Boucler la boucle du parcours de Morris

Arnaud Golfouse, Paul Patault

Boucler la boucle du parcours de Morris

Arnaud Golfouse[1] et Paul Patault[1]

[1]Université Paris-Saclay, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France

L'algorithme de Morris parcourt un arbre mutable en temps linéaire et en espace constant. L'arbre est modifié au cours de l'itération, mais restauré au fur et à mesure. Nous en proposons une preuve avec Creusot, un outil de vérification déductive de programme Rust.

1 Introduction

Le système de types de Rust est basé sur la notion de *possession*. Le langage est capable d'assurer une utilisation sûre des pointeurs, qu'ils soient mutables ou non, sans avoir de ramasse-miettes. Grâce à son vérificateur d'emprunt, le compilateur peut traquer statiquement si les pointeurs sont partagés, empruntés, pleinement possédés, ou rendus. Ce vérificateur est partie intégrante du typage de Rust et vérifie que la propriété « *partage* XOR *mutation* » est invariante. Cette propriété capture une très grande partie des erreurs faites par les programmeurs. Cependant, cela limite significativement le code qu'il est possible d'écrire. Par exemple, il est impossible de créer toute structure cyclique et mutable.

Pour pallier ces limites, le langage permet aussi une utilisation dite *unsafe* de pointeurs [1]. Dans un bloc annoté, le programmeur a accès à des pointeurs spéciaux (appelé pointeurs bruts) pour lesquels la vérification n'est pas effectuée. Ces blocs sont utiles dans deux cas de figure. Il est possible que la propriété « *partage* XOR *mutation* » soit vraie, mais que le raisonnement y conduisant soit trop complexe pour être mené par le vérificateur d'emprunts. Autrement, la propriété peut être volontairement cassée lorsque le programmeur souhaite écrire une structure impossible à représenter (*e.g.*, listes chainées mutables) : c'est le cas ici.

L'algorithme de Morris [Mor79] est un parcours d'arbre binaire mutable qui s'exécute en temps linéaire et en espace constant. Au cours de l'exécution, certains pointeurs sont modifiés, ce qui a pour conséquence de détruire la structure d'arbre en introduisant des cycles. Dans ce travail, nous proposons une vérification formelle d'une implémentation de ce parcours en Rust. Nous comprenons dès à présent une des difficultés : il s'agit de vérifier formellement un programme nécessairement *unsafe*.

Pour la preuve, nous utilisons Creusot [DJM22], un outil de spécification et vérification déductive pour Rust. Il s'appuie sur Coma [PPF25, PGD25], un langage intermédiaire pour la vérification, lui-même basé sur la plateforme de preuve de programmes Why3 [BFM+11]. Ce dernier offre une interface avec de nombreux prouveurs SMT auxquels sont envoyées les conditions de vérifications calculées. Ainsi, du point de vue de l'utilisateur de Creusot, la vérification est majoritairement automatique.

L'intégralité du code Rust et de la preuve Creusot sont disponibles à l'adresse `https://doi.org/10.5281/zenodo.17914344`.

1. Il est également possible d'utiliser le mécanisme de *mutabilité intérieure*. Ces deux mécanismes étant traités de la même manière par Creusot, nous ne parlerons pas de ce dernier.

2 Algorithme de Morris

L'algorithme de Morris [Mor79] est un parcours *infixe* d'arbre binaire mutable. Il s'exécute en espace constant grâce à des modifications temporaires de l'arbre. Pendant le parcours, certaines feuilles sont modifiées : le nœud le plus à droite d'un sous-arbre gauche peut pointer vers le parent de ce sous-arbre. Cela permet de retrouver à quel endroit il faut continuer l'itération une fois ce sous-arbre traité. Dans cette section, nous omettons les annotations *unsafes* de blocs, car ils le sont tous.

Pour représenter les arbres, nous utilisons le type :

```
pub struct Tree<T> { root: *const Node<T> }
```

où `*const` est le type de pointeur brut et `Node` est le type privé définissant les nœuds de l'arbre.

```
struct Node<T> {
    value: T,
    left:  *const Node<T>,
    right: *const Node<T>,
}
```

Ici, `value` est la valeur portée par un nœud. Si le nœud a un fils gauche (resp. droit), `left` (resp. `right`) est un pointeur vers celui-ci, sinon il est nul.

