Remonter les barrières pour ouvrir une clôture

Inférence de spécification des clôtures pour la preuve de programmes Rust avec Coma

Paul Patault[1], Arnaud Golfouse[2] et Xavier Denis[3]

[1,2]Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, LMF, 91190 Gif-sur-Yvette, France
[3]ETH Zurich, Switzerland

Dans de nombreux programmes, les clôtures permettent d'exprimer de manière concise des transformations de données. Mais lorsqu'un outil de vérification est utilisé, elles doivent être accompagnées de spécifications souvent plus longues que leur corps. C'est un problème particulièrement désagréable, car ces clôtures sont fréquemment simples et leur spécification est redondante.

Dans ce travail, nous présentons un mécanisme d'inférence de spécification des clôtures pour la vérification formelle de programmes Rust. Nous proposons l'utilisation du langage intermédiaire de vérification Coma comme backend par l'outil de vérification déductive Creusot. Notre conception est capable de gérer l'état mutable interne d'une clôture et d'inférer sa spécification. Nous utilisons ce mécanisme pour vérifier de manière ergonomique et modulaire une série de programmes Rust utilisant des fonctions d'ordre supérieur.

1 Introduction

Le langage de programmation Rust a révolutionné le domaine de la programmation système au cours de la dernière décennie grâce à son adoption de notions issue de la théorie des langages de programmation. Connu pour son système de types basé sur la notion de *possession*, Rust est capable d'assurer l'utilisation sûre des pointeurs mutables (et immuables) dans un langage sans ramasse-miettes. De plus, Rust intègre des constructions habituellement associées à des langages fonctionnels telles que les types algébriques ou les clôtures. Bien qu'atypiques dans un langage de programmation système, les clôtures de Rust sont fréquemment utilisées. En particulier, elles se combinent naturellement à des itérateurs.

Denis *et al.* [DJ23] ont montré comment vérifier la correction de l'itérateur `map` avec l'outil Creusot. La technique proposée fonctionne lorsque cet itérateur est appliqué à des fonctions partielles avec effets de bord. Cependant, Creusot est un vérificateur modulaire au sens classique, traitant comme opaque toute clôture passée en argument. La seule information connue sur celle-ci provient donc du contrat fourni par l'utilisateur :

```
map(i, #[requires(x < u32::MAX)] #[ensures(result == x + 1u32)] |x| x + 1)
```

Nous perdons ici les intérêts principaux de l'utilisation d'itérateurs : la clarté et la concision. Même dans un exemple aussi simple que l'incrément, le contrat est bien plus long que la clôture elle-même. L'attrait des itérateurs est donc perdu par rapport aux boucles ordinaires.

L'expansion des clôtures à leur point d'appel n'est pas une solution à ce problème. En effet, l'itérateur `map` (voir figure 1) est spécifié par rapport au contrat de la clôture attendue.

Il nécessite donc des symboles lui permettant de nommer les pré- et postconditions de cette clôture.

L'existence de ces symboles nous permet de raisonner abstraitement sur les propriétés que les instanciations individuelles de `map` doivent satisfaire. Lorsqu'une clôture reçoit un contrat explicite, ces symboles sont facilement générés par Creusot. En revanche, l'expansion de la clôture au point d'appel nous fait perdre la définition de ces symboles.

Le défi central de notre travail est donc : comment pouvons-nous préserver l'ergonomie des clôtures pendant la vérification de programmes Rust ?

Pour résoudre ce problème, nous utilisons Coma [PPF25], un nouveau générateur de conditions de vérification, en l'étendant avec la capacité de *réifier* les préconditions et postconditions des définitions comme symboles indépendants. Nous utilisons ce nouveau mécanisme pour étendre la traduction de Creusot en éliminant les spécifications utilisateur des clôtures et démontrons son fonctionnement sur des exemples techniques (incluant des clôtures avec état mutable, effets de bord et fonctions partielles).

Le reste de l'article est structuré comme suit. Dans la section 2, nous donnons un aperçu rapide de la traduction de Rust par Creusot, en particulier sa gestion des programmes d'ordre supérieur. Dans la section 3, nous présentons rapidement le langage Coma et son générateur de conditions de vérification. Nous définissons le nouveau mécanisme de « d'extraction de spécification » qui produit les pré- et postconditions d'une définition. Nous montrons ensuite en section 4 comment utiliser ce mécanisme pour inférer les spécifications manquantes dans les clôtures, permettant une vérification sans intervention de l'utilisateur. Dans la section 5, nous évaluons notre approche sur une série de benchmarks en montrant que celle-ci supprime la surcharge utilisateur introduite par l'usage traditionnel des clôtures. Enfin, nous laissons nos remarques finales dans la section 6.

