilleures performances pour le code extrait. Ce travail participe également à l'effort de vérification formelle des bibliothèques OCaml existantes.

1 Introduction

La programmation par contraintes est une technique permettant de résoudre des problèmes fortement combinatoires. Un problème de satisfaction de contraintes est défini à partir de variables munies chacune d'un domaine définissant l'ensemble des valeurs possibles, et de contraintes qui expriment des propriétés qui doivent être satisfaites par les variables. Un tel problème est ensuite soumis à un solveur dédié qui retournera une ou plusieurs solutions, voire toutes les solutions, ou UNSAT si le problème n'admet pas de solution. Il existe de nombreux solveurs ou bibliothèques de résolution de contraintes, comme par exemple Choco [15], Minion [6], Gecode [5], MiniZinc [3], ECLiPSe Prolog [4], SICStus Prolog [7], FaCiLe [8].

Carlier, Dubois et Gotlieb ont développé, en 2012, un solveur de contraintes binaires à domaines finis [9], appelé CoqBinFD dans la suite. Ce solveur a la particularité d'avoir été développé avec l'assistant à la preuve Coq et d'avoir été démontré correct et complet. Ainsi quand il répond UNSAT, il a été démontré qu'il n'y a effectivement pas de solution au problème soumis. Cependant, les performances actuelles de ce solveur sont loin d'être comparables à celles des autres solveurs. Outre l'introduction de techniques de résolution dédiées (comme par exemple pour la contrainte n-aire *alldifferent*), un axe d'optimisation concerne la représentation des domaines qui a une forte influence sur l'efficacité d'un solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à utiliser une séquence d'intervalles aussi larges que possible. La bibliothèque OCaml de programmation par contraintes sur les domaines finis FaCiLe [8], développée par Barnier et Brisset dans les années 90, ainsi que de nombreux solveurs, implantent les domaines avec une telle représentation.

Nous proposons de formaliser et vérifier formellement une représentation des domaines à base de listes d'intervalles et de l'intégrer au solveur CoqBinFD en nous appuyant sur l'implantation fournie dans la bibliothèque OCaml FaCiLe (http://facile.recherche.enac.fr/).

Dans cet article, une première contribution est la vérification formelle avec Coq d'une partie du module implantant les domaines de FaCiLe, complétée par quelques fonctions utiles au solveur CoqBinFD. Une deuxième contribution est la proposition d'une nouvelle implantation formellement vérifiée des ensembles finis d'entiers qui étend le développement formel précédent. Le code Coq est disponible à l'adresse suivante https://gitlab.com/finite_set_Coq/real_intervals_list.

Le plan de cet article est le suivant. Dans un premier temps, à la section 2, nous introduisons plus précisément le contexte, à savoir la définition mathématique d'un problème de satisfaction de contraintes, les grandes lignes de la résolution, ainsi qu'une rapide présentation concernant la représentation des domaines dans les solveurs actuels. La section 3 introduit la bibliothèque FaCiLe, et plus particulièrement son module dédié à la représentation de domaines entiers, et présente également sa traduction partielle en Coq. La section 4 traite plus en détails des méthodes suivies pour spécifier et prouver en Coq la correction des fonctions implantées. Dans la section 5, nous présentons différentes extractions vers OCaml et comparons les temps d'exécution. La dernière section conclut et présente quelques perspectives.

2 Contraintes sur des domaines finis

Cette section présente brièvement la notion de problème de satisfaction de contraintes à domaines finis, ainsi que les mécanismes de la résolution de ces problèmes. Les choix faits lors du développement de CoqBinFD y sont rappelés.

Un *problème de satisfaction de contraintes sur des domaines finis* (ou CSP en abrégé, pour *Constraint Satisfaction Problem*) modélise un problème à l'aide d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine donné. Formellement, un CSP est défini par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est l' ensemble des n variables du problème,
- $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ est l'ensemble des domaines finis des n variables,
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ est un ensemble de m contraintes. Chaque contrainte C_k met en relation certaines variables de $\mathcal{X}$, ce qui restreint les valeurs que peuvent prendre simultanément chacune de ces variables. Ces contraintes peuvent être numériques (sur des réels, entiers, linéaires ou non, etc.), booléennes, sur des ensembles, etc.

Dans la suite, nous considérons des problèmes où les contraintes ne changent pas au cours de la résolution.

Selon l'objectif recherché, *résoudre un problème de satisfaction de contraintes* signifie : prouver l'existence d'une solution ou non, trouver une solution, trouver l'ensemble des solutions du problème, trouver une solution optimale par rapport à un critère (généralement minimisation ou maximisation d'une fonction de coût), prouver l'appartenance d'une valeur d'une inconnue à une solution, prouver l'appartenance d'une valeur d'une inconnue à toutes les solutions, etc. Le plus souvent, l'objectif est de trouver au moins une solution ou de prouver que le problème est UNSAT, i.e. qu'il n'a pas de solution.

Pour résoudre un CSP, la plupart des solveurs existants entrelacent 3 processus : filtrage, propagation et énumération. Le *filtrage d'une contrainte* consiste à supprimer les valeurs inconsistantes des domaines des variables de la contrainte. La consistance est ici relative à une notion

de consistance locale. Par exemple, l'*hyper-consistance* requiert que, pour chaque variable de la contrainte, pour chacune de ses valeurs possibles, il existe des valeurs (appelées *supports*) pour les autres variables telles que la contrainte soit satisfaite. Si, pour une valeur d'un domaine, cette condition n'est pas satisfaite, alors cette valeur est retirée du domaine. La *consistance de bornes* affaiblit la condition en n'imposant l'existence d'un support que pour les bornes inférieure et supérieure des domaines. Le solveur CoqBinFD s'appuie sur la *consistance d'arc* (hyper-consistance réduite à deux variables), son filtrage est prouvé complet, c'est-à-dire qu'il retire toutes les valeurs inconsistantes des domaines.

Lors du développement d'un solveur de contraintes, la représentation des domaines des variables est un choix de conception important car ce dernier a une forte influence sur l'efficacité du solveur. Une représentation assez courante - et judicieuse lorsque les domaines sont grands et *avec peu de trous* - consiste à utiliser une séquence d'intervalles aussi larges que possible. Elle est implantée par exemple dans les solveurs ECLiPSe Prolog, SICStus Prolog, Choco et FaCiLe. On trouve également d'autres représentations [16], comme les *bitvectors* (par exemple par ILOG Solver [1]), les *gap interval trees* qui stockent les valeurs qui ne sont pas dans le domaine [14] (par exemple Naxos Solver [2]) ou les ensembles creux [13] (Oscar et Castor par exemple). Le solveur CoqBinFD, quant à lui, utilise des listes triées sans doublons pour représenter les domaines.

