L'arithmétique de séparation

Jean-Christophe Filliâtre, Andrei Paskevich

L'arithmétique de séparation

Jean-Christophe Filliâtre et Andrei Paskevich

[1] LMF — Université Paris-Saclay, CNRS, ENS Paris-Saclay
[2] Inria

Résumé

Nous, praticiens de la preuve de programmes, souhaitons que le processus de la vérification soit le plus automatique possible. Les meilleurs outils pour cela sont à l'heure actuelle les démonstrateurs SMT, qui combinent notamment la logique du premier ordre et l'arithmétique linéaire. Par opposition, le raisonnement inductif n'est pas un point fort des démonstrateurs automatiques. Or, les programmes utilisant des pointeurs le font souvent pour manipuler des structures récursives : listes, arbres, etc.

Dans cet article, nous décrivons une approche qui permet d'amener la preuve de programmes avec pointeurs à la portée des démonstrateurs automatiques. L'idée consiste à projeter une structure récursive sur un domaine numérique, de sorte que les propriétés de possession et de séparation deviennent exprimables en terme de simples inégalités arithmétiques. En plus de simplifier la preuve, cela permet une spécification claire et naturelle. On illustre cette approche avec l'exemple classique du renversement en place d'une liste simplement chaînée.

1 Introduction

Dans le domaine de la preuve de programmes, et plus spécifiquement dans le domaine de la vérification déductive, nous sommes aujourd'hui grandement aidés par les démonstrateurs SMT, qui combinent notamment la logique du premier ordre et l'arithmétique linéaire. Ils nous permettent parfois d'obtenir des preuves entièrement automatiques, là où hier encore il fallait laborieusement construire une preuve interactivement et la faire vérifier par un assistant de preuve.

Les démonstrateurs SMT ont cependant des limites. En particulier, le raisonnement inductif n'est pas leur point fort. Or, les programmes utilisant des pointeurs le font souvent pour manipuler des structures qui sont récursives par nature, telles que des listes ou des arbres. Il est alors naturel de recourir à des définitions récursives pour les propriétés que l'on cherche à démontrer. Il y alors une tension évidente entre deux principes, à savoir d'une part le principe qui affirme

La bonne spécification, c'est celle qui est facile à comprendre.

qui va dans le sens d'une définition récursive, naturelle pour la personne qui va lire et approuver la spécification, et le principe qui affirme d'autre part

Le bon invariant, c'est celui qui est facile à prouver.

qui va plutôt dans le sens d'une définition non récursive, plus adaptée aux démonstrateurs SMT.

Dans cet article, nous décrivons une approche qui permet dans certains cas de réconcilier ces deux principes, en amenant la preuve de programmes avec pointeurs à la portée des démonstrateurs automatiques. Nous l'illustrons sur l'exemple classique du renversement en place d'une liste simplement chaînée. L'idée clé consiste à se ramener à de simples inégalités dans l'arithmétique linéaire, un domaine de prédilection des démonstrateurs SMT, ce qui justifie le titre de cet article.

L'essentiel de cet article est composé autour de l'exemple du renversement d'une liste, avec sa spécification naturelle (section 2), une tentative échouée de preuve directe (section 3) et la proposition de notre approche arithmétique (section 4), que nous illustrons ensuite rapidement sur un autre cas d'étude (section 5). Nos expériences ont été conduites avec l'outil Why3 [2] et le code complet est donné en annexe. Toutefois, dans le but de donner plus d'universalité à notre propos, les éléments de spécification et de preuve sont donnés dans une syntaxe allégée.