2 Creusot

Creusot [DJM22, DJ23] est un outil de spécification et vérification déductive pour Rust. Il s'appuie sur Coma, un nouveau langage d'entrée pour la plateforme de preuve de programmes Why3 [FP13]. Le travail de Creusot est de traduire un programme impératif écrit et annoté en Rust, vers le langage fonctionnel Coma. Chaque fonction Rust du fichier source est traduite vers un module de Coma indépendant. Ensuite, les conditions de vérification (VC) du programme Coma sont calculées et envoyées à des prouveurs automatiques tels que Z3 [DMB08] et Alt-Ergo [CCIM18].

En Rust, les clôtures sont des fonctions anonymes qui peuvent capturer leur environnement. Elles utilisent la syntaxe `|x: i32, y: i32| x + y`, les paramètres sont définis entre les barres verticales (ici `x: i32` et `y: i32`) et sont suivis du corps de la clôture (ici `x + y`). De manière systématique et pour des raisons d'efficacité, les clôtures de Rust sont monomorphisées : si une fonction attend une clôture en argument, *chaque appel* de cette fonction génère un objet différent pour lequel la clôture appelée est connue.

Le programme Rust en figure 1 nous permet d'illustrer le traitement des clôtures par Creusot. Ce fragment de programme contient trois fonctions à vérifier, de manière indépendante. La première, `map`, transforme le contenu de l'itérateur `i` en le passant à la clôture `f`. La deuxième, `iter_add`, est le code client qui appelle `map`. Enfin, la troisième fonction est associée à la clôture `|x: i32| x + y`. Elle est générée par le compilateur qui s'occupe aussi de capturer et de lui passer l'environnement (ici seulement la variable `y`) comme un argument supplémentaire. Les spécifications sont écrites dans des attributs `#[...]` ignorés par le compilateur, mais utilisés par Creusot pour générer le code Coma.

D'une part, la vérification de la définition de `map` est basée sur deux prédicats abstraits `f.precondition` et `f.postcondition` qui axiomatisent le comportement de la clôture. La spécification de la fonction `map` est simplifiée pour faciliter la compréhension. En particulier, elle utilise le prédicat `i.emits(x)`, qui affirme que l'itérateur `i` peut produire l'élément `x`.

```
#[requires(forall<x> i.emits(x) ==> f.precondition(x))]
#[ensures(forall<x, r> i.emits(x) ==> result.emits(r) ==> f.postcondition(x, r))]
fn map<I: Iterator, F: Fn(i32) -> i32>(i: I, f: F) -> Map<I, F> { ... }

fn iter_add(i: I, y: i32) -> i32 {
    map(i,
        #[requires(i32::MIN <= x + y && x + y <= i32::MAX)]
        #[ensures(result == x + y)]
        |x: i32| x + y
    ).sum()
}
```

Figure 1. Utilisation des clôtures avec Creusot

D'autre part, la vérification de `iter_add` (dont la spécification est omise) demande de vérifier un appel à `map`. Cette vérification repose sur la monomorphisation de la clôture : Creusot sait précisément sur quel `f` la fonction `map` est appelée et peut donc instancier les prédicats de pré- et postcondition. Ainsi, au moment de prouver `iter_add`, le code Coma utilise une version spécialisée de `map` dans laquelle les prédicats `f.precondition` et `f.postcondition` sont remplacés par `i32::MIN <= x + y && x + y <= i32::MAX` et `result == x + y`.

Cependant, nous pouvons remarquer que la précondition (resp. postcondition) de la clôture est redondante avec la précondition (resp. postcondition) implicite de l'addition `x + y`. Ainsi, la spécification que l'on donne à la clôture n'a pas vraiment lieu d'être. De plus, cette fonction n'étant utilisée qu'une seule fois (lors de l'appel à `map`), la vérifier de manière indépendante perd de son intérêt.

Dans un monde idéal (cf. section 4), nous ne voulons pas écrire la spécification de cette clôture. En revanche, nous voulons garder la vérification modulaire des fonctions `map` et `iter_add` avec les prédicats `f.precondition` et `f.postcondition`.