3 FaCiLe et ses domaines

3.1 Description de la bibliothèque FaCiLe

Afin d'améliorer le type de structure utilisé dans le solveur CoqBinFD, nous avons traduit une partie d'une bibliothèque fonctionnelle de programmation par contraintes sur les domaines finis, nommée FaCiLe (*Functional Constraint Library*) [8]. Cette bibliothèque documentée[1] a été écrite en OCaml par Barnier et Brisset, notamment pour diversifier les langages de programmation avec lesquels utiliser un solveur. En effet, de très nombreux solveurs s'utilisent avec C++, Java ou Prolog. La bibliothèque FaCiLe est simple, gratuite et libre (licence LGPL), petite (moins de 5 000 lignes organisées en une douzaine de modules), d'une bonne efficacité et extensible car composée de briques de base. Elle a été inspirée notamment par ECLiPSe Prolog et ILOG Solver. Sa dernière mise à jour officielle date de décembre 2016.

FaCiLe permet de poser des contraintes sur des variables dont le domaine est un ensemble fini d'entiers ou encore un ensemble d'ensembles finis d'entiers. Nous nous intéressons uniquement au premier cas. Le module `Domain` est dédié à la représentation des domaines entiers : il offre le type abstrait `Domain.t` ainsi que de nombreuses fonctions de construction et de manipulation des domaines entiers. Un domaine est donc représenté par une liste d'intervalles eux-mêmes représentés par la donnée de leurs bornes inférieure et supérieure. L'hypothèse implicite est que la liste est triée dans l'ordre croissant des bornes et que les intervalles sont aussi grands que possible, par exemple l'ensemble {1, 2, 3, 4, 5, 7, 10, 11} sera représenté par la liste `[(1,5);(7, 7);(10, 11)]`. Ainsi la différence entre la borne inférieure d'un intervalle et la borne supérieure de l'intervalle précédent, s'il existe, est supérieure ou égale à 2. Plus précisément un domaine est un enregistrement contenant une liste d'intervalles (champ `domain`), la longueur de cette liste (champ `size`), ainsi que la valeur minimale (champ `min`) et la valeur maximale (champ `max`) de la liste d'intervalles. Lorsque le domaine est vide, les champs `min` et `max` reçoivent la valeur `min_int`.

[1] 1 600 lignes de signatures (interfaces) documentées (http://opti.recherche.enac.fr/facile/index.html.fr)

Les domaines sont construits à l'aide des fonctions suivantes (liste non exhaustive) : `Domain.empty` crée le domaine vide ; `Domain.create` crée un domaine à partir d'une liste d'entiers qui peut être non triée et contenir des doublons ; `Domain.interval` construit un domaine correspondant à un intervalle d'entiers ; `Domain.remove` retire un élément d'un domaine ; `Domain.add` ajoute un élément à un domaine. Le module offre également un grand nombre d'opérateurs ensemblistes comme l'intersection, l'union et la différence.

Les fonctions de manipulation des domaines permettent de tester l'appartenance d'un entier à un domaine (`Domain.member`), d'obtenir la longueur les valeurs minimale (`Domain.min`) et maximale (`Domain.max`) d'un domaine ou encore d'imprimer les domaines. Enfin une fonction `Domain.values` donne la liste exhaustive, et dans l'ordre croissant, des entiers contenus dans un domaine. Nous ne détaillons pas le code OCaml car il sera décrit au travers de sa traduction en Coq dans les sections suivantes.

3.2 Traduction manuelle vers Coq

L'étape de traduction du code OCaml en Coq a été réalisée manuellement, de manière quasi syntaxique. En effet, le code établi dans le module `Domain` est proche d'un code fonctionnel sans effet de bord, ce qui rend sa traduction en Coq assez simple. Parmi les fonctions traduites, une seule utilise une donnée mutable. Quelques fonctions utilisent des exceptions et des assertions. Ce caractère fonctionnel justifie notre choix d'utiliser ce module.
Afin d'assurer la traçabilité entre le code OCaml et le code Coq, les noms des fonctions n'ont pas été changés. Nous avons essayé de rester le plus proche possible du code OCaml. Cependant, quelques adaptations ont été nécessaires, notamment à cause des quelques différences existantes entre ces 2 langages : exceptions, assertions, filtrage exhaustif et terminaison des fonctions notamment. Pour plus de simplicité dans la preuve, nous avons désimbriqué les fonctions dont le code OCaml fait un usage fréquent. En ce qui concerne la terminaison, ce fut assez simple car les fonctions présentent pour la plupart une récursion structurelle ou une récursion générale dont la terminaison s'appuie sur la longueur des listes. La construction `Function` a été utilisée dans ce cas, ainsi que l'induction fonctionnelle associée lors de la démonstration de propriétés. Certaines fonctions ont donné lieu, néanmoins, à quelques modifications lors de leur traduction en Coq, pour raison d'optimisation par exemple.

Une alternative à la traduction manuelle est l'utilisation d'outils comme CoqOfOCaml [12] ou CFML [10]. Cependant, certains traits de OCaml utilisés dans cette bibliothèque ne sont pas gérés par CoqOfOCaml, ce qui empêche la traduction. De plus, cet outil introduit dans le code Coq des monades, ce qui obscurcirait inutilement le code Coq. Le code Coq obtenu avec la traduction de CFML est lui aussi éloigné du code OCaml initial, et ne facilite pas la preuve.

Finalement, 206 lignes de la bibliothèque FaCiLe définissant 29 fonctions et 2 types ont été traduites. Le code Coq correspondant compte 57 fonctions. La différence est due aux nombreuses fonctions imbriquées présentes dans le code OCaml.

4 Spécification et preuve formelles

En plus des fonctions évoquées plus haut, nous avons complété le développement avec quelques fonctions requises par le solveur CoqBinFD comme celles qui permettent d'explorer un domaine pour établir la consistance d'arc d'une contrainte. D'autre part, notre objectif étant également de fournir une implantation des ensembles finis d'entiers à l'aide de listes d'intervalles, nous avons enfin complété notre développement formel avec les fonctions manquantes spécifiées dans l'interface `FSet` de la bibliothèque standard de Coq. Le tableau de la figure 1 répertorie les

différentes fonctions de notre développement selon leur origine ou le besoin. La dernière colonne indique dans quel fichier la fonction a été définie. Ces fichiers sont accessibles directement en cliquant sur leur nom.