Le programme que nous vérifions est légèrement différent de l'algorithme de Morris original. Il s'agit d'une variante dans laquelle nous inversons le contrôle. Au lieu d'effectuer l'itération complète d'un coup, nous programmons un générateur. Son état est représenté par le type `TreeIter` et se construit de la manière suivante :

```
pub struct TreeIter<T> {
    curr_ptr: *const Node<T>,
}
pub fn iter<T>(t: Tree<T>) -> TreeIter<T> {
    TreeIter { curr_ptr: t.root }
}
```

La fonction `iter` doit prendre possession complète de l'arbre. En effet, en modifiant certains pointeurs, l'algorithme de Morris casse nécessairement la structure d'arbre au cours de l'itération. Ainsi, si cette dernière est interrompue prématurément, alors l'arbre se retrouverait dans un état incohérent.

Un pas d'itération est effectué avec la fonction `next`, de signature :

```
pub fn next<T>(t: &mut TreeIter<T>) -> Option<&mut T>
```

Sa définition commence par un simple test. Si le pointeur `t.curr_ptr` est nul, le résultat renvoyé est `None`, car le parcours est terminé.

```
    if t.curr_ptr.is_null() {
        return None;
    }
```

La prochaine étape du calcul consiste à déterminer si le sous-arbre gauche a déjà été visité. Le cas échéant, nous renvoyons la valeur de la racine et décalons notre itérateur à droite. Autrement, il faut visiter le sous-arbre gauche.

```
    if left_done {
        let res = &(*t.curr_ptr).value;
        t.curr_ptr = (*t.curr_ptr).right;
        Some(res)
    } else {
        t.curr_ptr = (*t.curr_ptr).left;
        next(t)
    }
```

Pour calculer `left_done`, nous devons chercher `pred_ptr`, le nœud le plus bas à droite du sous-arbre gauche dans l'arbre d'origine. En effet, le champ `pred_ptr.right` est nul si et seulement si nous n'avons pas encore visité le sous-arbre gauche.

S'il est nul, `left_done` est faux, et nous introduisons une *boucle arrière*. Pour ce faire, `pred_ptr.right` est modifié pour pointer vers `t.curr_ptr`. Cela permet de retrouver la racine après la visite du sous-arbre gauche.

Sinon, `pred_ptr.right` (nécessairement égal à `t.curr_ptr`) est une boucle arrière précédemment posée — ce qui indique que le sous-arbre gauche de `t.curr_ptr` est déjà visité. Dans ce cas, la condition `left_done` est vraie.

```
let left_done = (*t.curr_ptr).left.is_null() || {
    let mut pred_ptr = (*self.curr_ptr).left;
    loop {
        if (*pred_ptr).right.is_null() {
            // installation de la boucle
            (*(pred_ptr as *mut Node<T>)).right = self.curr_ptr;
            break false;
        }
        if (*pred_ptr).right == self.curr_ptr {
            // élimination de la boucle
            (*(pred_ptr as *mut Node<T>)).right = std::ptr::null();
            break true;
        }
        pred_ptr = (*pred_ptr).right;
    }
};
```

Ceci conclut la définition du générateur `next`, dont la vérification est l'objet de la section 3.

Pour illustrer l'algorithme, nous présentons en figure 1 un exemple d'exécution. La racine de l'arbre est le nœud de valeur 8. Le pointeur de l'état du générateur (`t.curr_ptr`) est représenté par le triangle rouge. Les nœuds encadrés en vert ont été visités. Les pointeurs `left` et `right` de chaque nœud sont les flèches pleines (omises lorsque le pointeur est nul). Enfin, les *boucles arrière* sont les flèches pointillées. L'étape précédant la première case illustrée est la suivante : `t.curr_ptr` pointe sur le nœud 3, qui n'a pas de fils gauche, nous le visitons avant de passer à son sous-arbre droit — qui se trouve être un parent, mais nous ne le savons pas encore.

