Rapport de projet

Implémenter (en C) et prouver (avec Frama-C) les algorithmes décrits dans les sections 3 et 4 de l'article donné en référence

Prudence Deteix, Paul Passeron, Jean Rousseau

Table des matières

1 Introduction **3**

2 Implémentation en C **4**
2.1 Partie 3 4
2.1.1 La fonction Back-Again 5
2.1.2 La fonction Tortoise-Hare 6
2.1.3 La fonction finale 7
2.2 Partie 4 8

3 Preuve avec Frama-C **10**
3.1 Partie 3 10
3.1.1 Prédicats 10
3.1.2 Contrats de fonctions 12
3.1.3 Lemmes auxilliaires 15
3.2 Partie 4 18

4 Problèmes **20**

5 Pistes d'amélioration **21**

6 Conclusion **22**

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.

2 Implémentation en C

2.1 Partie 3

L'algorithme que l'on va implémenter sur ces listes permet d'inverser ces listes chaînées, sans modifier les pointeurs `cdr`. Au contraire, on inverse ici les valeurs des champs `car`. Cela se passe en deux étapes :

1. **Phase du lièvre et de la tortue (Tortoise-Hare)** : On parcourt la liste en utilisant un pointeur lent et un pointeur rapide afin de trouver le milieu de la liste, tout en inversant les pointeurs `cdr` de la première partie.
2. **Phase Back-Again** : On traverse simultanément les deux moitiées, en échangeant les valeurs (`car`), tout en restorant la structure originelle de la liste au niveau des pointeurs (`cdr`).

Tout d'abord, il faut implémenter le type de liste chaînées défini par JCF tel que :

```
type lst =
    | Nil
    | Cons of {mutable car: elt; mutable cdr: lst}
```

On obtient ainsi le fichier `common.h`, en notant que les champs de structures sont mutables par défaut en C :

```
typedef int elt_t;

struct lst {
    elt_t car;
    struct lst *cdr;
};

typedef struct lst *lst_t;
```

2.1.1 La fonction Back-Again

```
1  void back_again(lst_t bp, lst_t sp, lst_t np)
2  /*@ ghost (list_shape ls, int k) */
3  {
4      if (bp == NULL || np == NULL)
5          return;
6      elt_t tmp = bp->car;
7      bp->car = np->car;
8      np->car = tmp;
9      lst_t nbp = bp->cdr;
10     ls.cells[i]->cdr == ls.cells[i + 1]; */
11     back_again(nbp, bp, np->cdr) /*@ ghost (ls, k-1)
       */;
12 }
```

Les paramètres fantômes `ls` et `k` servent à savoir où l'on est dans la forme de la liste. La fonction échange les valeurs a la position $k-1$ et a la position `ls.count` - k et ensuite restore le pointeur à la position $k-1$.

2.1.2 La fonction Tortoise-Hare

```
1  void tortoise_hare(
2      lst_t bp,
3      lst_t sp,
4      lst_t fp,
5      lst_t qp
6  )
7  /*@ ghost (list_shape ls, int k) */
8  {
9      lst_t nfp;
10     if (fp == qp) {
11         // La taille de la liste est paire, on est
       arrive a la fin
12         back_again(bp, sp, sp) /*@ ghost(ls, k) */;
13     } else {
14         nfp = fp->cdr;
15         if (sp && fp && nfp == qp) {
16             // La taille est impaire, on est a une
       cellule de la fin
17             back_again(bp, sp, sp->cdr) /*@ ghost (
       ls, k) */;
18         } else {
19             nfp = fp->cdr->cdr;
20             lst_t nsp = sp->cdr;
21             sp->cdr = bp; // On inverse le pointeur
22             tortoise_hare(sp, nsp, nfp, qp) /*@
       ghost (ls, k + 1) */;
23         }
24     }
25 }
```

On implémente la première phase avec l'aglorithme du lièvre et de la tortue.

