Rapport d'article

When Separation Arithmetic is Enough

Prudence Deteix, Paul Passeron, Jean Rousseau

Table des matières

Dépot du projet : `https://github.com/Paul-Passeron/PRRD.git`

1 Introduction

L'article "When Separation Arithmetic is Enough" de Filliâtre, Paskevich et Danvy présente une méthode pour la vérification formelle de programmes impératifs manipulant des structures de données récursives basées sur des pointeurs (ex : listes chaînées ou arbres binaires). Les auteurs proposent une solution qui réconcilie deux objectifs souvent contradictoires : maintenir des spécifications claires et compréhensibles tout en permettant une vérification largement automatisée par les solveurs SMT.

L'idée centrale de l'article consiste à projeter les structures récursives sur des séquences plates indexées par des entiers, transformant ainsi les propriétés de séparation (est-ce que plusieurs objets pointent vers le même endroit en mémoire) en contraintes d'arithmétique linéaires.

Dans ce rapport, nous nous concentrerons sur les quatres premières sections de l'article, qui présentent la méthode à travers trois exemples de plus en plus complexes : l'inversion classique de liste, une variante inversant les valeurs en espace constant, et l'algorithme de Morris pour le parcours d'arbres binaires. Nous avons choisi de ne pas traiter les sections 5 et 6, la première présentant un exemple bonus issu de la compétition VerifyThis 2021 qui reprend essentiellement les concepts déjà illustrés, et la seconde constituant une revue de littérature de travaux connexes qui sort du cadre de cet article.

2 Warm Up : Classic List Reversal

Dans cette partie, les auteurs proposent d'introduire leur méthode en présentant une structure de listes chaînées en Ocaml et une fonction revisitée de la manière "classique" de les inverser.

Les listes chaînées mutables sont définies de la façon suivante :

```
1 type ls t=
2     | Nil
3     | Cons of { mutable car: ekt; mutable cdr: lst }
```

2.1 Listes chaînées et renversement classique revisité

L'implémentation choisie repose sur une fonction auxiliaire et récursive prenant deux pointeurs, l'un vers la partie déjà inversée et l'autre vers le segment pas encore traité. À chaque itération de la boucle récursive, le premier élément du segment non traité est détaché et réinséré en tant que tête du

segment inversé et ce, jusqu'à ce que la liste initiale soit vide.

Néanmoins, les auteurs ont précisé dans la partie précédente que les solveurs SMT sont inefficaces quand il y a besoin d'instancier de la récursion. Ainsi, il est proposé dans l'article une nouvelle idée : représenter la structure initiale de la liste sous la forme dune séquence *ghost* dindices entiers.

2.2 Structure fantôme

La construction introduite dans cette partie consiste en un objet abstrait appelé *list_shape*. Celui-ci associe à la liste étudiée une représentation qui ne change pas l'état mémoire réel mais fournit seulement une représentation sous forme de tableau "classique" de cette liste. En effet, cet objet associe les n valeurs contenues dans la liste chaînée à des entiers de 0 à n par ordre d'apparition depuis la tête. Ainsi, cette représentation établit une correspondance bijective entre indices entiers et cellules réelles.

Grâce à ce nouveau modèle, les propriétés structurelles habituellement exprimées par induction peuvent être reformulées en contraintes arithmétiques. Par exemple, la propriété "chaque cellule pointe vers la suivante" peut se traduire par l'invariant "$\forall i \in [0, n-2]$, le pointeur *cdr* de la cellule d'indice i correspond à la cellule $i+1$". Ainsi, on obtient une version de la liste non définie récursivement mais par rapport à une plage d'indices.

2.3 Spécification et automatisation de la preuve

Une fois cette structure définie, lalgorithme de renversement peut être spécifié par des prédicats non récursifs décrivant séparément le segment traité et le segment restant. Ainsi, pour le raisonnement mathématique, déplacer la frontière entre ces deux segments ne se fait plus par récursivité où l'on détache les éléments un à un mais se traduit simplement par lincrément d'un indice. Il est donc simple de formuler l'invariant : "le segment inversé correspond aux indices $[0, i[$ et le segment restant aux indices $[i, n]$".

Les solveurs SMT peuvent alors utiliser et vérifier ces invariants, car ramenés à de l'arithmétique de base et de la logique du premier ordre, domaines dans lesquels ils sont performants.

Finalement, cette première étape prouve l'intérêt de traduire la récursivité par un modèle plat indexé pour l'automatisation de la preuve.

3 Reversing values in constant space

Jusqu'ici, on a pu inverser une liste chaînée en inversant les "maillons" de cette chaîne (les pointeurs, le champ `cdr`). Les valeurs de chaque noeuds (le champ `car`) n'ont pas bougées mais les pointeurs reliants les noeuds entre eux ont été inversés. L'article propose une solution différente pour résoudre le même problème. Sans toucher à la structure même de la liste (les pointeurs), on essaie maintenant d'inverser les valeurs, et ce sans allouer plus de mémoire (*constant space*).