Fonction	FaCiLe	CoqBinFD	FSet	Définition dans
create	X	X		*create.v*
values	X		X	*structure.v*
size		X	X	*structure.v*
is_empty	X	X	X	*structure.v*
is_singleton		X		*operation_solver_basic.v*
choose	X	X	X	*operation_solver_basic.v*
fold_left		X	X	*operation_solver_fold.v*
exist		X	X	*operation_solver_exist_forall.v*
for_all		X	X	*operation_solver_exist_forall.v*
included	X	X	X	*operation_solver_included.v*
member	X	X	X	*operation_solver_member.v*
remove	X	X	X	*operation_solver_remove.v*
equal			X	*equal.v*
partition	X		X	*operation_solver_filter_partition.v*
filter		X	X	*operation_solver_filter_partition.v*
add	X		X	*operation_FSet_union.v*
union	X		X	*operation_FSet_union.v*
intersection	X		X	*operation_FSet_intersection.v*
difference	X		X	*operation_FSet_difference.v*

Figure 1: Liste des fonctions définies et prouvées correctes en Coq

4.1 Structure des domaines

La représentation utilisée d'un domaine est constituée essentiellement d'une liste d'intervalles, de type `elt_list`, stockée dans un enregistrement (champ `domain`). Pour faciliter certaines opérations liées à la résolution des contraintes, la valeur minimale et la valeur maximale sont stockées dans les champs respectifs `min` et `max`, ainsi que la taille, dans le champ `size`. Le terme *taille* ici, correspond au nombre de valeurs présentes dans le domaine. Par exemple, la taille de la liste d'intervalles `[(1, 10)]` est 10. Les types `elt_list` et `t` sont définis dans le listing 1.

```
1 Inductive elt_list :=
2 | Nil : elt_list
3 | Cons : Z -> Z -> elt_list -> elt_list.
4
5 Record t := mk_t {  domain : elt_list ; size : Z ; max : Z ; min : Z }.
```

Listing 1: Types d'une liste d'intervalles et d'un domaine

Cependant, le champ `domain` doit contenir une liste d'intervalles avec la particularité que les intervalles doivent être rangés dans l'ordre croissant, tous disjoints et non vides. De plus, cette liste doit avoir un nombre d'intervalles minimum. Cette condition de bonne formation, appelée `invariant` dans la suite, est formalisée à l'aide du prédicat `Inv_elt_list` dont la définition est donnée dans le listing 2. Le prédicat (`Inv_elt_list j l`) spécifie plus généralement les

conditions précédentes sur la liste d'intervalles `l` et, que les entiers contenus sont tous supérieurs ou égaux à la borne `j`.

```
Inductive Inv_elt_list : Z -> elt_list -> Prop :=
| invNil : forall j, Inv_elt_list j Nil
| invCons : forall (a b j : Z) (q : elt_list),
      j <= a -> a <= b ->  Inv_elt_list (b+2) q ->
      Inv_elt_list j (Cons a b q).
```

Listing 2: Invariant pour une liste d'intervalles

L'invariant ou la bonne formation du domaine, `Inv_t` dont le code Coq est donné ci-dessous (listing 3), s'en déduit alors simplement : le champ `domain` doit être bien formé au sens de l'invariant précédemment défini, le champ `min` (resp. `max`) doit contenir le plus petit (resp. grand) élément de la liste d'intervalles définie au champ `domain`, enfin la valeur du champ `size` doit correspondre à la taille de la liste d'intervalles définie au champ `domain`. Les fonctions `get_min`, `process_max` et `process_size` permettent d'obtenir respectivement le minimum, le maximum et la taille d'une liste d'intervalles donnée. La valeur `min_int` dans la définition de `Inv_t` est une valeur par déefaut, paramètre non contraint de la formalisation. FaCiLe utilise la borne inférieure des entiers et c'est pour cette raison que notre constante s'appelle ainsi.

```
Definition Inv_t (d : t) := Inv_elt_list (min d) (domain d)
                              /\ (min d) = get_min (domain d) min_int
                              /\ (max d) = process_max (domain d)
                              /\ (size d) = process_size (domain d).
```

Listing 3: Invariant pour la structure de domaine

Par la suite, nous vérifierons que les fonctions définies établissent ou préservent ces invariants.

4.2 Vérification formelle des fonctions

De manière générale, pour la plupart des fonctions sur les domaines, est définie une fonction analogue sur une liste d'intervalles, comme dans FaCiLe. Par exemple, la fonction `remove` qui permet de retirer un élément d'un domaine (opération effectuée lors du filtrage par exemple), de type `Z -> t -> t`, est définie en utilisant la fonction `elt_list_remove` qui supprime un élément éventuellement présent dans une liste d'intervalles et retourne un booléen indiquant si la suppression a été effective ou non. Enlever un élément d'une liste d'intervalles consiste à rechercher l'intervalle $\mathcal{I}$ susceptible de contenir l'élément à supprimer, et selon les cas, à supprimer l'intervalle (cas où $\mathcal{I}$ est un singleton), modifier une des bornes (cas où l'élément était une des bornes de $\mathcal{I}$), ou scinder $\mathcal{I}$ en deux intervalles. La fonction `remove` s'occupe aussi de calculer les valeurs minimale et maximale, ainsi que la longueur. De même, les fonctions `elt_list_member` et `member` testent l'appartenance d'un élément à, respectivement, une liste d'intervalles (voir listing 4) ou un domaine. La première explore la liste jusqu'à trouver un intervalle contenant l'élément recherché ou une borne strictement supérieure à ce dernier.

```
Fixpoint elt_list_member (x : Z) (l : elt_list) : bool := match l with
|Nil => false
|Cons min max q => if x <=? max then  x >=? min
                   else elt_list_member x q
end.
```

Listing 4: Appartenance d'un élément à une liste d'intervalles

Du point de vue vérification, chaque fonction est accompagnée d'une preuve de préservation ou établissement de l'invariant associé (`Inv_elt_list` ou `Inv_t` selon que la fonction travaille sur une liste d'intervalles ou un domaine). Elle est également accompagnée d'un théorème de correction dont l'énoncé est inspiré de celui de l'interface des ensembles finis de Coq (`Fsetinterface`). Les différents théorèmes pour les fonctions `elt_list_remove` et `remove` sont donnés, pour illustration, dans le listing 5.

```
Lemma elt_list_remove_inv : forall (l1 : elt_list) (e : Z) l2 b y,
elt_list_remove l1 e = (b, l2) ->
Inv_elt_list y l1 ->
Inv_elt_list y l2.

Theorem remove_inv : forall (d : t),
Inv_t d -> forall (e : Z), Inv_t (remove e d).

Theorem elt_list_remove_spec_1 : forall l x y,
Inv_elt_list y l ->
elt_list_member x (snd (elt_list_remove l x)) = false.

Theorem remove_spec_1 : forall d v,
Inv_t d -> member v (remove v d) = false.

Theorem elt_list_remove_spec_2_3 : forall l v w y,
Inv_elt_list y l -> w <> v ->
elt_list_member w (snd (elt_list_remove l v)) = elt_list_member w l.

Theorem remove_spec_2_3 : forall d v y,
Inv_t d -> y <> v ->
member y (remove v d) = member y d.
```