2.1.3 La fonction finale

Pour inverser un segment de liste entre les pointeurs `bp` et `fp`, on crée une dernière fonction :

```
1 void value_reverse(lst_t sp, lst_t qp)
2  /*@ ghost (list_shape ls) */
3 {
4   tortoise_hare(NULL, sp, sp, qp) /*@ ghost (ls, 0)
     */;
5 }
```

2.2 Partie 4

Pour commencer, nous implémentons les arbres binaires à laide dune structure chaînée.

Chaque nud est défini par une donnée (de type int dans notre cas), un pointeur vers son sous-arbre gauche et un pointeur vers son sous-arbre droit.

```
1 struct tree {
2   struct tree *left;
3   struct tree *right;
4   elt_t dat;
5 };
```

L'étape suivante est d'implémenter l'algorithme de Morris.

L'algorithme de Morris sert à traverser les arbres binaires dans l'ordre sans utiliser de pile.

Voici la logique de cet algorithme, ci dessous :

On commence à la racine de l'arbre, puis on répète jusqu'au dernier nud.
Pour chaque nud courant :

Sil na pas de fils gauche :
— → on le visite
— → on va à son fils droit
Sil a un fils gauche :
— → on cherche son prédécesseur : le nud le plus à droite du sous-arbre gauche
Si ce prédécesseur ne pointe pas encore vers le nud courant :
— → on crée un lien temporaire vers le nud courant
— → on va à gauche
Si ce prédécesseur pointe déjà vers le nud courant :
— → on supprime le lien
— → on visite le nud courant
— → on va à droite

Pour implémenter cet algorithme, on crée quatre fonctions.

La fonction `void visit(tree_t t);` sert à visiter chaque nud, en affichant sa valeur.

La fonction `void traversal(tree_t t);` sert de point dentrée et lance le parcours de Morris à partir de la racine de larbre.

La fonction `bool warp(tree_t t, tree_t q);` sert à créer le lien entre le nud le plus à droite du sous-arbre gauche et le nud courant s'il n'existe pas encore (et retourne `true`), sinon elle l'enlève et retourne `false`.

La fonction `void morris_visit(tree_t t);` est la fonction principale de l'algorithme. Elle applique directement la logique de Morris décrite au-dessus en appelant les fonctions `warp` et `visit`.

3 Preuve avec Frama-C

3.1 Partie 3

Forme de la liste La structure importante ici est la suivante, qui projette la liste récursive en une séquence déroulée :

```
typedef struct list_shape_t {
    lst_t *cells; // Tableau des ptrs de cellules
    int count;    // Nombre de cellules
} list_shape;
```

Cette structure est passée en paramètre *ghost* aux fonctions, permettant de raisonner sur les indices plutôt que sur les pointeurs directement.

3.1.1 Prédicats

La vérification repose sur plusieurs prédicats **ACSL** qui permettent de décrire les listes sans récursion.

Validité de la forme de la liste On vérifie que la list_shape est bien formée, surtout du fait que toutes les cellules sont séparées (au sens de la mémoire) deux à deux.

```
/*@ predicate valid_ls(list_shape ls) =
    ls.count >= 0 &&
    (\forall integer i; 0 <= i < ls.count ==> \valid
   (ls.cells[i]) && valid_list(ls.cells[i])) &&
    (\forall integer i; 0 <= i <= ls.count ==>
   valid_list(ls.cells[i])) &&
    (\forall integer i, j; 0 <= i < j <= ls.count
   ==> \separated(ls.cells[i], ls.cells[j])) &&
    (\forall integer i; 0 <= i <= ls.count ==>
        \forall integer j; 0 <= j < length(ls.cells[
   i]) ==>
            (\exists integer n; ls.cells[n] == nth(
   ls.cells[i], j))
    ) &&
    \valid(&ls.cells[0..ls.count]);
*/
```

Les conditions essentielles sont :

— Le compteur est positif.

— Toutes les cellules sont valides (lecture / écriture).
— Toutes les cellules sont séparées deux à deux.
— Le tableau `cells` lui-même est valide.