L'algorithme illustré dans le papier utilise trois fonctions :

— `back_again (bp: lst) (sp: lst) (np: lst): unit`

— `tortoise_hare (bp: lst) (sp: lst)`
`(fp: lst) (qp: lst): unit`

— `value_reverse (sp: lst) (qp: lst): unit`

Les fonctions suivantes ne sont pas exactement comme écrites dans le papier mais ont exactement la même valeur sémantique et sont du `OCaml` valide.

3.1 La fonction `back_again`

```
let rec back_again bp sp np = match bp, np with
    | Cons bc, Cons nc ->
        let tmp = bc.car in
        bc.car <- nc.car;
        nc.car <- tmp;
        let nbp = bc.cdr in bc.cdr <- sp;
        back_again nbp bp nc.cdr
    | _ -> ()
```

FIGURE 1 – La fonction `back_again`

`back_again` parcourt deux portions de liste en parallèle et échange leur valeurs.

Les paramètres sont :

— `bp` (*back pointer*) : Pointe vers la portion de la liste déjà traitée (inversée) qui remonte vers le début initial de la liste.
— `sp` (*slow pointer*) : Le noeud après `bp` dans la liste originale et donc le noeud précédent dans la reconstruction de la chaîne inversée
— `np` (*next pointer*) : Pointe vers la portion de la liste que l'on n'a pas traitée (donc vers la fin de la liste).

3.1.1 Explication

— si `bp` ou `np` est vide, alors on ne fait rien. En effet, la fonction n'est pas faite pour traiter le cas où `bp` est vide car on est censé l'appeler depuis `tortoise_hare` uniquement ; qui l'empêche d'être vide par par construction. De plus, lorsque `np` est vide, on a fini.
— `bp` et `np` sont tous les deux des `Cons`.

1. On échange les valeurs de `bp` et `np`.
2. On inverse le sens du pointeur. On sauvegarde le prochain noeud de `bp` dans `nbp` et on fait pointer `bp.cdr` vers `sp` (On inverse les pointeurs mais ils ils sont en fait déjà inversés à cause de `tortoise_hare`. Ils retrouvent donc leur sens original)
3. On récurse avec `nbp` comme nouveau *back pointer* (avance dans la première partie), `bp` comme nouveau *slow pointer* (devient le précédent) et `nc.cdr` comme nouveau *next pointer* (avance dans la deuxième partie).

3.2 La fonction `tortoise_hare`

```
1  let rec tortoise_hare bp sp fp qp =
2  match sp, fp with
3      | _ when fp == qp ->
4          back_again bp sp sp
5      | Cons sc, Cons {cdr = nfp} when nfp == qp ->
6          back_again bp sp sc.cdr
7      | Cons sc, Cons {cdr = Cons {cdr = nfp}} ->
8          let nsp = sc.cdr in sc.cdr <- bp;
9          tortoise_hare sp nsp nfp qp
10     | _ -> () (* Unreachable by assumptions *)
```

FIGURE 2 – La fonction `tortoise_hare`

`tortoise_hare` Utilise l'algorithme du lièvre et de la tortue pour trouver le milieu de la liste, inverser les pointeurs de la première moitié pendant le parcours et ensuite appeler `back_again` pour échanger les valeurs.

Les paramètres sont :

— `bp` (*back pointer*) : Pointe vers la portion de la liste déjà traitée (inversée) qui remonte vers le début initial de la liste.
— `sp` (*slow pointer*) : La "tortue" qui avance d'un noeud à la fois
— `fp` (*fast pointer*) : Le "lièvre" qui avance de deux noeuds à la fois (losrque celui-ci est Nil, alors `sp` est le milieu de la liste).
— `qp` (*queue pointer*) : La fin de la liste (reste fixe et sert à arrêter l'algorithme si on ne veut inverser qu'une portion d'une liste plus grande).

3.2.1 Explication

La fonction à trois cas principaux

Cas 1 : `fp == qp` Le lièvre a atteint la fin de la liste. Cela signifie d'ailleurs que la liste est de longueur paire. La tortue est donc exactement au milieu. On appelle alors `back_again bp sp sp` pour inverser les valeurs. On note que `sp` est passé deux fois en argument car on veut inverser la première moitié (`bp`) avec la seconde moitié `sp`, qui commence originellement juste après le premier noeud de `bp` et qui est le premier `np`.