Listing 5: Théorèmes relatifs aux fonctions `elt_list_remove` et `remove`

4.3 Zoom sur la fonction `create`

La fonction `create` (voir son code dans le listing 6) prend en paramètre une liste d'entiers l et crée le domaine correspondant. La liste l est quelconque, elle peut donc contenir des doublons et ne pas être triée dans l'ordre croissant. La fonction `create` agit de la façon suivante :

- elle trie la liste l dans l'ordre croissant en éliminant les doublons, en appliquant la fonction `sortWithoutDup` qui est ici une adaptation du tri par insertion (alors que le code OCaml de FaCiLe utilise la fonction `List.sort` de la bibliothèque standard - plus efficace que notre fonction de tri - et ensuite enlève les doublons),
- elle crée ensuite le domaine par un appel à la fonction `unsafe_create`.

```
Definition create (Zl : list Z) : t := unsafe_create (sortWithoutDup Zl).
```

Listing 6: Définition de la fonction `create`

L'objectif, à présent, est de montrer que cette fonction établit bien l'invariant précédemment défini, c'est-à-dire que le domaine créé respecte l'invariant.

Un premier travail consiste alors à montrer que la fonction `sortWithoutDup` est correcte, c'est-à-dire qu'elle produit une liste triée dans l'ordre croissant sans doublons, et que la liste produite contient les mêmes éléments que la liste initiale. Pour ce faire, nous avons utilisé le prédicat `NoDup` de la bibliothèque standard de Coq, adapté le prédicat `Sorted` à nos besoins, et introduit un prédicat `SameElt` formulant que 2 listes ont les mêmes éléments :

```
Definition SameElt l1 l2 : Prop := forall (x : Z), In x l1 <-> In x l2.
```

Listing 7: Prédicat formulant que 2 listes ont les mêmes éléments ou non

Nous pouvons alors écrire la spécificationde la fonction `sortWithoutDup` :

```
Theorem sortWithoutDup_spec :
forall l, Sorted (sortWithoutDup l)
       /\ SameElt l (sortWithoutDup l)
       /\ NoDup (sortWithoutDup l).
```

Listing 8: Spécification de la fonction `sortWithoutDup`

Ensuite, nous avons montré que la fonction `unsafe_create` est correcte, c'est-à-dire qu'à partir d'une liste triée dans l'ordre croissant et sans doublons, elle construit un domaine respectant l'invariant précédemment établi. Cette fonction utilise la fonction `make` pour créer la liste d'intervalles correspondant à une liste d'entiers supposée triée dans l'ordre croissant et sans doublons. En effet, la fonction `make a b l` parcourt la liste l à la recherche d'une suite $b+1, b+2 \ldots b+n$ jusqu'à une rupture, et construit alors l'intervalle de a à $b+n$ et recommence avec les éléments suivants de l.
Pour démontrer que `create` établit l'invariant concernant la liste d'intervalles, nous avons prouvé le théorème suivant qui porte sur la fonction `make` :

```
Theorem elt_list_make_inv : forall l a b j,
Sorted l -> NoDup l -> j <= a -> a <= b ->
(forall x, In x l -> b+1 <= x) -> Inv_elt_list j (make a b l).
```

Listing 9: Théorème essentiel sur la fonction `make`

Nous avons donc ainsi pu démontrer que la fonction `create` produit un domaine qui respecte l'invariant :

```
Theorem create_inv : forall l, Inv_t (create l).
```

Listing 10: Établissement de l'invariant par la fonction `create`

Il reste à démontrer la correction de la fonction `create` (listing 11), à savoir l'équivalence entre l'appartenance au domaine `create l` et l'appartenance à la liste initiale l (`In` est ici le prédicat d'appartenance à une liste de la bibliothèque standard de Coq).

```
Theorem create_spec : forall l,
(forall y, member y  (create l) = true <-> In y l).
```

Listing 11: Spécification de la fonction `create`

Ce théorème découle de propriétés sur la fonction `make`, telle que celle présentée dans le listing 12, qui énonce que tout élément de la liste d'intervalles `make a b l` est, soit un entier

compris entre `a` et `b`, soit un élément de `l`. On démontre également la propriété réciproque avec des hypothèses supplémentaires spécifiant que la liste `l` ne doit contenir que des entiers strictement supérieurs à `b` et que cette liste est triée et sans doublons.

```
1 Lemma elt_list_member_make : forall l a b, a <= b ->
2  forall y, elt_list_member y (make a b l) = true ->
3  a <= y <= b \/ In y l.
```

Listing 12: Propriété essentielle sur la fonction `make` pour prouver la spécification de `create`

Dans cette partie du développement formel, la difficulté a été de retrouver les hypothèses nécessaires pour réussir à démontrer les propriétés.

4.4 Opérations ensemblistes classiques

Les opérations ensemblistes telles que l'intersection, l'union et la différence, présentes dans FaCiLe ont été traduites et prouvées correctes par rapport aux spécifications *classiques* de la théorie des ensembles énoncées dans l'interface `FSetinterface`. Toutes ces fonctions, lorsqu'elles sont définies au niveau des listes d'intervalles, utilisent une récursion générale s'appuyant sur la décroissance de la taille d'une des deux listes en entrée. Les preuves de terminaison sont très simples. Les preuves de préservation d'invariant et de correction sont fastidieuses et longues, requérant de s'intéresser à de nombreux cas. Il faut noter que, lors de la traduction, la fonction qui calcule l'intersection des domaines `s1` et `s2` a été simplifiée car elle commence par quelques optimisations (si les deux ensembles sont égaux, alors l'intersection est l'un des ensembles; si la taille de la liste d'intervalles de l'intersection est égale à la taille de `s1` (resp. `s2`), alors l'intersection est égale à `s1` (resp. `s2`)) qui requièrent des démonstrations que nous n'avons pas eu le temps de faire. Ces dernières entrent dans les perspectives de ce travail. Une simplification similaire (comparaison préalable des tailles des domaines et des valeurs minimales et maximales) a été faite, pour le code de la fonction `included` qui teste si un domaine est inclus dans un autre, par manque de temps également. Une perspective est bien sûr de considérer ces optimisations, néanmoins le code Coq devra distinguer les cas des ensembles vides, ce que ne fait pas le code FaCiLe, car il s'appuie sur le fait que `min_int` est inférieur ou égal à tout `int`. La preuve des propriétés de correction de la fonction `elt_list_included` a permis de proposer une version plus efficace de cette fonction, version utilisée dans les expérimentations présentées dans la section 5.