2 Un grand classique

On illustre notre propos avec l'exemple classique du renversement en place d'une liste simplement chaînée. La figure 2 contient un code C qui implémente cet algorithme et illustre son fonctionnement sur une liste de cinq éléments. L'algorithme parcourt la liste une fois, avec la variable `l`, et redirige le pointeur `next` sur l'élément précédent, stocké dans la variable `r`.

```
typedef struct Cell list;
struct Cell { int value; list *next; };

list *list_reversal(list *l) {
  list *r = NULL;
  while (l != NULL) {
    list *tmp = l;
    l = l->next;
    tmp->next = r;
    r = tmp;
  }
  return r;
}
```

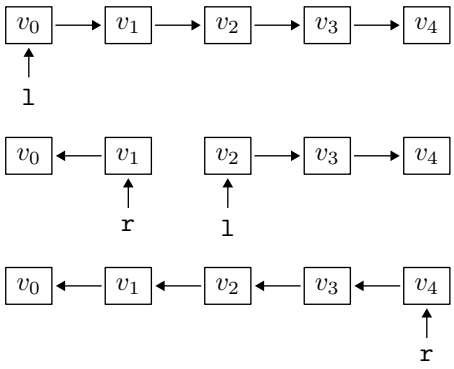


FIGURE 1 – Renversement en place d'une liste chaînée.

L'intérêt de ce programme pour la vérification est la coexistence, pendant la durée de la boucle, de deux listes simplement chaînées. Le fait que ces deux listes sont spatialement séparées nous permet d'établir la préservation des invariants. Plus précisément, l'affectation du pointeur `tmp->next` dans le code préserve la queue de la liste `l` d'une part et la liste `r` d'autre part.

Une spécification de la fonction `list_reversal` doit exprimer que les cellules de la liste initiale sont chaînées dans l'ordre inverse à l'issue de la fonction et le résultat de la fonction est la tête de cette nouvelle liste. Notons que la spécification ne peut pas se contenter de parler seulement du *contenu* des cellules, ce qui serait moins précis. En effet, le code pourrait par exemple inverser les valeurs contenues dans les cellules, mais laisser les champs `next` inchangés.

Pour écrire confortablement la spécification de la fonction, on doit représenter dans la logique, en plus des pointeurs manipulés par le programme, la mémoire elle-même (qu'on va appeler `mem`) et le modèle mathématique de la liste, sous la forme d'une séquence d'adresses mémoire (qu'on va appeler `s`). Étant donnés `mem`, `l` et `s`, nous pouvons définir un *prédicat de représentation*, `list mem l s`, qui exprime que, à l'adresse `l` dans la mémoire `mem`, se trouve une liste simplement chaînée dont les cellules forment la séquence `s`. En particulier, `l` est égal à `NULL` si et seulement si la séquence `s` est vide. Il est naturel de définir ce prédicat récursivement, sur la longueur de la séquence `s`.

```
predicate list mem l s =
  if length s = 0 then l = NULL
  else l = s[0] <> NULL && list mem[l].next s[1:]
```

Ici, `mem[l].next` désigne le contenu du champ `next` de la cellule à l'adresse `l` dans `mem` et `s[1:]` désigne la séquence `s` privée de son premier élément.

La spécification de `list_reversal` est alors très simple, en supposant donnée une fonction logique `rev` qui renverse une séquence :

```
list_reversal l (ghost s) : r
  requires list mem l s
  modifies mem
  ensures  list mem r (rev s)
```

Ici, la séquence `s` est fournie comme un argument fantôme [3] de la fonction `list_reversal`, c'est-à-dire un argument qui n'est là que pour les besoins de la spécification.

La variable globale `mem` représente l'état de la mémoire : dans la précondition, à l'entrée de la fonction ; dans la postcondition, à la sortie. La spécification ci-dessus est incomplète si on ne dit pas que la mémoire reste inchangée en dehors des adresses qui composent la liste `l`. On peut exprimer cette propriété avec une seconde postcondition

```
  ensures  forall p. not (mem p s) -> mem[p] = old mem[p]
```

où `old mem[p]` désigne le contenu de la mémoire à l'adresse `p` à l'entrée de la fonction.

3 Droit dans le mur

Dans cette section, nous allons montrer en quoi la spécification ci-dessus, aussi naturelle qu'elle soit, ne se prête pas facilement à une preuve automatique. La première étape de la vérification consiste à exhiber les invariants de boucle. Ils se déduisent trivialement de la figure 2, en exprimant que `l` est un suffixe de la liste initiale et `r` un renversement du préfixe correspondant. En se donnant une variable fantôme `i` qui maintient le nombre de cellules déjà renversées, on a donc les deux invariants suivants

```
invariant list mem l s[i:]
invariant list mem r (rev s[:i])
```

où `s[i:]` est le suffixe de `s` à partir de la position `i` incluse et `s[:i]` le préfixe de `s` jusqu'à la position `i` exclue. En particulier, la sortie de boucle, où `i` est égal à la longueur de `s`, nous donne exactement la postcondition attendue. Jusque là, tout est parfaitement naturel et plutôt simple.