Cas 2 : `nfp == qp` Le lièvre a atteint l'avant dernier maillon de la liste. Cela signifie que la liste est de longueur impaire. On appelle ainsi `back_again bp sp sc.cdr`. Les deux premiers éléments sont les mêmes que dans le cas 1 et pour la même raison. Le dernier argument est `sc.cdr` car on saute le noeud `sc` étant le vrai milieu (la liste est de longueur impaire) et donc le seul noeud dont la valeur ne change pas.

Cas 3 : Progression normale On fait ce que l'on a décrit plus haut dans le rôle de la fonction. On sauvegarde la prochaine valeur de la tortue que l'on nomme `nsp`. Ensuite, on inverse le pointeur `sc.cdr <- bp` (On construit la moitié de chaîne inversée). Finalement, on fait une récursion où :

— `sp` est le nouveau `bp` (ajoute à la chaîne inversée).
— `nsp` est le nouveau `sp` (on a avancé la tortue d'un noeud).
— `nfp` est le nouveau `fp` (on a avancé le lièvre de deux noeuds).

3.3 La fonction `value_reverse`

```
1 let value_reverse sp qp =
2     tortoise_hare Nil sp sp qp
```

FIGURE 3 – La fonction `value_reverse`

Cette fonction sert à inverser la liste en espace constant. Elle est assez simple puisqu'elle ne fait qu'appeler `tortoise_hare` avec les bons arguments.

Les paramètres sont :

- `sp` (*slow pointer*) : Le premier maillon de la liste à inverser.
- `qp` (*queue pointer*) : La fin de la liste (reste fixe et sert à arrêter l'algorithme si on ne veut inverser qu'une portion d'une liste plus grande).
- `tortoise_hare :: bp` ⇒ `Nil`
  `bp` pour *back pointer* représente la portion de la liste que l'on a déjà inversé. Ici, comme on commence le processus, sa valeur initiale est la liste vide, soit `Nil`
- `tortoise_hare :: sp, fp` ⇒ `sp`
  `sp` et `fp` représentent les deux pointeurs que l'on utilise pour parcourir la liste. Comme on commence le processus, leur valeur initiale est la tête de la liste à inverser, soit `sp`.
- `tortoise_hare :: qp` ⇒ `qp`
  `qp` représente la queue de la liste à inverser. Comme on commence le processus, sa valeur initiale est la queue de la liste à inverser, soit `qp`.

3.4 Espace constant ?

Pour se convaincre que l'algorithme s'effectue bien en espace constant, on peut noter plusieurs choses :

- Comme décrit dans l'article, tous les appels effectués dans les trois fonctions sont des *tail calls* ou appels terminaux / de queue. Cela signifie qu'au lieu de réellement "appeller" les fonctions et de garder sur la pile les informations de l'appel courant, on peut simplement *jump* à la fonction appelée, ce qui signifie que l'espace utilisé sur la pile n'est que le maximum utilisé par l'appel d'une des trois fonctions. Ici, chaque fonction n'utilise qu'un nombre fixe de variables locales : `tmp`, `nbp`, `nfp` et `nsp`. Puisque les valeurs sont des entiers et des pointeurs et qu'elles tiennent sûrement dans des registres, il est bien possible que l'empreinte sur la pile soit nulle (ou au pire constante).

— Aucune allocation dynamique (création d'objet) n'est faite. Aucune cellule `Cons` n'est allouée durant l'éxécution. On modifie simplement les valeurs en place, ce qui assure ici aussi que l'algorithme s'effectue en espace constant.

3.5 Véfication

Cet algorithme étant tout aussi récursif que celui explicité dans l'échauffement, pose ainsi les mêmes problèmes de vérifications, notamment pour les prouveurs SMT. On réutilise alors le type `list_shape` ainsi que les prédicats utilisés précédemment. On peut le constater assez facilement si l'on va voir les définitions *Why3* des trois fonctions. Le principe est toujours d'utiliser des paramètres fantômes (*ghost parameters*) pour faciliter la vérification grâce aux prouveurs SMT. Dans les fonctions `tortoise_hare` et `back_again` on utilise en plus des paramètres fantômes entiers pour relier les différents pointeurs à leur place dans la `list_shape`.

Finalement, ce nouvel algorithme permet d'inverser une liste chaînée, tout comme le fait la première méthode, mais avec deux garanties supplémentaires :

— La liste inversée garde la même structure. La chaîne des pointeurs reste la même (grâce à la double inversion dans `tortoise_hare` et dans `back_again`).

— La liste est inversée en espace constant, aucune allocation mémoire n'est faite, que ce soit sur le tas (*heap*) ou sur la pile (*stack*).