4.5 For_all, Exist, Partition, Filter, Fold

Ces fonctions ne sont pas présentes dans le module `Domain` de FaCiLe. Elles sont néanmoins nécessaires dans le solveur CoqBinFD, par exemple pour réaliser un filtrage, et sont requises par l'interface `FSetinterface` de Coq. Elles suivent un schéma légèrement différent de celui des fonctions précédentes, car il est besoin d'explorer chaque valeur des intervalles. On aurait pu, en utilisant la fonction `values` obtenir la liste des éléments et ensuite appliquer les fonctions correspondantes sur les listes. Néanmoins ce processus algorithmique se révèle trop coûteux.

La fonction `filter` doit construire un domaine à partir d'un domaine argument et d'une fonction booléenne. La fonction `elt_list_filter`, qui travaille sur les listes d'intervalles, construit, pour chaque intervalle présent dans la liste de départ, un ou plusieurs intervalles qu'elle assemble ensuite. Par exemple, si la liste d'intervalles est `[(1, 6);(40, 43)]` la fonction qui filtre tous les entiers pairs construira la liste `[(2, 2);(4, 4);(6, 6);(40, 40);(42, 42)]`. La fonction est inspirée de la fonction `make` précédemment décrite. La fonction `partition p`

`d` fait deux appels à `filter`, l'un pour construire le domaine qui contient les éléments de `d` qui satisfont le prédicat `p`, l'autre pour construire le domaine des éléments de `d` qui ne satisfont pas `p`. Cette implantation n'est certes pas efficace, l'optimisation qui consiste à construire les deux domaines simultanément reste à faire mais devrait suivre un schéma similaire.

4.6 Conformité à l'interface `FSet` de Coq

Une nouvelle implantation des ensembles finis d'entiers, qui s'appuie sur les domaines utilisant les listes d'intervalles, est disponible (dans le fichier `bridge.v`). Le module définit le type `t_FSet` (défini dans le listing 13, où le type `structure.t` désigne le type des domaines défini dans le listing 1) comme un enregistrement contenant un domaine et une preuve de sa bonne formation. Chaque fonction de ce module encapsule le résultat de la fonction correspondante sur les domaines, ainsi que la preuve de préservation ou d'établissement de l'invariant le cas échéant.

```
1 Record t_Fset := mk_t_Fset {
2   set : structure.t ;
3   wf : Inv_t set
4 }.
```

Listing 13: Type d'un domaine dans notre module

Le code de `remove` est donné en illustration dans le listing 14.

```
1 Definition remove x s :=
2   let res := operation_solver.remove x (set s) in
3   mk_t_Fset res (remove_inv (set s) (wf s) x).
```

Listing 14: Définition de la fonction `remove` dans notre module

La relation d'ordre permettant d'ordonner totalement les ensembles est définie à partir d'une relation d'ordre totale portant sur les listes ordonnées d'entiers via la fonction `values` (`elements` selon l'interface `FSetinterface`).

5 Extractions et expérimentations

Le mécanisme d'extraction de Coq permet d'obtenir un code OCaml exécutable, mécanisme que nous avons utilisé en vue de pouvoir utiliser les listes d'intervalles comme représentation des domaines avec le solveur de contraintes CoqBinFD.

Nous présentons dans cette section quelques tests que nous avons réalisés avec le code extrait afin de comparer les temps d'exécution obtenus, en fonction notamment de la forme du domaine. Nous comparons également ces temps d'exécution avec les temps obtenus en réalisant les mêmes tests avec la bibliothèque FaCiLe.

Le mécanisme d'extraction de Coq permet de faire quelques ajustements dans le code extrait, par exemple remplacer certains types (ou fonctions) définis en Coq par des types (ou fonctions) de OCaml, et par conséquence, dans certains cas, *casser* la correction du code extrait. La formalisation en Coq présentée dans cet article utilise les entiers mathématiques (de type `Z`) alors que la bibliothèque FaCiLe utilise les `int` de OCaml, entiers bornés *modulo*. L'extraction des entiers mathématiques vers les entiers de OCaml, dans laquelle la définition du type Z (resp. certaines fonctions de calcul sur `Z`) est remplacée par le type `int` (resp. par

des fonctions OCaml manipulant des `int`), n'est en ce sens pas correcte : le code extrait peut, par exemple, occasionner des dépassements de capacité alors que le calcul en Coq ne pose pas de problème. Néanmoins nous utilisons cette extraction incorrecte[2] pour pouvoir comparer, en termes de temps d'exécution, le code issu de notre développement avec le code FaCiLe. Dans la suite, ce code extrait est appelé code CoqInt. Nous présentons également une extraction dite propre (ou extraction vers Z), où la traduction de la définition de Z est utilisée dans le code extrait - appelé dans la suite CoqZ - ainsi qu'une extraction vers les grands entiers de OCaml[3]. Le code extrait issu de cette dernière extraction est appelé CoqBG. Notons que, dans chaque expérience d'extraction, nous avons remplacé les booléens de Coq par les `bool` de OCaml et leurs opérations `andb` et `orb` par les opérateurs paresseux de OCaml correspondants.

Pour nos tests de comparaison, quatre catégories de domaines ont été définies, chacune contenant 5 domaines :

- *Petits domaines* : moins de 10 valeurs, c'est-à-dire `size` ≤ 10;
- *Grands domaines* : plus de 100 valeurs, c'est-à-dire `size` ≥ 100;
- *Domaines compacts* : peu de *trous* (`size` $= 100$).
- *Domaines creux* : des valeurs éparpillées (`size` $= 100$);

Cela permet d'avoir de nombreux cas de figure des domaines possibles, ce qui aurait été plus compliqué à maîtriser si les domaines avaient été générés aléatoirement.

Les résultats obtenus sont présentés, par catégorie, sur les graphiques des figures 3, 4, 5 et 6. Les fonctions énumérées dans le tableau de la figure 1 sont exécutées sur chacun des domaines. Chaque valeur correspond au temps d'exécution moyen de 1 000 itérations par domaine. L'unité utilisée est la seconde. Dans les graphiques 4, 5 et 6, la fonction `create` n'apparaît pas pour des raisons d'échelle, tous ses temps d'exécution sont reportées sur le graphique de la figure 2.

Les tests ont été réalisés sur un MacBook Pro *2.3 GHz Intel Core I5* avec *8 GB 2133 MHz LPDDR3*, avec OCaml 4.05 et Coq 8.9.1.