3 Coma

Le langage Coma [PPF25] (abréviation pour *Continuation Machine*) est un langage intermédiaire pour la vérification de programmes. De la même manière que Boogie [BCD+06] ou Viper [MSS16], il peut servir de représentation intermédiaire pour des outils de preuves de programmes de multiples langages. Il est implémenté comme un nouveau langage d'entrée du système Why3. Cela permet de profiter pleinement de son backend et en particulier de l'interface avec divers prouveurs automatiques.

Paskevich *et al.* définissent formellement le système de types, la sémantique opérationnelle et le calcul de conditions de vérification de Coma. Plus de détails sur l'implémentation se trouvent dans la documentation [PP24] du langage. Cependant, pour aider à la compréhension du lecteur, nous détaillons dans la partie suivante la syntaxe utilisée.

Syntaxe. Les constructions de base sont les *expressions* qui représentent les calculs. Elles peuvent être encapsulées dans des *handlers*[1] nommés ou anonymes, qui peuvent être appelés ou passés en paramètre de continuation. À l'inverse, les *termes* correspondent aux données pures qui peuvent être stockés dans des variables (notées x et y) et utilisés dans les assertions. Ils comprennent des constantes, des variables et des opérations pures et totales. Nous omettons leurs définitions et celles de leurs types et prenons celles usuelles *à la* ML.

La spécification est basée sur la logique du premier ordre. Les formules notées φ peuvent contenir des termes et des variables, mais pas des expressions.

1. Nom donné aux « fonctions » de Coma.

signature de type :	π	$::=$	$(\alpha)\star\ \ (x : \tau)\star\ \ (h : \pi)\star$	liste de paramètres
expression :	e	$::=$	h	handler
		$\mid$	π `→` e	handler anonyme
		$\mid$	$e\ \ \tau\star\ \ s\star\ \ e\star$	application
		$\mid$	e `/` $h\ \ \pi$ `=` e	définition locale
		$\mid$	`{` φ `}` e	assertion
		$\mid$	`↑` e	barrière opaque

Figure 2. Syntaxe de Coma.

La syntaxe des expressions et de leurs types est détaillée en figure 2. La signature de type d'une expression (notée π) est la liste des paramètres attendus par celle-ci. Un handler peut prendre en argument des variables de types, des termes et des handlers (qui sont les continuations). Comme le langage Coma est en style par passage de continuation, les handlers n'ont pas de valeur de retour. Notons enfin qu'une définition de handler à haut niveau est introduite avec le mot clé `let`.

À des fins de simplicité, nous supposons avoir une bibliothèque standard minimale comprenant le handler `fail`, qui arrête le programme avec la précondition $\bot$ (équivalent de `assert false`) et le handler `unList`, qui correspond à `match-with` sur une liste. Ce dernier prend en paramètre une liste `l` et deux continuations : si `l = Cons h t`, le contrôle est passé à la première continuation qui reçoit `h` et `t` en argument ; sinon `l` est vide, le contrôle est passé à la seconde continuation.

Prenons comme exemple le handler `tail` défini ci-dessous. Il attend une liste d'entiers `l` et appelle sa continuation `return` avec la suite de cette liste, ou échoue si elle est vide.

```
1  let tail (l: list int) (return (r: list int))
2  = { l ≠ Nil }
3      ↑ unList l ((h: int) (t: list int) → break t) fail
4    / break (tl: list int) = { exists h. l = Cons h tl } ↑ return tl
```

L'implémentation de ce handler commence avec l'assertion `{ l ≠ Nil }` qui garde l'expression de la ligne 3. Cette expression est également placée sous une *barrière opaque* (opérateur `↑`) qui n'affecte pas l'exécution, mais uniquement le calcul des conditions de vérification. La barrière cache l'expression qui n'est vérifiée qu'une seule fois, pour toutes valeurs de paramètres. Le symbole *slash* en ligne 4 sépare l'expression principale de la définition locale de `break` et peut être interprété comme le mot « où ».

Remarquons que le handler `break` enveloppe l'appel à la continuation `return` en insérant l'assertion `{ exists h. l = Cons h tl }` et une barrière opaque. Cette définition n'est pas cachée sous la barrière de la ligne 3. Nous avons ajouté une précondition à la continuation, ou autrement dit, une postcondition au handler `tail`.