Les choses se compliquent lorsque l'on s'attaque à prouver que ces invariants de boucle sont préservés par une itération de la boucle. En effet, considérons l'état du programme à la fin d'une itération. L'invariant sur la nouvelle valeur de `l` se déduit directement de l'invariant sur sa valeur précédente, car on a uniquement suivi le champ `next` et `s[i+1:]` est égal à `s[i:][1:]`. L'invariant sur la nouvelle valeur de `r` se déduit de l'invariant sur la valeur précédente, car à la tête de `rev s[:i+1]` on trouve `r` et `(rev s[:i+1])[1:]` est égal à `rev s[:i]`.

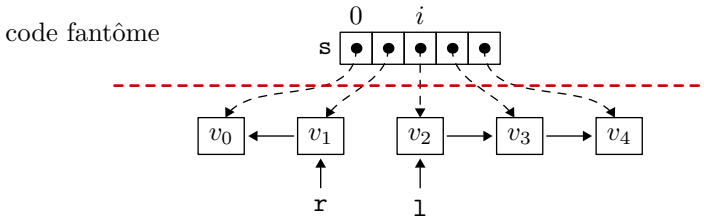
La difficulté consiste alors à prouver que les deux portions de la liste, à gauche de `r` et à droite de `l`, ont conservé la propriété `list`, puisque la modification de `tmp->next` n'a pas affecté les cellules qui les composent. La définition de `list` étant récursive, cette preuve nécessite une induction. Bien qu'il soit possible d'effectuer une telle induction, par exemple en utilisant un

démonstrateur comme Coq ou un autre moyen de preuve interactive, on préférerait une solution qui soit entièrement à la portée d'un démonstrateur automatique.

4 Approche arithmétique

La définition récursive du prédicat `list` introduite dans la section 2 nous pousse à représenter tout fragment de liste par un objet mathématique différent (ici une séquence). On a vu dans la section précédente en quoi cette approche est inadaptée à la preuve. Or, nous pouvons choisir une approche différente, "bas niveau", et naviguer dans une description *globale* à l'aide d'indices.

Reprenons la séquence `s` des adresses mémoire des cellules de la liste initiale. À toute itération de l'algorithme, un préfixe $[0, i[$ de cette séquence, lu de droite à gauche, correspond à la liste `r` et le suffixe $[i, n[$ correspond à la liste `l`, ce que l'on peut représenter ainsi :



Pour l'instant, on va supposer de plus que les éléments de `s` sont non nuls et deux à deux distincts ; nous montrerons plus loin que ces deux propriétés peuvent être déduites du prédicat de représentation `list`. C'est ce que décrit le prédicat suivant :

```
predicate valid_cells s =
  (forall i. 0 <= i < length s -> s[i] <> null) &&
  (forall i j. 0 <= i,j < length s -> i <> j -> s[i] <> s[j])
```

On peut maintenant définir deux nouveaux prédicats de représentation pour une liste correspondant à *un préfixe* ou *un suffixe* de `s` :

```
predicate prefix mem s r i =
  0 <= i <= length s &&
  if i = 0 then r = null else
    r = s[i-1] && mem[s[0]].next = null &&
    forall k. 0 < k < i -> mem[s[k]].next = s[k-1]

predicate suffix mem s l i =
  0 <= i <= length s &&
  if i = length s then l = null else
    l = s[i] && mem[s[length s - 1]].next = null &&
    forall k. i < k < length s -> mem[s[k-1]].next = s[k]
```

Ces prédicats remplacent la récursivité par un quantificateur universel sur les indices à l'intérieur de `s`. C'est pourquoi on parle ici d'approche *arithmétique*.