5.1 Comparaison CoqInt - FaCiLe

La comparaison ne porte ici que sur les fonctions communes aux deux codes. Lorsque nous utilisons des *petits domaines*, les temps obtenus en exécutant le code FaCiLe et le code extrait CoqInt sont sensiblement les mêmes, à l'avantage de FaCiLe sauf pour dans le cas de la fonction `is_empty` qui, pourtant, ne réalise qu'un accès à un champ d'un enregistrement dans les deux codes. Concernant les *grands domaines*, les temps obtenus sont sensiblement les mêmes, sauf pour `create`. En effet, la fonction `create` de FaCiLe est 25 fois plus rapide que celle du code Coq extrait. Nous payons ici le choix d'un tri moins efficace. Avec des *domaines compacts*, nous constatons les mêmes résultats. Enfin, pour les *domaines creux*, les temps obtenus restent également proches. Les mêmes tests ont été réalisés sur la fonction `create` en modifiant dans les codes extraits de Coq le tri vérifié formellement par un appel à `List.sort` et la fonction de FaCiLe qui enlève les doublons. Les résultats sont montrés à la figure 2, pour les catégories des domaines grands, compacts et creux. Les temps d'exécution pour les petits domaines n'apparaissent pas car ils sont très inférieurs aux autres. La barre int (resp. Z et big_int) concrétise le temps pour le code extrait CoqInt (resp. CoqZ et CoqBG) alors que la barre

[2] Nous avons utilisé le module `Coq.extraction.ExtrOcamlZInt` de la bibliothèque standard de Coq
[3] en utilisant le module `Coq.extraction.ExtrOcamlZBigInt` de la bibliothèque standard de Coq

Z/sort désigne le même code avec la fonction non certifiée de tri. Il apparaît clairement que le choix du changement de tri rend le code moins efficace lors de la création de domaines de grande taille.

Nous avons également réalisé quelques tests aux limites, par exemple la création du domaine le plus grand possible, à l'aide de la fonction `interval`, soit `interval min_int max_int`. Le calcul de la taille de ce domaine est, sans surprise, égale à 0 en exécutant le code FaCiLe et le code CoqInt. Un problème est également détecté lors du calcul de l'union du domaine de valeurs 0 et 1 (`boolean`) et du domaine réduit à l'intervalle `min_int` .. `min_int` + 2 (`interval min_int (min_int + 2)`). Les intervalles du résultat sont les bons mais mal ordonnés dans la liste, alors que l'union de `(interval min_int (min_int + 2))` et de `boolean` est calculée correctement. Des problèmes de dépassement arithmétique sont également constatés sur la différence, dès lors qu'une borne `min_int` ou `max_int` est utilisée. Ces tests s'exécutent, bien entendu, sans problème dans Coq et avec les autres extractions, puisque `min_int` et `max_int` y sont des entiers comme les autres. Cette possibilité de dépassement de capacité arithmétique est mentionnée dans la documentation de FaCiLe sous la forme de la petite note suivante : *Users should be careful when expecting the arithmetic solver to compute bounds from variables with very large domain, that means with values close to max_int or min_int (depending on the system and architecture).*

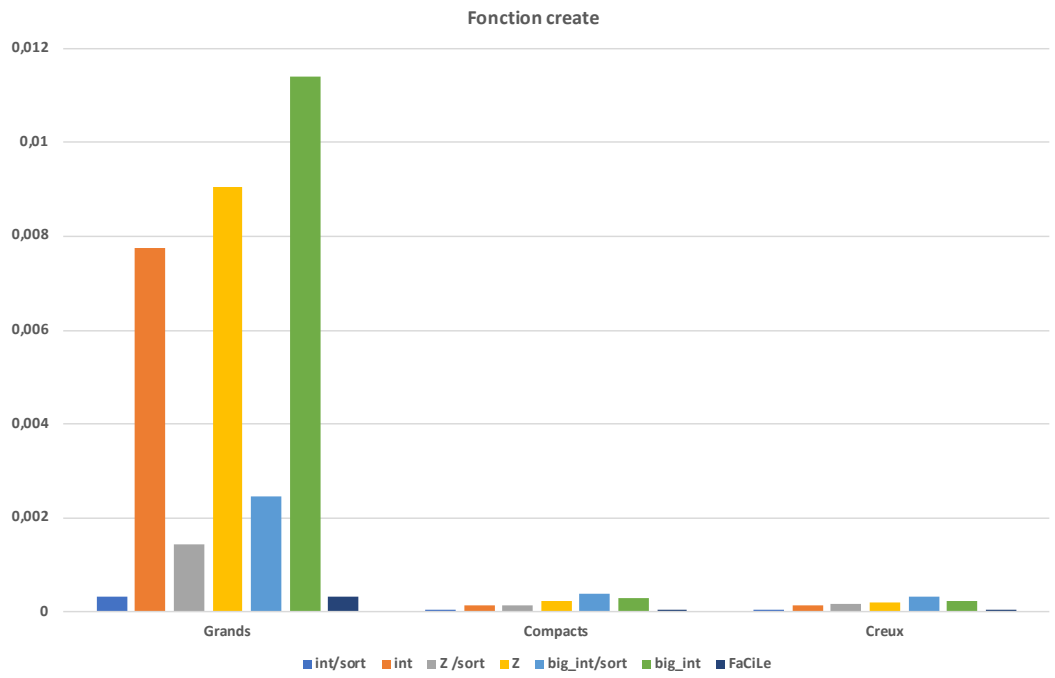

Figure 2: Temps d'exécution moyen en seconde pour 1 000 itérations de la fonction `create` (axe horizontal) sur 5 domaines des catégories Grands, Compacts et Creux

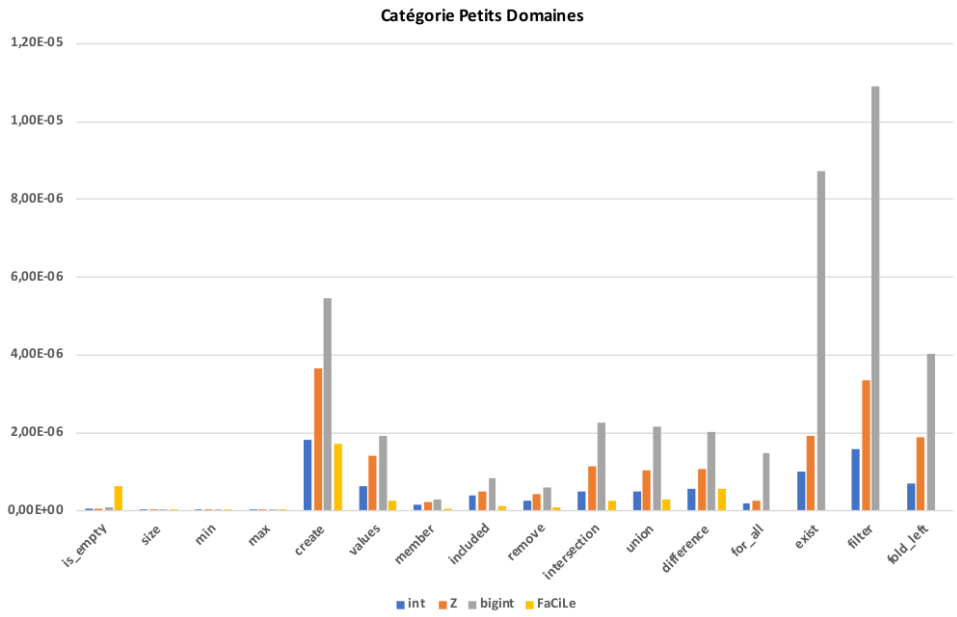

Figure 3: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Petits domaines* pour les 3 codes extraits et le code FaCiLe