- ◧ Le nœud 4 ayant un fils gauche, nous devons installer (ou détecter la présence d') une *boucle arrière*. Celle-ci étant déjà présente, nous concluons que le sous-arbre gauche a été visité. Il faut retirer la boucle, renvoyer 4, et passer au sous-arbre droit pour l'appel suivant.
- ◨ Le nœud 6 ayant un fils gauche, nous devons installer (ou détecter la présence d') une *boucle arrière*. Une fois mise en place, nous visitons le sous-arbre gauche.
- ◧ Le nœud 5 n'a pas de fils gauche, nous le visitons et descendons à droite.
- ◨ Similaire à ◧.

3 Preuve

Nous prouvons la correction de l'algorithme de Morris avec Creusot, un outil de vérification déductive pour Rust.

La spécification principale est la postcondition de la fonction `next` :

```
#[ensures(match result {
    None    => (*t)@ == Seq::empty() && *t == ^t,
    Some(x) => (*t)@ == (^t)@.push_front(*x),
})]
```

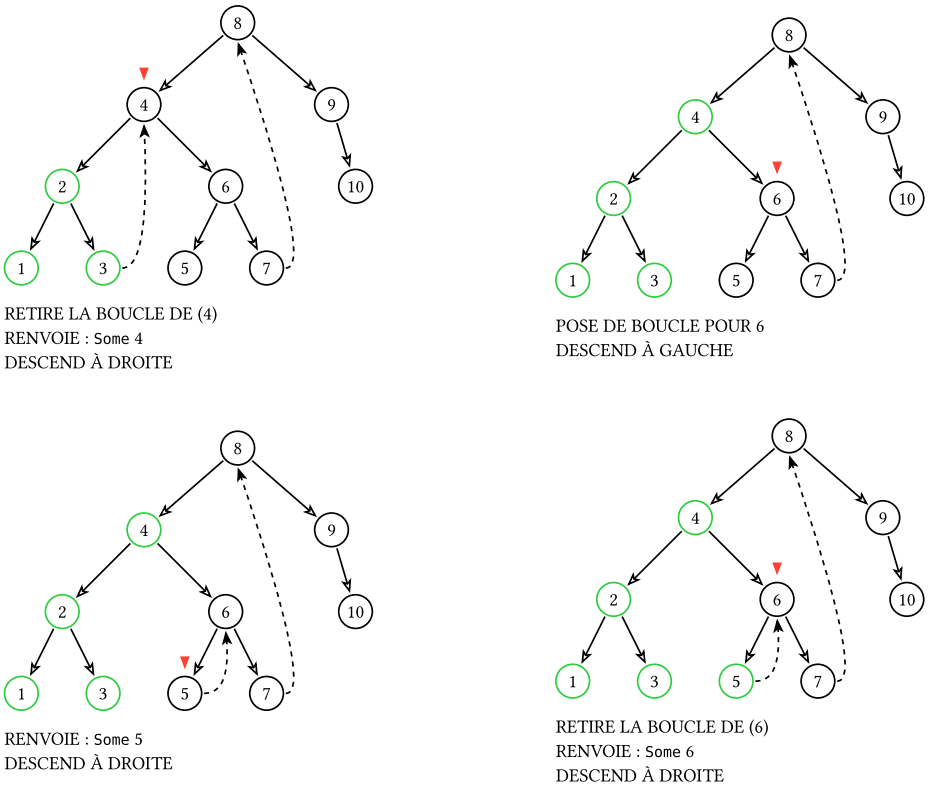

Figure 1. Quatre pas d'étape du parcours.

Cette formule s'écrit dans le langage logique de Creusot. La variable `result` correspond à la valeur de retour de la fonction. L'opérateur postfixe `@` permet de récupérer le modèle logique de son paramètre. Nous définissons, sur le type de `t`, ce modèle logique comme étant la séquence des éléments qu'il reste à visiter dans l'ordre infixe. L'opérateur `*` récupère la valeur de `t` au début de l'exécution de la fonction. Et l'opérateur `^` récupère la valeur de `t` au moment où l'emprunt initial est rendu, après l'exécution de la fonction.