Avec ces définitions, la fonction `list_reversal` a maintenant une spécification un peu différente de celle de la section 2 :

```
list_reversal (ghost s) l : r
```

```
  requires valid_cells s
  requires suffix mem s l 0
  modifies mem
  ensures  prefix mem s r (length s)
  ensures  forall p. not (mem p s) -> mem[p] = old mem[p]
```

Mais surtout, les *invariants* de la boucle s'expriment eux aussi en termes des prédicats `suffix` et `prefix`,

```
invariant suffix mem s l i
invariant prefix mem s r i
```

et c'est là la clé d'une démonstration aisée. De fait, la preuve ne nécessite plus de raisonnement par induction et est maintenant entièrement à la portée des démonstrateurs automatiques.

Retomber sur ses pattes. Pour être complets, il nous reste cependant à faire le lien entre la spécification naturelle, introduite dans la section 2, et la preuve ci-dessus. Pour cela, il convient d'établir, à partir de la définition du prédicat récursif `list`, les deux préconditions de la fonction `list_reversal`, c'est-à-dire, `valid_cells` et `suffix`.

On peut le faire agréablement à l'aide d'une fonction fantôme pure, dite *fonction-lemme*, dont le seul objectif est de construire cette preuve.

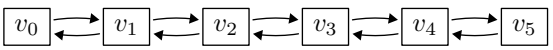
```
lemma cells_of_list l s
  requires list mem l s
  ensures  valid_cells s
  ensures  suffix mem s l 0
  variant  length s
= if s <> empty then cells_of_list mem[l].next s[1:]
```

Cette fonction récursive suit la structure du prédicat `list`. De facto, le code de cette fonction représente une preuve par induction, écrite par l'utilisateur. Par le mécanisme de vérification de Why3, cette fonction, et donc le lemme, peuvent être vérifiés de manière purement automatique, quand bien même l'énoncé du lemme est en soi hors de portée d'un démonstrateur automatique. De la même façon, on peut écrire et prouver une fonction-lemme `list_of_cells` qui conclut `list mem r (rev s)` à partir des hypothèses `valid_cells` et `prefix`.

En composant ces deux lemmes et la fonction `list_reversal`, on obtient une fonction vérifiée, avec la spécification naturelle introduite au début de cet article.

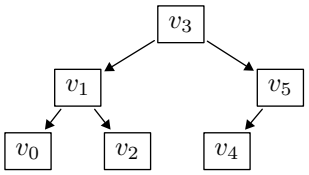
5 Un autre exemple

Nous décrivons ici un autre exemple de la même approche, un peu plus complexe. Il s'agit d'un problème proposé en 2021 dans le cadre de la compétition de preuve de programmes VerifyThis [1]. Un algorithme pour convertir une liste doublement chaînée en arbre binaire était donné, et il fallait en proposer une implémentation vérifiée. Plus précisément, on part d'une liste doublement chaînée contenant une séquence de valeurs $v_0, v_1, \ldots$:



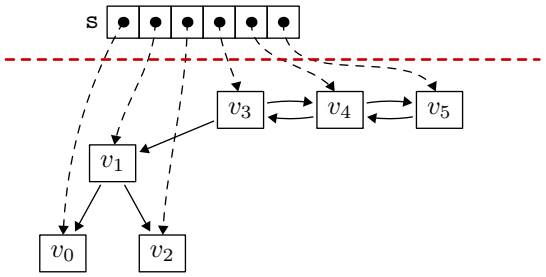
Chaque cellule contient un pointeur `prev` vers la cellule précédente (ou la valeur `null` pour la première cellule) et un pointeur `next` vers la cellule suivante (ou la valeur `null` pour la dernière cellule). L'algorithme réorganise alors les pointeurs `prev` et `next` pour former un arbre binaire,

le pointeur `prev` pointant maintenant vers le sous-arbre gauche et le pointeur `next` vers le sous-arbre droit.



Pour cela, l'algorithme commence par calculer la longueur n de la liste, puis se repose sur une fonction récursive qui prend en paramètres un pointeur vers le début de la liste et un nombre d'éléments à considérer, et qui renvoie la racine de l'arbre construit et le pointeur vers le premier élément non consommé. L'ordre des éléments est conservé, au sens où un parcours infixe de l'arbre énumère les valeurs $v_0, v_1, \ldots$ dans l'ordre.