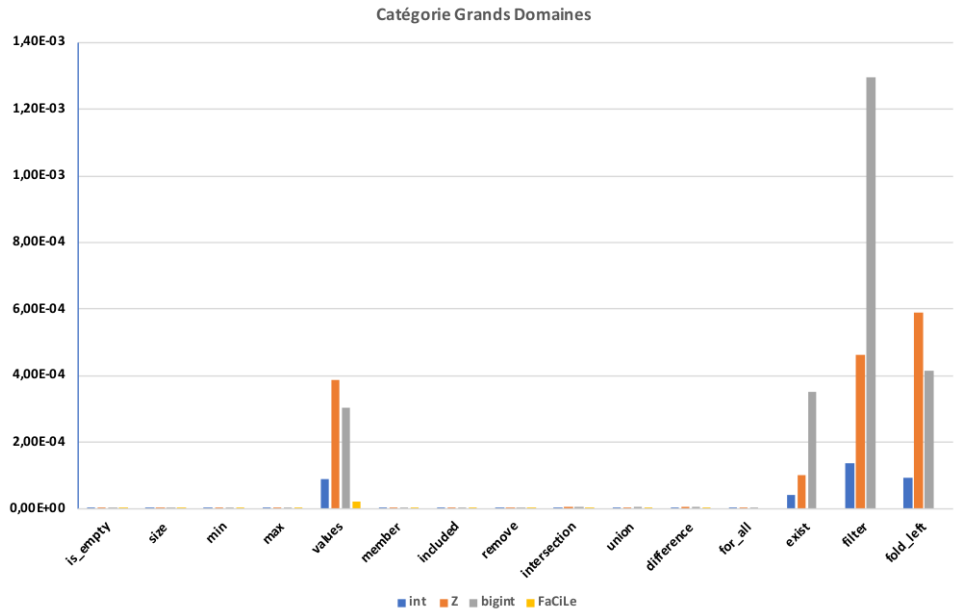

Figure 4: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Grands domaines* pour les 3 codes extraits et le code FaCiLe

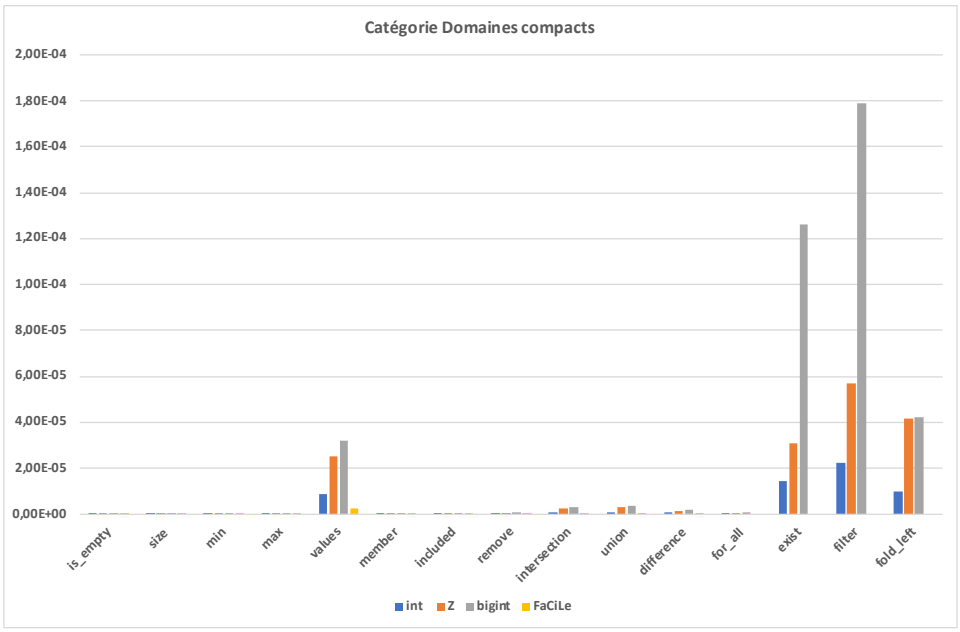

Figure 5: Temps d'exécution moyen en seconde pour 1 000 itérations des fonctions sur 5 domaines de la catégorie *Domaines compacts* pour les 3 codes extraits et le code FaCiLe

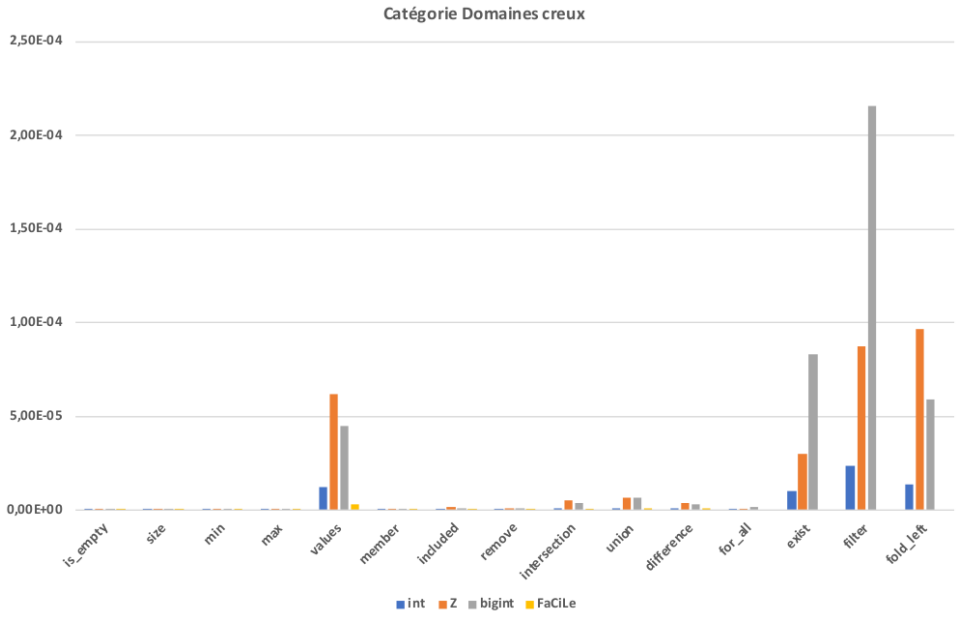

Figure 6: Temps d'exécution moyen en seconde pour 1 000 itérations (axe horizontal) sur 5 domaines de la catégorie *Domaines creux* pour les 3 codes extraits et le code FaCiLe