Calcul des conditions de vérification. Les conditions de vérification (VC) sont les propriétés qu'une expression doit satisfaire pour assurer sa correction fonctionnelle. Elles sont calculées différemment selon le mode courant. Nous distinguons deux modes :

$\mathcal{D}$: *mode appelé* détermine la correction d'une définition de handler ;

$\mathcal{A}$: *mode appelant* détermine la correction d'un appel à un handler.

Par opposition aux méthodes traditionnelles, le changement de mode n'est pas automatiquement fait au niveau des fonctions. Coma propose un mécanisme de barrière d'abstraction *flexible* qui permet de passer intentionnellement d'un mode à un autre. En particulier,

$$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e) \qquad \mathcal{A}(\uparrow e) \triangleq \top$$

L'opérateur de *barrière opaque*, noté $\uparrow$, sépare explicitement le code faisant partie de l'*interface* et le code tenant purement de l'*implémentation*. Le *mode appelé* suppose les préconditions et vérifie la correction de la partie implémentation sous la barrière opaque. À l'inverse, le *mode appelant* extrait la spécification d'un handler en traitant toute assertion comme une précondition à vérifier au point d'appel mais s'arrête au niveau de la barrière [2].

$$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e) \qquad \mathcal{A}(\{\varphi\}\, e) \triangleq \varphi \wedge \mathcal{A}(e)$$

La correction totale d'une expression est assurée par la preuve de la formule obtenue par le *mode total* qui vérifie les assertions avant et après la barrière opaque, c'est la conjonction de $\mathcal{A}$ et $\mathcal{D}$.

Nous définissons deux fonctions $\hat{\mathcal{A}}$ et $\hat{\mathcal{D}}$, générant chacune un prédicat à partir d'un handler h monomorphe et d'ordre un (ses paramètres de continuations n'ont pas de paramètres de continuations) [3]. Ces fonctions préfixent le calcul de VC appliqué au code du handler (noté h_{body}) en introduisant ses paramètres, avec des lambdas pour le premier cas et des quantificateurs universels pour le second. Les paramètres de termes sont inchangés et les paramètres de continuations sont transformés en prédicats (qui représentent leur VC) avec l'opérateur $\lceil\cdot\rceil$ défini ci-dessous.

$$\hat{\mathcal{A}}(h) = \lambda(x:\tau)\star.\, \lambda(g:\lceil\pi\rceil)\star.\, \mathcal{A}(h_{body})$$
$$\hat{\mathcal{D}}(h) = \forall(x:\tau)\star.\, \forall(g:\lceil\pi\rceil)\star.\, \mathcal{D}(h_{body})$$

$$\lceil(y_1:\tau_1)\cdots(y_t:\tau_t)\rceil = (y_1:\tau_1) \to \cdots \to (y_t:\tau_t) \to \mathrm{Prop}$$

Nous pouvons appliquer ces fonctions à notre exemple `tail`. Les deux auront deux arguments : la liste d'entrée ℓ et la VC de la continuation *ret*. Mais dans le premier cas il faut prouver la précondition *sur des arguments concrets* (qui doivent être fournis). En retour, nous gagnons la postcondition comme hypothèse pour la preuve de la VC de la suite du programme.

$$\begin{aligned}\hat{\mathcal{A}}(\texttt{tail}) = \lambda\ell:\texttt{list int}.\, \lambda ret:\texttt{list int} \to \mathrm{Prop}.\\ \ell \neq \texttt{Nil} \wedge (\forall t\ell.\, (\exists h.\, \ell = \texttt{Cons}\ h\ t\ell) \to ret\ t\ell)\end{aligned}$$

À l'inverse, le mode appelé suppose la précondition et demande la preuve du corps du handler `tail` *pour toutes valeurs de paramètres*.

$$\begin{aligned}\hat{\mathcal{D}}(\texttt{tail}) = \forall\ell:\texttt{list int}.\, \forall ret:\texttt{list int} \to \mathrm{Prop}.\\ \ell \neq \texttt{Nil} \to\\ (\forall h.\, \forall t\ell.\, \ell = \texttt{Cons}\ h\ t\ell \to (\exists h'.\, \ell = \texttt{Cons}\ h'\ t\ell)) \wedge\\ (\ell = \texttt{Nil} \to \bot)\end{aligned}$$

Ainsi, il faut montrer la précondition de `break` (c'est-à-dire la postcondition de `tail`) si la première continuation de `unList` est appelée, et la précondition de `fail` (c'est-à-dire $\bot$) sinon. Remarquons que le prédicat *ret* n'apparait pas dans cette formule : cela vient du fait que l'appel à cette continuation est caché sous la barrière opaque.

