Domaines formellement vérifiés

Catherine Dubois
ENSIIE, INRIA, Université Paris-Saclay, LMF, France

catherine.dubois@ensiie.fr

Résumé

Cet article s'intéresse à la correction de la représentation des domaines des variables dans un solveur de contraintes. Nous proposons d'utiliser la vérification déductive, et plus précisément l'outil Why3, afin de fournir des implémentations correctes des domaines. L'article résume les travaux réalisés par l'autrice, concernant la vérification formelle des sparse sets pour représenter les domaines finis entiers. La contribution nouvelle concerne le développement d'une implémentation formellement vérifiée des domaines ensemblistes, basée, elle aussi, sur les sparse sets.

Mots-clés

vérification formelle, domaines entiers, domaines ensemblistes, sparse sets, Why3

1 Introduction

Développer un solveur de contraintes requiert entre autres de choisir une représentation efficace et correcte pour les domaines des variables. Différentes structures de données sont utilisées dans les solveurs existants : séquences ou arbres d'intervalles, bit-vecteurs, ensembles épars (*sparse sets* en anglais et dans la suite), etc. Nous proposons de développer de telles représentations, en utilisant un assistant à la preuve puis d'en extraire un code exécutable. L'assistant à la preuve nous permet d'implémenter, spécifier et prouver que l'implémentation est correcte par rapport à la spécification. A l'aide de Coq, nous avons formalisé les séquences d'intervalles pour représenter les domaines des variables entières [6]. Ici, nous nous concentrons sur les *sparse sets* et les domaines ensemblistes représentés avec une variante des *sparse sets*, à l'aide de l'assistant à la preuve Why3. La contribution nouvelle est la formalisation en Why3 des domaines ensemblistes. La première contribution a fait l'objet de la publication [2] que nous résumons dans l'article.

Nous mettons l'accent sur les spécifications des structures de données, en particulier leurs invariants ainsi que sur les spécifications des opérations. Nous démontrons en particulier que les opérations établissent ou préservent les invariants. Ces développements, dont le code complet est disponible, ont permis d'extraire du code exécutable en OCaml.

Nous avons choisi l'outil de vérification déductive Why3 car il y est aisé de formaliser et prouver des algorithmes manipulant des données mutables, puis d'en extraire du code exécutable efficace. Il est plus difficile de représenter des structures de données mutables en Coq, les fonctionnalités impératives, comme par exemple la modification en place de tableaux, n'y étant pas directement disponibles.

Une approche alternative consiste à prouver la correction des implémentations existantes. Cette approche est impossible pour le code d'un solveur dans son ensemble car celui-ci est trop complexe. Elle pourrait cependant être possible pour les modules relatifs aux domaines (e.g. dans FaCiLe [6]). Cependant, en lisant, par exemple, le code de MiniCP ou OscaR, on se rend compte que de nombreux aspects sont imbriqués et qu'il est difficile d'isoler les seuls éléments des structures de données d'étude.

2 Présentation rapide de Why3

Why3 [3] est une plateforme de vérification déductive de programmes qui fournit un langage de spécification et de programmation, appelé WhyML. Ce dernier permet à l'utilisateur d'écrire des programmes fonctionnels ou impératifs comportant du polymorphisme, des types et prédicats inductifs, du filtrage, des exceptions, des références, des tableaux, etc. Les programmes sont spécifiés à l'aide de contrats contenant des pré- et post-conditions et complétés éventuellement par des invariants de boucle. La logique sous-jacente est une extension de la logique du premier ordre. A partir d'un programme annoté par sa spécification, Why3 génère des conditions de vérification (VC), formules logiques obtenues par un calcul de plus faible pré-condition et dont la preuve assure la correction du programme par rapport à sa spécification. Pour démontrer les VCs, Why3 s'appuie sur des prouveurs automatiques standards (nous utilisons ici Alt-Ergo, CVC4 et Z3) ainsi que des assistants de preuve interactifs tels que Coq ou Isabelle (que nous n'avons pas utilisés ici). Des tactiques de preuve sont également fournies, pour guider la preuve ou comprendre les manques ou erreurs au cours du développement.