5.2 Comparaison CoqInt - CoqZ et CoqBG

Pour toutes les catégories, on peut constater, de manière générale, que le temps d'exécution pour le code extrait CoqBG est plus important que celui obtenu avec CoqZ, lui-même supérieur à celui obtenu avec CoqInt. Pour la plupart des fonctions, les temps de calcul obtenus avec l'extraction propre, CoqZ, restent raisonnables. Les fonctions qui parcourent toutes les valeurs du domaine, par exemple `filter`, ont un temps d'exécution trop important sur les domaines grands et compacts. Ce temps dépend aussi des calculs faits sur chaque entier du domaine.

Nous pouvons distinguer 4 groupes de fonctions et donc 4 profils d'exécution. Les fonctions `is_empty`, `size`, `min` et `max` accèdent à un champ de la structure et sont donc réalisées en temps constant. Les fonctions ensemblistes, `member`, `included`, `remove`, `union`, `intersection` et `difference`, parcourt la liste des intervalles et font quelques comparaisons relativement aux bornes des intervalles. Le temps d'exécution est donc linéaire par rapport au nombre d'intervalles. Le troisième groupe regroupe les fonctions `for_all`, `exist`, `filter`, `fold_left` et `values` qui requièrent d'énumérer, une à une, les valeurs du domaine, en totalité ou non selon les fonctions. Les temps d'exécution sont par conséquent liés à la taille de l'ensemble fini encodé. Enfin le quatrième groupe est composé de la fonction `create` déjà évoquée plus haut, qui requiert l'utilisation d'un algorithme de tri efficace.

6 Conclusion

Le développement formel que nous avons présenté dans cet article permet de fournir une implantation vérifiée formellement des domaines d'entiers contenant les fonctions nécessaires à une nouvelle implantation du solveur CoqBinFD. Ce travail fournit également une implantation formellement vérifiée des ensembles finis d'entiers utilisable dans tout développement Coq. Il est également un début de vérification formelle du module dédié à la représentation des domaines de la bibliothèque OCaml existante FaCiLe.

Au total, le développement formel Coq compte environ 7 750 lignes de code (hors le module faisant le lien avec `FSetinterface`) et comprend 2 définitions de type, 2 prédicats logiques qui définissent la bonne formation des listes d'intervalles et des domaines, 70 définitions de fonctions dont 57 sont issues de la traduction en Coq de fonctions OCaml extraites de FaCiLe. Environ 263 lemmes et théorèmes ont été démontrés. Le fichier `bridge.v` ajoute environ 950 lignes de code, la majeure partie de ses définitions et lemmes étant des *redirections* vers des définitions et lemmes existants. Le code Coq ainsi que les fichiers de test sont disponibles à l'adresse suivante https://gitlab.com/finite_set_Coq/real_intervals_list.

Nous envisageons de poursuivre ce travail en prenant en compte les quelques optimisations *oubliées* par manque de temps et en reliant cette représentation des domaines au solveur CoqBinFD. De manière générale nous en envisageons une refonte plus modulaire, permettant ainsi d'être indépendant de la représentation des domaines utilisée. La variante utilisant la consistance de bornes décrite dans [11] pourrait également bénéficier utilement du développement formel présenté ici, car un accès efficace aux bornes y est crucial. Une autre perspective importante concerne l'obtention d'un code OCaml extrait efficace. Nous projetons pour cela de reprendre le développement formel en utilisant les entiers cycliques de la bibliothèque Coq, ce qui nous permettrait d'avoir un code extrait efficace et certifié. Ce travail contribue à la vérification formelle de bibliothèques OCaml existantes, pour cette raison nous avons tenu à rester le plus proche possible de l'implantation existante. Néanmoins une perspective de ce travail est de généraliser la notion de domaine à un autre support que celui des entiers, per-

mettant ainsi de définir et manipuler des domaines sur un type fini quelconque. Enfin, un travail envisagé à plus long terme est de formaliser les autres représentations utilisées dans les solveurs existants (cf section 2), en particulier les représentations qui se concentrent sur les valeurs absentes (*gap interval list* ou *gap interval tree*) avec un objectif de réutilisation des preuves effectuées dans le cadre du développement formel présenté dans l'article.

Remerciements : nous remercions vivement les rapporteurs anonymes qui, par leurs commentaires, nous ont permis d'améliorer cet article. Ce travail a été financé partiellement par la direction de la recherche de l'ENSIIE.

Bibliographie

[1] Documentation du solveur ilog solver. http://lia.deis.unibo.it/Courses/AI/applicationsAI2009-2010/materiale/cp15doc/usrsolver/usrsolverpreface.html.

[2] Github du solveur naxos solver. https://github.com/pothitos/naxos.

[3] Site officiel de minizinc. https://www.minizinc.org/.

[4] Site officiel du solveur eclipse prolog. http://eclipseclp.org/.

[5] Site officiel du solveur gecode. http://www.gecode.org/.

[6] Site officiel du solveur minion. https://constraintmodelling.org/minion/.

[7] Site officiel du solveur sicstus prolog. https://sicstus.sics.se/.

[8] P. Brisset and N. Barnier. FaCiLe : a Functional Constraint Library. In *CICLOPS 2001, Colloquium on Implementation of Constraint and LOgic Programming Systems*, Paphos, Cyprus, 2001.

[9] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 116–131, Paris, France, 2012.

[10] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011.

[11] C. Dubois and A. Gotlieb. Solveurs cp(fd) vérifiés formellement. In *Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 115–118, 2013.

[12] C. Guillaume. Coq of OCaml. The OCaml Users and Developers Workshop, Gothenburg, Sweden, September 5, 2014.

[13] V. Le clément de saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[14] N. Pothitos and P. Stamatopoulos. Flexible management of large-scale integer domains in csps. In *Artificial Intelligence: Theories, Models and Applications, 6th Hellenic Conference on AI, SETN 2010, Athens, Greece*, volume 6040 of *LNCS*, pages 405–410. Springer, 2010.

[15] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. http://www.choco-solver.org.

[16] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 495–526. Elsevier, 2006.