Si la valeur de retour est `None`, le parcours de l'arbre est terminé : la séquence des éléments à parcourir est vide, et `t` n'est pas modifié par la fonction. En revanche, si `result` est de la forme `Some(x)`, alors `x` était le prochain élément à visiter. Ainsi, `x` est en première position dans la séquence à parcourir en entrée, et en sortie, il est enlevé de la séquence.

Cette postcondition ne suffit pas à prouver la correction de l'algorithme, ni même sa sûreté mémoire. En effet, il n'y a pour le moment ni garantie que le pointeur `t.curr_ptr` soit valide, ni qu'il pointe vers une structure bien formée. Pour s'en assurer, nous devons ajouter des champs fantômes à `TreeIter` et un invariant de type.

Gestion des pointeurs. Contrairement au Rust classique, Creusot n'autorise pas directement le déréférencement de pointeurs bruts, et ce, même dans un bloc annoté *unsafe*. En effet, leur caractère librement duplicable casse la propriété « *partage* XOR *mutation* » supposée par Creusot. Pour combler ce manque, Creusot adopte un mécanisme similaire à Verus [LHC+23] : les pointeurs bruts sont gérés avec du code fantôme.

La création d'une valeur de type `*const T` se fait à travers la fonction

```
fn PtrOwn::new(x: T) -> (*const T, Ghost<PtrOwn<T>>)
```

en stockant la donnée `x` sur le tas. La première composante de la paire correspond au pointeur (de programme) vers la donnée. La seconde composante contient, pour Creusot, l'équivalent d'un *points-to* de logique de séparation. Le type `PtrOwn` représente une permission `p`, qui combine le pointeur renvoyé `p.ptr()` et la valeur pointée `p.val()`. De plus, une permission n'étant pas duplicable, la propriété « *partage* XOR *mutation* » est de nouveau satisfaite ; ce qui permet à Creusot de suivre l'évolution de sa valeur. Remarquons enfin que la seconde composante est enveloppée dans le type `Ghost`. Ceci a pour effet de l'effacer, d'un point de vue opérationnel, à la compilation ; tout en conservant son caractère non duplicable.

Pour déréférencer le pointeur ainsi créé, Creusot fournit les fonctions `PtrOwn::as_ref` et `PtrOwn::as_mut`. Leurs signatures sont

```
fn PtrOwn::as_ref(ptr: *const T, perm: Ghost<&PtrOwn<T>>) -> &T
fn PtrOwn::as_mut(ptr: *const T, perm: Ghost<&mut PtrOwn<T>>) -> &mut T
```

Du point de vue opérationnel, ces fonctions rendent tel quel leur premier argument (`ptr`). Tandis que, du point de vue de la vérification, la valeur renvoyée est celle provenant du second (`perm.val()`) : les références étant considérées comme des valeurs par Creusot. Ajoutons à ceci que les fonctions ont pour précondition l'égalité entre `ptr` et `perm.ptr()`.

Dans notre cas, nous ajoutons une séquence fantôme `permissions` à `TreeIter`. Ce champ, de type `Ghost<Seq<PtrOwn<Node<T>>>>`, stocke les permissions associées aux nœuds de l'arbre. Cette séquence indique l'ordre d'itération des nœuds : l'ordre infixe. L'invariant de `TreeIter` ci-dessous manipule des indices dans cette séquence.

Instantanés. Le mécanisme d'instantanés de Creusot permet de capturer un état du programme à partir d'une expression logique via la macro `snapshot!`. La donnée renvoyée est enveloppée dans le type `Snapshot`. De la même manière que les données `Ghost`, ces données sont effacées du point de vue opérationnel. Cependant, contrairement au type `Ghost`, le type `Snapshot` n'induit pas de contrainte de possession : il est librement duplicable.

Dans notre cas, les instantanés sont utilisés pour garder la trace de propriétés purement logiques. La structure logique de l'arbre d'origine est gardée comme `Snapshot` dans `TreeIter`.