Il existe de nombreuses bibliothèques qui peuvent être utilisées, pour l'arithmétique des entiers, les listes polymorphes, les ensembles finis, les fonctions partielles, etc.

À partir des programmes WhyML vérifiés, des programmes OCaml (et récemment des programmes C) corrects par construction peuvent être automatiquement extraits.

3 *Sparse sets* vérifiés

Un *sparse set* permet de représenter le domaine d'une variable entière, inclus dans un intervalle $[0, N-1]$ [1], où

1. Ceci est un peu restrictif mais il est facile de retrouver le cadre plus général en utilisant un *offset* comme cela est fait dans OscaR par exemple.

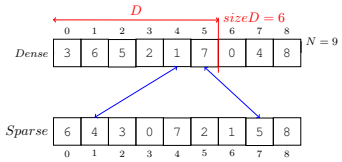


FIGURE 1 – Un *sparse set*

N est un entier naturel non nul. Cette structure de données utilise deux tableaux de longueur N appelés $Dense$ et $Sparse$, et un entier naturel $sizeD$. Les éléments du domaine, $setD$, sont les entiers présents dans le sous-tableau $Dense[0..sizeD-1]$. Le tableau $Sparse$ associe toute valeur v de $[0, N-1]$ à sa position i dans $Dense$. Les invariants de la structure de données sont les suivants :

$$setD \subseteq 0..N-1 \wedge setD = Dense[0..sizeD-1] \quad (P_1)$$
$$Sparse[v] = i \iff Dense[i] = v, \text{ for all } i, v \quad (P_2)$$

Un exemple est donné à la figure 1. Les flèches bleues y illustrent l'invariant P_2 : $Dense[1] = 4, Sparse[4] = 1$.
Cette représentation est particulièrement efficace pour le test d'appartenance et les opérations remove (qui retire un élément du domaine) et bind (qui lie une variable à une valeur), qui sont en temps constant. Tester l'appartenance d'un élément v au domaine consiste à évaluer l'expression $v < sizeD$, retirer v revient à échanger, dans $Dense$, v avec $Dense[N-1]$, et de faire les modifications correspondantes dans $Sparse$, réduire le domaine à $\{v\}$ consiste à échanger v avec $Dense[0]$ et mettre à jour $Sparse$.

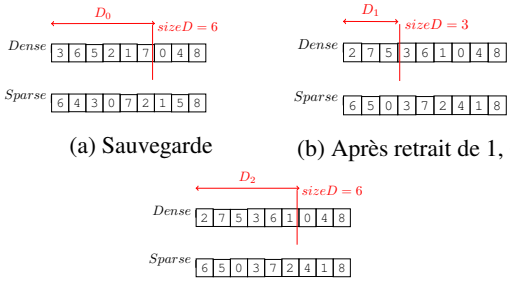
La formalisation en WhyML suit très fidèlement cette présentation. Un extrait en est donné dans le listing 1. Le type des *sparse sets*, `tsparse`, est défini ici comme un type enregistrement comprenant, outre les champs `n`, `dense` et `sparse`, un champ `setD` dit fantôme. Il aura pour valeur le domaine courant et n'est là que pour les besoins de la spécification et faciliter la preuve. Lors de l'extraction du code OCaml, il sera effacé. Cette valeur fantôme, un ensemble mathématique de type polymorphe `fset int`, est mise à jour dans les différentes fonctions qui modifient un *sparse set*. Dans la suite, nous écrivons ce code fantôme en italiques et utilisons les notations mathématiques usuelles pour dénoter les opérations ensemblistes mathématiques. Le type `tsparse` est accompagné d'un invariant qui reprend les deux propriétés P_1 et P_2 énoncées plus haut. `dom_ran dense n` spécifie que `n` est positif ou nul, que le tableau `dense` est de longueur `n` et que ses éléments sont compris entre 0 et `n-1`. L'invariant est vérifié aux limites des appels de fonction : chaque fois qu'un *sparse set* est créé ou modifié, une VC correspondant resp. à l'établissement ou la préservation de l'invariant est générée.
Le contrat associé à la fonction `remove` spécifie que la valeur `v` à retirer doit appartenir au domaine [2] et que son effet est de retirer cette valeur. Pour exprimer cette propriété, le modèle abstrait, soit la variable fantôme `setD`, est utilisé. L'opérateur `old`, utilisé dans une post-condition, fait référence à la valeur d'un terme au point d'appel de la fonction. Une condition de vérification correspondant à la satisfaction de cette post-condition est générée.

```
type tsparse = { n : int;
    mutable dense: array int;  mutable sparse: array int;
    mutable sizeD: int;
    mutable ghost setD : fset int; }
invariant {
dom_ran dense n  && dom_ran sparse n  && 0<=sizeD<=n &&
setD ⊆ [0, n-1] &&
(forall x. 0<=x<n -> (sparse[x]<sizeD <-> x ∈ setD)) &&
(forall i. 0<=i<n -> sparse[dense[i]]=i) }

let remove (v : int) (a : tsparse)
requires 0<=v<a.n && v ∈ a.setD
ensures {a.setD == (old a.setD) - {v}}
=
swap_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
a.setD <- a.setD - {v}
```