4 Binary Tree Traversal Using Morriss Algorithm

Algorithme de Morris

Cette section présente la vérification de l'algorithme de Morris : le parcours en ordre d'un arbre binaire. En OCaml, la structure d'arbre binaire mutable est définie comme suit :

```
type tree = E
          | N of { mutable left: tree;
                   mutable dat: elt;
                   mutable right: tree }
```

L'algorithme permet d'éviter l'allocation mémoire en ajoutant temporairement des arêtes arrière qui créent des cycles depuis le nud le plus à droite de chaque sous-arbre gauche vers le nud parent. Le code OCaml donné définit une fonction `morris` qui traverse l'arbre selon ce principe, et utilise une fonction auxiliaire `warp`. Celle-ci descend vers le nud le plus à droite du sous-arbre gauche. Si ce nud possède un pointeur droit vide, `warp` crée un lien temporaire vers le parent et renvoie `true`. Si ce nud pointe déjà vers le parent, le lien temporaire est supprimé et `warp` renvoie `false`. La fonction `morris` utilise cette information pour décider sil faut parcourir le sous-arbre gauche ou visiter le nud courant.

Projection sur une séquence plate

Pour vérifier l'algorithme de Morris, nous projetons la structure d'un arbre sur une séquence fantôme plate, dans l'ordre du parcours en ordre. En effet, les outils SMT ne gèrent pas bien la récursion sur les arbres. Pour chaque nud, nous stockons les positions de ses enfants et descendants extrêmes :

```
type tree_shape = {
  cell : int -> tree;   (* noeuds *)
  lchd : int -> int;    (* enfant gauche *)
  rchd : int -> int;    (* enfant droit *)
  lmdt : int -> int;    (* descendant + a gauche *)
  rmdt : int -> int;    (* descendant + a droite *)
  size : int }
```

Seuls les noeuds non-vides sont stockés. Par exemple, pour un nud `i` avec sous-arbre gauche vide, on met `i` dans `lchd` et `lmdt` (idem à droite avec `rchd` et `rmdt`).

Invariants et prédicats

Le type `tree_shape` possède des invariants : cellules non-vides, cellules deux-à-deux distinctes, listées selon le parcours en ordre, sans cellules superflues. Les prédicats `wf_lhs` et `wf_rhs` relient la forme d'arbre au contenu de la mémoire, pour les côtés gauche et droit séparément. Ils sont définis sans récursion, uniquement avec des quantificateurs universels et de l'arithmétique linéaire.

États intermédiaires

Le prédicat `warped_rhs` étend `wf_rhs` pour confirmer aussi que les états intermédiaires en mémoire définissent exactement l'arbre. En effet dans les état intermediaires certains sous-arbres gauches sont déformés et bouclent vers leurs parents. Un nud `i` avec enfant droit vide peut avoir son champ droit pointant vers `cell[i+1]`. Seuls les sous-arbres contenant l'indice `lo` (fixé au nud courant) sont déformés.

Vérification

Pour vérifier l'algorithme on utilise une liste globale `trace` qui enregistre les noeuds visités. On doit prouver donc deux choses : (1) la trace finale contient tous les noeuds dans le bon ordre, et (2) l'arbre final est revenu à son état initial.

Un lemme mathématique est nécessaire : deux sous-arbres quelconques sont soit imbriqués (l'un dans l'autre), soit complètement séparés avec au moins un nud entre eux.

Enfin, une fonction fantôme (utilisée uniquement pour la preuve) transforme une définition récursive classique d'arbre binaire en notre représentation plate `tree_shape`, permettant de partir d'une spécification naturelle vers une forme vérifiable automatiquement.

5 Conclusion

L'approche de "séparation arithmétique" présentée dans cet article permet de transformer le problème de vérification des structures récursives en un problème d'arithmétique linéaire. Les auteurs ont réussis à largement automatiser le processus de preuve avec des solveurs SMT grâce à cette méthode.

Les trois exemples étudiés démontrent la polyvalence de la méthode. L'inversion classique sert à introduire la notion de `list_shape` et les différents prédicats non-récursifs. L'algorithme d'inversion des valeurs en espace constant illustre la modification temporaire des structures (la structure de la liste est au final inchangée). Enfin, l'algorithme de Morris montre que cette méthode est extensible à d'autres structures plus complexes, comme ici les arbres binaires avec la notion de `tree_shape`, prouvant que la technique n'est pas limitée aux structures linéaires.

Cette méthode a cependant des limites. En effet, son application à des structures plus complexes comme les graphes orientés acycliques ou les graphes généraux reste un défi ouvert. De plus, la nécéssité de définir manuellement

les structures fantômes appropriées pour chaque type de données peut être un frein à l'adoption de cette méthode.

Finalement, l'article démontre quand même que pour une grande classe de programmes (au moins les opérations sur les listes chaînées et les arbres binaires), la séparation arithmétique offre un bon compromis entre expréssivité (cf. Introduction) et automatisation par solveurs SMT.