Invariant de type. Nous définissons un invariant de type sur `TreeIter`, qui sera automatiquement ajouté au sein des préconditions et postconditions de la fonction `next`. Pour exprimer cet invariant, nous ajoutons à `TreeIter` des champs `Snapshot` décrivant la structure de données sous-jacente :

— `visited`, l'indice du prochain élément à visiter dans la séquence des permissions ;
— `curr`, l'indice de la permission correspondant à `curr_ptr` ;
— `left`, `right`, `rightmost` et `leftmost`, quatre tableaux d'entiers ;
— `loopbacks`, un tableau de booléens représentant la présence des boucles arrière.

L'invariant de type peut se découper en quatre parties :

1. le tableau `left` (resp. `right`) contient à l'indice `i` l'indice du fils gauche (resp. droit) du nœud `i` s'il existe, et `i` sinon. Le tableau `leftmost` (resp. `rightmost`) contient à l'indice `i` l'indice du plus petit fils le plus à gauche (resp. à droite) du nœud `i` s'il existe, et `i` sinon. Voici les valeurs de ces tableaux pour l'arbre représenté en figure 1 :

`left` :	1	1	3	2	5	5	7	4	9	10
`right` :	1	3	3	6	5	7	7	9	10	10
`leftmost` :	1	1	3	1	5	5	7	1	9	10
`rightmost` :	1	3	3	7	5	7	7	10	10	10

2. la mémoire doit être cohérente : si le nœud à l'indice `i` a un fils gauche, le champ `left` de la valeur de la permission à l'indice `i` doit être égal au pointeur de la permission à l'indice `left[i]` ; s'il n'a pas de fils gauche, ce champ doit être nul. La situation est légèrement différente pour le fils droit, à cause des boucles arrière. Si le nœud `i` n'a pas de fils droit, mais que `loopbacks[i]` est vrai, le champ `right` de la valeur de la permission à l'indice `i` doit être égal au pointeur de la permission à l'indice `i + 1`.

3. les boucles arrière doivent être caractérisées. Un nœud ne peut avoir une boucle arrière que s'il n'a pas de fils droit dans l'arbre. Ensuite, il y a deux types de boucles arrière :
   — celles posées lors du parcours vers le nœud courant. Elles ont un indice `i` supérieur ou égal à `curr`, et la cible de la boucle (`i + 1`) doit être un parent de `curr`. Nous exprimons cela par `leftmost[i + 1] <= curr`.
   — celle dans le sous-arbre gauche de `curr` lorsque celui-ci est complètement visité.
4. Enfin, il y a quelques conditions auxiliaires. Les indices `visited` et `curr` doivent être dans les bornes de `permissions` ; `visited` est égal soit à `curr` (si l'on vient de visiter le sous-arbre gauche), soit à `leftmost[curr]` (si l'on s'apprête à le visiter) ; le champ `curr_ptr` doit bien être le pointeur de la permission à l'indice `curr` ; sauf si ce champ est nul, auquel cas `visited` doit être égal à la longueur de `permissions`.

Initialisation des invariants de type. D'une part, l'arbre initial doit contenir les champs `root`, `root_idx`, `l`, `r`, `leftmost`, `rightmost` et la séquence `permissions`. Ainsi, l'invariant de type de l'arbre est le sous-ensemble de l'invariant de l'itérateur ne mentionnant que ces champs.

D'autre part, le champ `visited` de l'itérateur est initialisé à `0`, chacune des cases de `loopback` à `false`. Enfin, l'itérateur hérite de toutes les propriétés communes avec l'arbre bien formé sur lequel il s'appuie.

```
TreeIter {
    curr_ptr: self.root,              // provient de l'arbre
    permissions: self.permissions,    // idem
    visited: snapshot!(0),            // initialisation des nouveaux champs
    loopback: snapshot!(|_| false)    // idem
    ...
}
```

Remarquons qu'il est inévitable que l'itérateur consomme l'arbre initial car son utilisation (par la fonction `next`) brise l'invariant de l'arbre.

Invariants de boucle et axiome fantôme. La fonction `next` est moralement composée de deux boucles. La boucle extérieure est la plus simple. Il s'agit de la fonction elle-même, lorsque le sous arbre gauche n'est pas visité et qu'il faut faire un appel récursif.