Segment de liste en avant On décrit un segment de liste de l'indice `lo` à `hi` dans `ls` où tous les `cdr` des cellules pointent vers la prochaine.

```
/*@ predicate listLR (list_shape ls, lst_t p,
   integer lo, integer hi, lst_t q) =
    0 <= lo <= hi <= ls.count &&
    p == ls.cells[lo] &&
    q == ls.cells[hi] &&
    \forall integer i; lo <= i < hi ==> ls.cells[i
   ]->cdr == ls.cells[i + 1];
*/
```

Ce prédicat exprime que le segment [`lo`, `hi`) forme une liste chaînée valide où chaque cellule pointe vers la suivante.

Segment de liste inversé On décrit un segment inversé de la liste où chaque cellule pointe à l'envers (sauf la première, qui pointe vers `NULL`).

```
/*@ predicate listRL (list_shape ls, lst_t p,
   integer k) =
    (k == 0 && p == NULL) ||
    (0 < k <= ls.count && p == ls.cells[k - 1] &&
    ls.cells[0]->cdr == \null &&
    \valid(ls.cells[0]) &&
    \forall integer i; 0 < i < k ==> ls.cells[i]->
   cdr == ls.cells[i - 1] && \valid(ls.cells[i]));
*/
```

Ce prédicat est crucial pour décrire l'état intermédiaire pendant la phase tortoise-hare, où la première moitié de la liste a ses pointeurs inversés.

Prédicat de reversal Ce prédicat exprime que les valeurs `car` ont été échangées symétriquement.

```
/*@ predicate reversal{L1, L2}(list_shape ls,
   integer k) =
    (\forall integer i; 0 <= i < k ==>
        \at(ls.cells[i]->car, L1) == \at(ls.cells[i
   ]->car, L2)) &&
    (\forall integer i; \at(ls.count, L1) - k <= i <
    \at(ls.count, L1) ==>
        \at(ls.cells[i]->car, L1) == \at(ls.cells[i
   ]->car, L2)) &&
    (\forall integer i; k <= i < \at(ls.count, L1) -
    k ==>
        \at(ls.cells[i]->car, L2) == \at(ls.cells[\
   at(ls.count, L1) - 1 - i]->car, L1));
*/
```

Le paramètre k indique la profondeur de la récursion. Les éléments aux indices $[0, k)$ et $[count - k, count)$ sont inchangés, tandis que ceux dans $[k, count - k)$ ont été échangés avec leur symétriques.

Appartenance à la liste Ce prédicat est utile pour vérifier qu'un pointeur appartient bien à la liste.

```
/*@ predicate in_ls(list_shape ls, lst_t l) =
    valid_ls(ls) &&
    valid_list(l) &&
    \exists integer i; 0 <= i <= ls.count ==>
        l == ls.cells[i];
*/
```

3.1.2 Contrats de fonctions

Fonction `value_reverse`

```
1  /*@ requires qp == NULL || \exists int i; 0 <= i <
       ls.count ==> ls.cells[i] == qp;
2    @ requires valid_ls(ls);
3    @ requires valid_list(sp);
4    @ requires valid_list(qp);
5    @ requires listLR(ls, sp, (int)0, (int)ls.count,
       qp);
6    @ assigns ls.cells[0..ls.count]->car \from(ls.
       cells[0..ls.count]->car);
7    @ assigns ls.cells[0..ls.count]->cdr \from(ls.
       cells[0..ls.count]);
8    @ ensures listLR(ls, sp, (int)0, (int)ls.count, qp
       );
9    @ ensures \forall int i; 0 <= i < ls.count ==> ls.
       cells[i]->car == \old(ls.cells[ls.count - 1 - i
       ]->car) && ls.cells[i]->cdr == \old(ls.cells[i]->
       cdr);
10 */
11 void value_reverse(lst_t sp, lst_t qp)
12 /*@ ghost (list_shape ls) */;
```

On assure donc ici que :

— La structure de la liste est préservée (`listLR` reste valide).

— Les valeurs `car` sont inversées.

— Les pointeurs cdr sont bien à leur place initiale.