Listing 1 – *Sparse set* en WhyML (extrait)

Un *sparse set* est également peu coûteux à restaurer dans le cadre d'une recherche de solution utilisant le *backtracking*. Il suffit en effet de sauvegarder et restaurer la limite `sizeD`. Nous retrouvons le même ensemble mathématique, même si les deux tableaux sont différents. Cette situation est illustrée par un exemple dans la figure 2.



(a) Sauvegarde (b) Après retrait de 1, 6 et 3

(c) Restauration ($D_2 = D_0$)

FIGURE 2 – Restaurer un *sparse set*

Nous avons implémenté une fonction `undo` qui prend en paramètre la valeur de `sizeD` à restaurer. Si le code de la fonction est extrêmement simple, sa spécification est plus délicate et demande de revoir la définition du type `tsparse` ainsi que la plupart des opérations. Nous introduisons pour cela, dans `tsparse`, une nouvelle variable fantôme, `states`, qui stocke les différents états du domaine, à savoir les valeurs successives de l'ensemble `setD`. Elle est définie comme une fonction partielle qui associe à `p` le modèle de cardinal `p` [3]. Ainsi `undo(p)` consiste à revenir à la situation antérieure où le modèle `setD` était `states(p)`. La variable `states` est soumise à l'invariant spécifié par le prédicat `inv_states` (cf listing 2), qui est ajouté comme invariant du type `tsparse`. La dernière propriété fait le lien entre `states` et `sparse` : si `states(i) = s` alors `s = dense[0..i]` (et donc tous les éléments de s ont une valeur dans `sparse` strictement inférieure à i).

```
predicate inv_states states setD n sparse =
-- le domaine de définition est inclus dans [|setD|,n]
(dom states) ⊆ [|setD|, n] &&
```

2. Cette pré-condition est ajoutée ici pour simplifier la présentation. Le code vérifié et extrait ne fait pas cette hypothèse et teste l'appartenance de `v` au domaine.

3. le cardinal de `setD` est exactement `sizeD`.

```
-- l'état courant y est enregistré
states (|setD|) = setD &&
-- chaque état stocké est bien formé et contient setD
forall i, s. states(i) = s ->
  s ⊆ [0, n-1] &&  |setD| = i &&  setD ⊆ s
-- lien avec sparse
(forall i, s. 0<=i<=n -> states(i) =  s ->
  (forall x. 0<=x<n -> (sparse[x]<i <-> x ∈ s)))

let undo (a : tsparse) (p : int) : unit
requires {a.sizeD< p<= a.n && p ∈ dom(a.states)}
ensures {exists s. (old a).states(p) = s && a.setD == s}
ensures {forall i, v. p < i <= a.n ->
  (a.states(i) = v <-> (old a).states(i) = v)}
=
let ghost (v : fset int) = a.states(p) in
a.setD <- v ; a.states <- [0, p-1] ⩤ a.states ;
a.sizeD <- p
```