Parmi les propriétés de l'algorithme à vérifier, il y avait le caractère équilibré de l'arbre obtenu au final, ou encore le fait que, si les valeurs v_i sont triées par ordre croissant, alors l'arbre obtenu est un arbre binaire de recherche. Pour cette vérification, nous pouvons avantageusement utiliser l'approche arithmétique. Comme pour le renversement de liste, on introduit une séquence fantôme `s` de toutes les cellules mémoires :



Sur la base de cette séquence, il est maintenant possible de définir un prédicat `dll s p lo hi` qui exprime que le pointeur `p` est la première cellule d'une liste doublement chaînée formée par les cellules de `s` situées entre les indices `lo` inclus et `hi` exclu, et de même un prédicat `tree s p lo hi` qui exprime que le pointeur `p` est la racine d'un arbre binaire dont les nœuds, dans l'ordre infixe, sont les cellules de `s` entre `lo` et `hi`. Une preuve Why3 réalisant cette idée est disponible en ligne [4]. Elle est majoritairement automatique.

6 Conclusion

Sur la base de deux exemples, nous avons montré comment, dans le contexte de la vérification déductive, une spécification récursive peut être avantageusement remplacée par une spécification ne faisant intervenir que des inégalités arithmétiques, et conduire alors à une preuve bien plus automatique. Nous anticipons que cette technique s'applique au-delà de ces deux exemples, sur toute structure récursive où un ordre de parcours définit une sérialisation de ses éléments. Pour poursuivre cette étude, il conviendrait d'explorer aussi des programmes qui allouent dynamiquement de la mémoire.

Bien évidemment, l'approche que nous avons présentée ici n'a aucune prétention à être aussi expressive et puissante que la logique de séparation [7]. Néanmoins, elle a le mérite de ne pas nécessiter une logique spécifique et des outils dédiés. Puisque nous travaillons dans un modèle mémoire plat, nous pouvons par ailleurs nous abstraire de toute problématique de ressource,

de possession, d'alias, etc. Et puisque notre représentation logique est construite sur la base d'adresses explicites, elle nous donne immédiatement accès à l'empreinte mémoire (*footprint*) de la structure. On se rapproche là des *dynamic frames* [5], une autre technique de preuve de programmes avec pointeurs utilisée par exemple dans Dafny [6]. Mais contrairement à cette approche, nos empreintes sont dotées d'une structure supplémentaire, à savoir leur caractère séquentiel, ce qui nous permet justement de faire de l'*arithmétique de séparation*.

Remerciements. Nous remercions nos collègues Claude Marché et Jacques-Henri Jourdan pour toutes les discussions relatives à la vérification du renversement d'une liste.

Références

[1] The VerifyThis competition, Since 2011. https://www.pm.inf.ethz.ch/research/verifythis.html.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform. http://why3.lri.fr/.

[3] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. Solution to verifythis 2021 challenge 2, 2021. https://toccata.gitlabpages.inria.fr/toccata/gallery/verifythis_2021_dll_to_bst.en.html.

[5] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.

[6] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[7] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

A Le source Why3 complet

Le code ci-dessous a été vérifié avec la version 1.5.1 de Why3 (https://why3.lri.fr/) et les deux démonstrateurs SMT Alt-Ergo 2.4.0 (https://alt-ergo.ocamlpro.com/) et CVC5 1.0.0 (https://github.com/cvc5/cvc5/). La session Why3 peut être téléchargée à l'adresse https://www.lri.fr/~filliatr/jfla-2023/list_reversal.zip et ouverte avec les commandes suivantes :

```
$ unzip -x list_reversal.zip
$ cd jfla-2023
$ why3 ide list_reversal
```