Fonction `tortoise_hare`

```
1  /*@ requires fp == \null ==> qp == \null;
2    @ requires bp != \null && sp != \null ==> sp != bp
       ;
3    @ requires k >= 0;
4    @ requires valid_ls(ls);
5    @ requires in_ls(ls, bp);
6    @ requires in_ls(ls, sp);
7    @ requires in_ls(ls, fp);
8    @ requires in_ls(ls, qp);
9    @ requires qp == ls.cells[ls.count];
10   @ requires ls.cells[2 * k] == fp;
11   @ requires listLR(ls, sp, k, ls.count, qp);
12   @ requires listRL(ls, bp, k);
13   @ requires 2 * k <= ls.count;
14   @ decreases ls.count - k;
15   @ assigns ls.cells[0..ls.count]->car \from(ls.
       cells[0..ls.count]->car);
16   @ assigns ls.cells[0..ls.count]->cdr \from(ls.
       cells[0..ls.count]);
17   @ ensures listLR(ls, ls.cells[0], (int)0, ls.count
       , qp);
18   @ ensures reversal{Old, Post}(ls, (int)0);
19 */
20 void tortoise_hare(lst_t bp, lst_t sp, lst_t fp,
       lst_t qp)
21 /*@ ghost (list_shape ls, int k) */
22 ;
```

Le lièvre (`fp`) avance deux fois plus vite que la tortue (`sp`). Cela permet de détecter le milieu de la liste. La relation `ls.cells[2 * k] == fp` l'exprime bien.

Fonction `back_again`

```
/*@ requires ((bp == \null || sp == \null) && k >= 0) || k > 0;
  @ requires bp != \null && sp != \null ==> sp != bp;
  @ requires valid_ls(ls);
  @ requires in_ls(ls, sp);
  @ requires in_ls(ls, bp);
  @ requires in_ls(ls, np);
  @ requires listLR(ls, sp, k, ls.count, ls.cells[ls.count]);
  @ requires listRL(ls, bp, k);
  @ requires k <= ls.count - k;
  @ requires ls.cells[ls.count - k] == np;
  @ decreases k;
  @ assigns ls.cells[0..ls.count]->car \from(ls.cells[0..ls.count]->car);
  @ assigns ls.cells[0..ls.count]->cdr \from(ls.cells[0..ls.count]);
  @ ensures listLR(ls, ls.cells[0], (int)0, (int)(ls.count), ls.cells[ls.count]);
  @ ensures reversal{Old, Post}(ls, (int)0);
  @ ensures ls.cells[ls.count] == \old(ls.cells[ls.count]);
*/
void back_again(lst_t bp, lst_t sp, lst_t np)
/*@ ghost (list_shape ls, int k) */
;
```

On assure échanger les valeurs et restaurer les pointeurs.

3.1.3 Lemmes auxilliaires

Plusieurs lemmes ont été nécessaires pour guider le prouveur :

Lemmes sur la séparation

```
/*@ lemma valid_ls_means_all_sep_from_cdr:
    \forall list_shape ls; valid_ls(ls) ==> \forall integer i; 0 <= i < ls.count ==> ls.cells[i]->cdr == \null || \separated(ls.cells[i], ls.cells[i]->cdr);
*/
```

Qui assure que modifier le `cdr` d'une cellule ne la modifiera pas elle même.
D'autres lemmes tels que des lemmes de frame on été utilisés.
Certains de ces lemmes n'ont malheureusement pas pu être prouvés :

— `listLR_length`
— `valid_ls_means_all_sep_from_cdr`
— `in_ls_not_end_means_valid_aux`

Sinon, tous les buts sont prouvés.
Voici la commande utilisée pour lancer frama-c :

```
frama-c -wp -wp-rte -wp-prover alt-ergo,z3,cvc5
   const_space_impl.c -wp-auto wp:split -wp-timeout
   =30
```