Listing 2 – Invariant de `states` et `undo` en WhyML

La fonction `undo` (cf listing 2) a une large part "fantôme" qui consiste à mettre à jour `setD` et `states`. L'opérateur ⩤ permet de supprimer les potentiels éléments de l'intervalle [0, p-1] du domaine de `states`. Nous assurons ainsi que les états stockés sont antérieurs à celui caractérisé par `p`.
La fonction `undo` a une pré-condition contraignant son paramètre à être dans le domaine de définition de `states`. Si la fonction extraite est utilisée sans vérifier cette pré-condition, l'invariant pourrait être violé. Une version défensive, vérifiant dans le corps de `undo` que le paramètre `p` est bien un "cardinal rencontré précédemment", a également été développée et vérifiée, impliquant d'utiliser, pour l'ensemble de la formalisation, un entier réversible pour la limite `sizeD`. On dispose ainsi non seulement de la valeur courante de `sizeD` mais aussi de ses valeurs antérieures. Le type `rint` d'un entier réversible est donné ci-dessous.

```
type rint = {mutable value : int;
             mutable back : list int;}
invariant {sorted back && no_duplicates back &&
           forall x. x ∈ back -> value < x}
```

Ainsi, dans la fonction `remove`, la valeur courante de l'entier réversible `sizeD` est poussée dans la liste `sizeD.back` avant d'être décrémentée.
Pour assurer la cohérence entre `sizeD` et les états stockés dans `states`, nous ajoutons la propriété suivante dans l'invariant du type `tsparse` :

```
forall x. (x = sizeD.value || x ∈ sizeD.back) <->
          x ∈ dom states
```

Les preuves des VCs sont automatiques, moyennant l'ajout de quelques lemmes, eux aussi prouvés automatiquement. Pour plus de détails, se reporter au code en ligne (https://gitlab.com/cdubois/why3_sparsesets) et à [2].

4 *Sparse subsets* vérifiés

Cette section présente la formalisation en Why3 des domaines des variables qui modélisent des ensembles d'entiers, plus exactement des sous-ensembles d'un intervalle $[0, N-1]$. Le domaine d'une variable X est alors représenté par un intervalle d'ensembles, dont la borne inférieure, appelée $required$, contient les éléments qui doivent être dans l'ensemble X, et la borne supérieure, appelée $possible$, contient les éléments qui peuvent apparaître dans l'ensemble X [4]. Ainsi le domaine de X est l'ensemble