Le code Why3 commence par modéliser les pointeurs (type `loc`) et la mémoire (variable globale `mem`), car les structures récursives mutables ne sont pas acceptées nativement par Why3. La séquence `s` est représentée par une simple fonction de type `int -> loc`.

```
(* Le modèle mémoire *)

  use int.Int
  use map.Map
```

```
  type loc

  val (=) (l1 l2: loc) : bool ensures { result <-> l1 = l2 }

  val constant null : loc

  type mem = { mutable next: loc -> loc }

  val mem: mem

  let cdr (p: loc) : loc
    requires { p <> null }
    ensures  { result = mem.next p }
  = mem.next p

  let set_cdr (p: loc) (v: loc) : unit
    requires { p <> null }
    ensures  { mem.next = (old mem.next)[p <- v] }
  = let m = mem.next in
    mem.next <- fun x -> if x = p then v else m x

(* Les définitions de prédicats, avec l'approche arithmétique *)

  predicate valid_cells (s: int -> loc) (n: int) =
    (forall i. 0 <= i < n -> s i <> null) &&
    (forall i j. 0 <= i < n -> 0 <= j < n -> i <> j -> s i <> s j)

  predicate listLR (m: mem) (s: int -> loc) (l: loc) (lo hi: int) =
    0 <= lo <= hi &&
    if lo = hi then l = null else
      l = s lo && m.next (s (hi-1)) = null &&
      forall k. lo <= k < hi-1 -> m.next (s k) = s (k+1)

  predicate listRL (m: mem) (s: int -> loc) (l: loc) (lo hi: int) =
    0 <= lo <= hi &&
    if lo = hi then l = null else
      m.next (s lo) = null && l = s (hi-1) &&
      forall k. lo < k < hi -> m.next (s k) = s (k-1)

  predicate frame (m1 m2: mem) (s: int -> loc) (n: int) =
    forall p. (forall i. 0 <= i < n -> p <> s i) ->
      m1.next p = m2.next p

(* Le code *)

  let list_reversal (ghost s: int -> loc) (ghost n: int) (l: loc) : (r: loc)
    requires { valid_cells s n }
    requires { listLR mem s l 0 n }
```

```
    ensures  { listRL mem s r 0 n }
    ensures  { frame mem (old mem) s n }
  = let ref l = l in
    let ref r = null in
    let ghost ref i = 0 in
    while l <> null do
      invariant { if n = 0 then l = r = null else
                    i = 0     && r = null    && l = s 0
                 || i = n     && r = s (n-1) && l = null
                 || 0 < i < n && r = s (i-1) && l = s i }
      invariant { listRL mem s r 0 i }
      invariant { listLR mem s l i n }
      invariant { frame mem (old mem) s n }
      variant   { n - i }
      let tmp = l in
      l <- cdr l;
      set_cdr tmp r;
      r <- tmp;
      i <- i + 1
    done;
    return r

(* Avec la spécification récursive cette fois *)

  let rec ghost predicate is_list (m: mem) (l: loc) (s: int -> loc) (n: int)
    requires { n >= 0 }
    variant  { n }
  = if n = 0 then l = null else
      l = s 0 <> null && is_list m (m.next l) (fun i -> s (i+1)) (n - 1)

  let rec lemma cells_of_list (l: loc) (s: int -> loc) (n: int)
    requires { n >= 0 }
    requires { is_list mem l s n }
    variant  { n }
    ensures  { valid_cells s n }
    ensures  { listLR mem s l 0 n }
  = if n <> 0 then cells_of_list (cdr l) (fun i -> s (i+1)) (n - 1)

  let rec lemma list_of_cells (r: loc) (s: int -> loc) (n: int)
    requires { n >= 0 }
    requires { valid_cells s n }
    requires { listRL mem s r 0 n }
    variant  { n }
    ensures  { is_list mem r (fun i -> s (n-1-i)) n }
  = if n <> 0 then list_of_cells (cdr r) s (n - 1)

  let list_reversal_final (ghost s) (ghost n: int) (l: loc) : (r: loc)
    requires { n >= 0 }
```

```
  requires { is_list mem l s n }
  ensures  { is_list mem r (fun i -> s (n-1-i)) n }
  ensures  { frame mem (old mem) s n }
= cells_of_list l s n;
  let r = list_reversal s n l in
  list_of_cells r s n;
  r
```