Et la sortie finale :

```
[kernel] Parsing const_space_impl.c (with
   preprocessing)
[wp] Running WP plugin...
[rte:annot] annotating function back_again
[rte:annot] annotating function tortoise_hare
[rte:annot] annotating function value_reverse
[wp] Computing [100 goals...]
[wp] 170 goals scheduled
[wp] [Timeout] typed_lemma_listLR_length (Alt-Ergo)
   (Cached)
[wp] [Unknown]
   typed_lemma_valid_ls_means_all_sep_from_cdr (CVC5
   ) (Cached)
[wp] [Timeout]
   typed_lemma_in_ls_not_end_means_valid_aux (Tactic
   ) (Alt-Ergo) (Cached)
[wp] [Cache] found:162, updated:7
[wp] Proved goals:  167 / 170
  Qed:              52 (0.54ms-19ms-239ms)
  Alt-Ergo 2.6.2:   95 (10ms-338ms-6.9s)
  CVC5 1.1.2:        1 (593ms)
  Z3 4.8.12:        16 (51ms-3.7s-21.7s)
  Script:            3 (Tactics 6) (Qed 2/14 425ms)
   (Alt-Ergo 2/14 186ms) (CVC5 2/14 1.6s) (Z3 2/14
   11.9s) (Cached)
  Timeout:           2
  Unknown:           1
```

3.2 Partie 4

Pour prouver plus facilement la correction des fonctions de l'algorithme de Morris, on a commencé par redéfinir un type d'arbre à "plat". On a donc défini le type `tree_shape_t` :

```
typedef struct tree_shape_t {
    tree_t *cell; // tree node
    int *lchd;    // left child
    int *rchd;    // right child
    int *lmdt;    // leftmost descendant
    int *rmdt;    // rightmost descendant
    int size;
} tree_shape_t;
```

On obtient donc un tableau de taille 6 * le nombre de nuds de l'arbre, et pour chaque nud son nud (pour relier à la forme définie avant), son enfant gauche et droit, son enfant le plus à gauche et le plus à droite et enfin la taille de l'arbre.

De plus nous avons défini des variables *ghost* :

- `ghost tree_shape_t morris_t` : arbre "plat"
- `ghost int morris_r` : indice du nud à la racine
- `ghost int morris_c` : nombre de nuds déjà visités
- `ghost int morris_k` : indice du nud courant
- `ghost int warp_j` : indice du nud courant dans warp (recherche du nud le plus à droite du sous-arbre gauche)
- `ghost struct tree *trace[MAX_NODES]` : tableau contenant les nuds déjà visités

Ces variables *ghost* vont nous aider à écrire les contrats de fonction et donc prouver les fonctions.

Après avoir défini ce type, nous avons traduit l'ensemble des prédicats donnés en `Why3` par l'article. Ces prédicats nous serviront ensuite à remplir les contrats de fonction. On trouve donc les prédicats `wf_lst` et `wf_rst` qui garantissent que les sous-arbres gauche et droit : commencent bien juste après i, sont contigus dans le tableau, et correspondent bien aux pointeurs `left` et `right` respectivement. On a aussi le prédicat `warped_rst` qui décrit l'arbre comme valide mais dont les pointeurs droits peuvent être modifiés. Puis le prédicat `frame_tree` qui indique que seul l'arbre qui lui est passé en argument sera modifié. Enfin le prédicat `valid_tree` qui ressemble à `valid_list` et qui confirme que l'arbre est valide (donc le nud courant et ses sous-arbres).

Après cela, nous avons traduit l'ensemble des contrats de fonctions pour

les 4 fonctions (`warp`, `morris_visit`, `traversal`, `visit`). Pour chaque fonction nous avons les `requires`, `assigns`, `ensures` et parfois des variants de boucles : `decreases`.

Une fois les contrats de fonctions réalisés, nous avons réalisé que les buts étaient trop compliqués pour le prouveur. De plus les fonctions étant récursives, les contrats de fonctions doivent être vérifiés à plusieurs reprises. Pour essayer de simplifier ces preuves, on a donc ajouté des `assert` dans les fonctions. Ces `assert` rajoutent des goals plus faciles à démontrer et aident à montrer les `requires` du prochain appel à une fonction.

4 Problèmes

5 Pistes d'amélioration

6 Conclusion