

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	2
2	Implémentation en C	3
2.1	Partie 3	3
2.2	Partie 4	4
3	Preuve avec Frama-C	6
3.1	Partie 3	6
3.2	Partie 4	6
4	Problèmes	8
5	Pistes d'amélioration	9
6	Conclusion	10

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

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

On obtient ainsi le fichier common.h :

```
1 #include <stddef.h>
2
3 typedef int elt_t;
4
5 struct lst {
6     elt_t car;
7     struct lst *cdr;
8 };
9
10 typedef struct lst *lst_t;
```

Dans le fichier lst.h dans lequel on inclut les bibliothèques suivantes :

```
1 #include "common.h"
2 #include <stddef.h>
3 #include <stdbool.h>
4 #include <stdio.h>
```

On va, traduire la fonction blable : CODE CAML

Ce qui donne :

```
1 lst_t aux(lst_t r, lst_t l) {
2     if (!l) {
3         return r;
4     }
5     else {
6         lst_t n = l->cdr;
7         l->cdr = r;
8         return aux(l, n);
9     }
10 }
```

```
1 lst_t list_reversal(lst_t l) {
2     return aux(NULL, l);
3 }
```

2.2 Partie 4

Pour commencer, nous implémentons les arbres binaires à l'aide d'une structure chaînée.

Chaque nœud 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 nœud.

Pour chaque nœud courant :

S'il n'a pas de fils gauche :

— → on le visite

— → on va à son fils droit

S'il a un fils gauche :

— → on cherche son prédécesseur : le nœud le plus à droite du sous-arbre gauche

Si ce prédécesseur ne pointe pas encore vers le nœud courant :

— → on crée un lien temporaire vers le nœud courant

— → on va à gauche

Si ce prédécesseur pointe déjà vers le nœud courant :

— → on supprime le lien

— → on visite le nœud courant

— → on va à droite

Pour implémenter cet algorithme, on crée quatre fonctions.

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

La fonction `void traversal(tree_t t);` sert de point d'entrée et lance le parcours de Morris à partir de la racine de l'arbre.

La fonction `bool warp(tree_t t, tree_t q);` sert à créer le lien entre le nœud le plus à droite du sous-arbre gauche et le nœud 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

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` :

```
1 typedef struct tree_shape_t {
2     tree_t *cell; // tree node
3     int *lchd; // left child
4     int *rchd; // right child
5     int *lmdt; // leftmost descendant
6     int *rmdt; // rightmost descendant
7     int size;
8 } tree_shape_t;
```

On obtient donc un tableau de taille $6 * \text{le nombre de nœuds de l'arbre}$, et pour chaque nœud son nœud (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 nœud à la racine
- *ghost* `int morris_c` : nombre de nœuds déjà visités
- *ghost* `int morris_k` : indice du nœud courant
- *ghost* `int warp_j` : indice du nœud courant dans warp (recherche du nœud le plus à droite du sous-arbre gauche)
- *ghost* `struct tree *trace[MAX_NODES]` : tableau contenant les nœuds 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 prédictats donnés en Why3 par l'article. Ces prédictats nous serviront ensuite à remplir les contrats de fonction. On trouve donc les prédictats `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 prédictat `warped_RST` qui décrit l'arbre comme valide mais dont les pointeurs droits peuvent être modifiés. Puis le prédictat `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 noeud 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