Le calcul fait par la boucle interne, en revanche, est plus complexe. Celle-ci permet de trouver le pointeur sur lequel il faut installer (ou retirer) la boucle arrière. Nous décrivons dans ce paragraphe les invariants de boucle nécessaires à cette tâche.

Initialement, le pointeur `pred_ptr` est le fils gauche du nœud courant `curr_ptr`. Pour accéder à sa valeur — requise pour être comparée dans la condition de boucle — nous devons en avoir la permission. Cette dernière peut être récupérée grâce au tableau fantôme que nous avons ajouté (`t.permissions[t.left[t.curr]]`). Ensuite, au cours de l'itération, le pointeur `pred_ptr` suit son fils droit tant que possible et la variable `pred` contient l'indice fantôme de la permission correspondante (initialement `t.left[t.curr]`).

Les deux premiers invariants de la boucle assurent que l'indice `pred` corresponde à l'indice de la permission de `pred_ptr` :

```
#[invariant(0 <= pred && pred < t.permissions.len())]
#[invariant(t.permissions[pred].ptr() == pred_ptr)]
```

Nous pouvons donc accéder cette permission, de manière à pouvoir suivre le fils droit de `pred_ptr` :

```
let right = PtrOwn::as_ref(pred_ptr, t.permissions[pred].borrow()).right;
```

Le troisième invariant est le plus important. Il indique la cible de la boucle, le nœud ayant l'indice `t.curr - 1` :

```
#[invariant(t.rightmost[pred] == t.curr - 1)]
```

Il y a deux façon de sortir de la boucle. Dans les deux cas, il est demandé de prouver l'égalité `pred == t.curr - 1`. Nous procédons par analyse de cas, selon la façon dont nous sortons de la boucle. La première, lorsque l'on arrive en bas du sous arbre : `right.is_null()`. La seconde, lorsque l'on revient par la boucle arrière : `right == t.curr_ptr`.

Dans le premier cas, l'invariant de type nous garantit que `t.right[pred] == pred`, et l'égalité `pred == t.curr - 1` en est une conséquence simple. En revanche, le second cas est moins évident. Lorsque `right` est égal à `t.curr_ptr`, il y a une boucle arrière. Mais pour en être certain, il y a un cas que nous devons exclure en vérifiant que `pred_ptr` n'a pas de fils droit égal à `t.curr_ptr` *dans l'arbre d'origine*. C'est-à-dire, que la permission située à l'indice `t.right[pred]` n'est pas associée au pointeur `t.curr_ptr`.

Cette déduction requiert l'utilisation d'un *axiome fantôme* de la bibliothèque standard de Creusot :

```
#[ensures(own1.ptr().addr_logic() != own2.ptr().addr_logic())]
#[ensures(*own1 == ^own1)]
pub fn disjoint_lemma<T>(own1: &mut PtrOwn<T>, own2: &PtrOwn<T>) {}
```

où `addr_logic()` convertit un pointeur en un entier. Cet axiome est admis dans Creusot car chaque pointeur renvoyé par `PtrOwn::new` est unique. Remarquons aussi que le typage de Rust garantit que `own1` et `own2` pointent vers des objets disjoints.

Dans notre cas, l'utilisation de cet axiome nous permet de savoir que toutes les permissions contenues dans la séquence `t.permissions` sont, nécessairement, associées à des pointeurs différents. L'inégalité `t.right[pred]` $\leqslant$ `t.curr - 1` étant vraie, les permissions associées au fils droit de `pred_ptr` et à `t.curr_ptr` sont disjointes. Ainsi, nous pouvons déduire que `pred == t.curr - 1`.

Ceci conclut la preuve des invariants de la boucle interne. Celle-ci est le point critique de la preuve du parcours de Morris. La spécification principale (postcondition de `next`) est relativement simple à établir, en aidant les prouveurs SMT avec des assertions fantômes bien choisies.