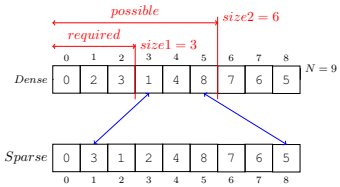


FIGURE 3 – Un *sparse subset*

de tous les sous-ensembles de $[0, N-1]$ qui contiennent $required$ et sont contenus dans $possible$: $\{s \subseteq [0, N-1] \mid\mid required \subseteq s \subseteq possible\}$. Par exemple, l'intervalle d'ensembles $[\{1,3\}, \{1,2,3,4\}]$ représente le domaine ensembliste $\{\{1,3\}, \{1,2,3\}, \{1,3,4\}, \{1,2,3,4\}\}$.
La représentation concrète vérifiée réutilise et adapte la structure de données de *sparse set*, comme cela est proposé par Le Clément de Saint-Marcq et al. dans [5] et implémenté, par exemple, dans le solveur OscaR. Nous appelons cette structure de données un *sparse subset*. L'adaptation consiste en la manipulation de deux limites, $size_1$ et $size_2$, au lieu d'une seule. Le sous-tableau $dense[0..size_1 - 1]$ contient les éléments requis (c'est-à-dire la borne inférieure de l'intervalle d'ensembles), le sous-tableau $dense[0..size_2 - 1]$ contient les éléments possibles (c'est-à-dire la borne supérieure de l'intervalle d'ensembles), le reste du tableau $dense$ contient, quant à lui, les éléments qui ont été exclus du domaine au cours des filtrages successifs. Un exemple est présenté à la figure 3, il représente l'intervalle d'ensembles $[\{0,2,3\}, \{0,1,2,3,4,8\}] \subseteq \mathbb{P}([0,8])$.
La spécification d'un *sparse subset* est présentée en WhyML dans le listing 3. Nous retrouvons la taille `n`, les deux tableaux `dense` et `sparse`, les deux limites rassemblées en un couple "réversible", `size12` (de type `rpair`), ceux-ci constituant la partie calculatoire, ainsi que les variables fantômes `setD`, `required` et `possible` qui servent à la spécification. Comme précédemment, cette structure de données est réversible : pour restaurer un *sparse subset*, il suffit de restaurer les valeurs des deux limites, donc la valeur du couple `size12`. Nous introduisons donc également une variable fantôme `states`, qui stocke les états successifs de la structure de données, c'est-à-dire ici un couple composé des deux ensembles `required` et `possible`. Elle est, de la même façon que pour les *sparse sets*, une fonction partielle définie sur des couples d'entiers. Les propriétés invariantes sont similaires à celles pour les *sparse sets*. Nous retrouvons, dans l'ordre, les propriétés qui décrivent le modèle abstrait, la propriété P_2 qui impose à `dense` et `sparse` d'être inverses l'un de l'autre, les propriétés qui font le lien entre les ensembles abstraits du modèle et les bornes $size_1$ et $size_2$ (`get1` et `get2` sont resp. la première et la deuxième projection des couples), la propriété spécifiant `states` et enfin la propriété assurant la cohérence entre `states` et le couple réversible des deux bornes. Le prédicat `inv_states_set states`, non développé ici, est similaire à celui du listing 2, il repose sur la définition d'un ordre sur les couples qui reflète le fait qu'au cours des filtrages successifs, l'ensemble `required` grossit (et donc $size_1$ augmente) alors que `possible` rétrécit (et donc $size_2$ décroît). Le champ `back` d'un couple réversible

d'entiers forme une liste triée selon cet ordre.

```
type tsparsesubset = {
  n : int;
  mutable dense: array(int); mutable sparse: array(int);
  mutable size12: rpair;
  ghost mutable setD: fset (fset int);
  ghost mutable required: fset int;
  ghost mutable possible: fset int;
  ghost mutable states: fmap (int,int) (fset int, fset int) }
invariant {
possible ⊆ (interval 0 n) &&
required ⊆ possible &&
forall s. s ∈ setD <->(required ⊆ s ⊆ possible) &&
dom_ran dense n && dom_ran sparse n &&
(forall i. 0<=i<n -> sparse[dense[i]]=i) &&
0<=get1(size12.value)<= n  &&
get1(size12.value)<=get2(size12.value) <= n &&
(forall x. 0<=x<n ->
   (sparse[x]<get1(size12.value) <-> x ∈ required)) &&
(forall x. 0<=x<n ->
   (sparse[x]<get2(size12.value) <-> x ∈ possible)) &&
inv_states_set states required possible size12 sparse &&
forall x. (x = sizeD.value || x ∈ sizeD.back) <->
          x ∈ dom states }
```