4 Discussion

Limitations. Dans ce paragraphe, nous détaillons des améliorations possibles de notre implémentation de l'algorithme de Morris. Nous n'anticipons pas de difficultés fondamentales à la réalisation de ces idées. D'abord, il est possible de faire renvoyer par `next` la possession complète de la donnée (`T`) plutôt qu'un emprunt mutable (`&mut T`). Cela nécessite de modifier la séquence des permissions au fil de l'itération en retirant un élément à chaque étape. Ce changement complexifie la gestion des indices dans l'invariant mais semble faisable. Autrement, nous pouvons reconstruire l'arbre dans son état initial, à condition de ne pas faire renvoyer par `next` la possession des données. Cette reconstruction n'est possible qu'après avoir exécuté entièrement l'itération (`t.completed()`) :

```
#[requires(t.completed())]
fn reconstruct<T>(t: TreeIter<T>) -> Tree<T>
```

Travaux connexes. Filliâtre et Paskevich [FPD25] ont vérifié l'algorithme de Morris avec Why3. Cependant, cet outil ne permet pas de définir le type des arbres récursifs et mutables ce qui les contraints à utiliser un modèle mémoire explicite pour programmer l'algorithme. Par opposition, le code fantôme de Creusot nous permet d'utiliser naturellement le type `Node` des nœuds. Ainsi, le programme vérifié est directement exécutable.

La méthode appliquée ici est adaptée du celle développée autour de l'outil Verus [LHC+23]. La vérification de l'algorithme de Morris semble réalisable d'une manière proche de la notre. Cependant, le support des emprunts mutables dans Verus est plus restrictif que celui de

Creusot. Cela rend impossible l'utilisation de `Option<&mut T>` comme la même type de retour pour la fonction `next` en les restreignant à un simple emprunt partagé.

Enfin, l'utilisation du code fantôme a permis de vérifier d'autres algorithmes de graphe, tels qu'une bibliothèque de tableaux persistants et la structure *union-find*. Ces exemples sont décrits par Golfouse, Guéneau et Jourdan [GGJ26] dans le travail introduisant le code fantôme avec possession dans Creusot.

5 Conclusion

Nous avons présenté une implémentation Rust de l'algorithme de Morris. Nous prouvons avec l'outil Creusot que la séquence générée est bien un parcours infixe des éléments de l'arbre initial et la sûreté de l'exécution.

Remerciements. Cette recherche a été [partiellement] soutenue par les projets Gospel et Décysif, financés par la région Île-de-France et par le gouvernement français dans le cadre du programme « Plan France 2030 ».

Références

[BFM+11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond et Andrei Paskevich : *The Why3 platform*, février 2011. `https://www.why3.org/`.

[DJM22] Xavier Denis, Jacques-Henri Jourdan et Claude Marché : Creusot : a foundry for the deductive verification of Rust programs. *In International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022. `https://hal.science/hal-03737878`.

[FPD25] Jean-Christophe Filliâtre, Andrei Paskevich et Olivier Danvy : When Separation Arithmetic is Enough. *In iFM 2025 - 20th International Conference on Integrated Formal Methods*, Paris, France, novembre 2025. Caterina Urban, Inria & ENS | PSL, France.

[GGJ26] Arnaud Golfouse, Armaël Guéneau et Jacques-Henri Jourdan : Using Ghost Ownership to Verify Union-Find and Persistent Arrays in Rust. *In Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France*, Rennes, France, janvier 2026. SIGPLAN.

[LHC+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno et Chris Hawblitzel : Verus : Verifying rust programs using linear ghost types. *Software Artifact (virtual machine, pre-built distributions) for "Verus : Verifying Rust Programs using Linear Ghost Types"*, 7(OOPSLA1):85 :286–85 :315, avril 2023.

[Mor79] Joseph M Morris : Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979.

[PGD25] Paul Patault, Arnaud Golfouse et Xavier Denis : Remonter les barrières pour ouvrir une clôture. *In JFLA 2025 - 36es Journées Francophones des Langages Applicatifs*, Roiffé, France, janvier 2025. `https://hal.science/hal-04859517`.

[PPF25] Andrei Paskevich, Paul Patault et Jean-Christophe Filliâtre : Coma, an intermediate verification language with explicit abstraction barriers. `https://hal.science/hal-04839768`, 2025.