Listing 3 – Type `tsparsesubset` en WhyML

Nous avons spécifié, implémenté et vérifié les opérations `create`, `full_interval`, `excludes_val`, `requires_val` (cf listing 4), `excludesAll`, `requiresAll...`, inspirées de celles développées dans OscaR ainsi qu'une opération `undo`. Les opérations `excludes`, `requires_val`, `excludesAll`, `requiresAll`, sont en temps constant et consistent en des échanges et des modifications des bornes. L'opération `undo` prend cette fois en paramètre deux entiers censés décrire une "situation antérieure", son code WhyML consiste à mettre à jour `size12` par le couple formé des deux paramètres, ainsi que les variables fantômes (exactement comme dans le cas des *sparse sets*).

```
--La fonction impose v parmi les valeurs requises - Elle
lève une exception si v est déjà exclue.
let requires_val (v : int) (a: tsparsesubset) : unit
requires {0<=v<a.n}
ensures {a.required == (old a.required) ∪ {v}}
ensures {forall x. x ∈ a.setD <->
           exists y. y ∈ (old a).setD && x == y ∪ {v}}
raises {ImpossibleOp ->  v ∉ a.possible) && a = old a}
=
let i = a.sparse[v] in
let (size1, size2) = a.size12.value in
if i < size1 then ()
else
  if i >= size2 then raise ImpossibleOp
  else
  begin
  swap_two_arrays a.dense a.sparse a.n a.sparse[v] size1;
  set (size1+1, size2) a.size12;
  a.required <- a.required ∪ {v};
  a.setD <- addSet[4] v a.setD;
  a.states <- a.states ∪
                {(a.size12, (a.required, a.possible))}
  end
```

Listing 4 – Type `tsparse` en WhyML

Afin d'extraire un code OCaml exécutable efficace, travaillant sur des entiers machine bornés et non des entiers mathématiques[5], il reste une dernière étape. L'implémentation présentée dans cette section est modifiée pour utiliser des entiers machine, et ce, sans difficulté : remplacement du type `int` par le type `int63` de la bibliothèque standard de Why3 et insertion de quelques conversions dans les parties logiques qui continuent de manipuler des entiers mathématiques. Une modification a dû toutefois être introduite dans les spécifications pour ne pas occasionner d'*overflow* dans l'opération *size* qui calcule la hauteur du treillis des sous-ensembles sous-jacent. Ainsi `a.n` est contraint à être strictement inférieur à `max_int`. Les preuves des VCs demeurent automatiques. Le code complet est disponible à l'adresse suivante : https://gitlab.com/cdubois/why3_sparsesubsets.

4. `addSet` est une fonction logique qui ajoute son premier paramètre dans chacun des ensembles de son deuxième paramètre.

5. le type `int` de WhyML désigne le type des entiers mathématiques

5 Conclusion

Le travail présenté ci-dessus a permis de mettre à disposition une bibliothèque d'implémentations formellement vérifiées pour représenter des domaines de variables dans un solveur de contraintes.
Le développement réalisé en Why3 pour les domaines ensemblistes a bénéficié grandement de celui réalisé au préalable concernant la représentation des *sparse sets*. Il est plus complexe dans la mesure où l'on est amené à manipuler un couple d'entiers (`size12`) au lieu d'un entier unique (`sizeD`) et deux ensembles (`required` et `possible`) au lieu d'un seul (`setD`) mais les spécifications sont très similaires. Il est sans doute possible de réutiliser une partie du code WhyML en généralisant certaines propriétés, ce qui n'a pas été fait ici. Nous envisageons également d'explorer en Why3 la représentation basée sur les *sparse bitsets* [1].

Références

[1] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. Régin, and P. Schaus. Compact-table : Efficiently filtering table constraints with reversible sparse bit-sets. In *Principles and Practice of Constraint Programming (CP 2016), Toulouse, France*, volume 9892 of *LNCS*, pages 207–223. Springer.

[2] C. Dubois. Deductive Verification of Sparse Sets in Why3. In *Verified Software. Theories, Tools and Experiments, Prague, Czech Republic (VSTTE 2024)*, volume 15525 of *LNCS*, pages 28–46. Springer.

[3] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *European Symposium on Programming (ESOP 2013), Rome, Italy*, volume 7792 of *LNCS*, pages 125–128. Springer.

[4] C. Gervet. Conjunto : Constraint propagation over set constraints with finite set domain variables. In *International Conference on Logic Programming, Santa Marherita Ligure, Italy*. MIT Press, 1994.

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

[6] A. Ledein and C. Dubois. Facile en coq : vérification formelle des listes d'intervalles. In *Journées Francophones des Langages Applicatifs (JFLA 2019), Gruissan, France*.