Il est important de constater que la formule $\hat{\mathcal{D}}(\texttt{tail})$ est une trivialité. Cela vient du fait que la spécification répète précisément ce que calcule le programme : la spécification est *redondante*. Ce problème apparait souvent sur des fonctions courtes, en particulier lorsque l'on doit donner une spécification à une clôture. Or Coma nous permet d'omettre complètement l'ajout d'une barrière dans un handler non récursif. Nous pouvons donc définir le handler `tail` sans barrière de la manière suivante

2. Le reste des règles est omis, voir [PPF25] pour le détail.
3. Ces deux contraintes sont nécessaires pour que la transformation du type des continuations en prédicats WhyML soit possible.

```
let tail (l: list int) (return (r: list int))
= unList l ((h: int) (tl: list int) → return tl) fail
```

Ce qui est équivalent d'un point de vue opérationnel, mais a pour effet d'expanser systématiquement la VC de `tail` dans le contrat de l'appelant. Dans ce cas, la formule $\hat{\mathcal{A}}(\texttt{tail})$ ne change pas, en revanche $\hat{\mathcal{D}}(\texttt{tail})$ devient $\top$. C'est équivalent au cas précédent, sauf que nous n'avons pas spécifié `tail` inutilement.

Inférence de spécification. Le langage Coma nous permet de générer automatiquement des prédicats de spécification d'un handler h. C'est un mécanisme qui se révèle particulièrement utile pour des fonctions auxquelles nous ne voulons pas donner explicitement de spécification, mais avec lesquelles nous voulons raisonner [4]. À partir de la formule $\hat{\mathcal{A}}(h)$, des prédicats correspondants aux pré- et postconditions peuvent être calculés par une simple transformation syntaxique. L'ajout de l'annotation `[@coma:extspec]` à la définition d'un handler permet l'activation de cette génération, sur `tail` (sans barrières) nous obtenons

```
predicate tail'pre  (l: list int) = l ≠ Nil
predicate tail'post (l: list int) (r: list int) = exists h:int. l = Cons h r
```

La précondition d'un handler h correspond moralement à ce que nous obtiendrions avec un calcul de plus faibles préconditions traditionnel (WP) appliquée au corps de h et la postcondition triviale « $\mathrm{WP}(h_{body}, \top)$ ». Nous pouvons donc la récupérer à partir de la formule en mode appelant par η-expansion des paramètres et en donnant $\top$ comme VC de continuation :

$$\begin{aligned}
\texttt{tail'pre} &\equiv \lambda \ell\colon \texttt{list int}.\ \hat{\mathcal{A}}(\texttt{tail})\ \ell\ (\texttt{fun _} \mapsto \top) \\
&\equiv \lambda \ell\colon \texttt{list int}.\ \ell \neq \texttt{Nil} \wedge (\forall t\ell.\, (\exists h.\, \ell = \texttt{Cons}\ h\ t\ell) \to (\texttt{fun _} \mapsto \top)\ t\ell) \\
&\equiv \lambda \ell\colon \texttt{list int}.\ \ell \neq \texttt{Nil}
\end{aligned}$$

La postcondition d'un handler h correspond en revanche à une formule obtenue avec une transformation plus complexe. Cette dernière est assimilable à ce que nous pouvons obtenir en appliquant un calcul de plus forte postcondition (SP) au corps de h et à sa WP appliqué à une postcondition triviale « $\mathrm{SP}(h_{body}, \mathrm{WP}(h_{body}, \top))$ ». Pour ce faire, nous utilisons une opération de neutralisation notée $\natural$. Cette opération a pour effet de retirer les obligations de preuves propres à la formule à laquelle elle est appliquée. Sur $\hat{\mathcal{A}}(\texttt{tail})$, nous obtenons

$$\begin{aligned}
\hat{\mathcal{A}}^{\natural}(\texttt{tail}) = \lambda \ell\colon \texttt{list int}.\, \lambda ret\colon \texttt{list int} \to \mathrm{Prop}.\\
\forall t\ell.\, (\exists h.\, \ell = \texttt{Cons}\ h\ t\ell) \to ret\ t\ell
\end{aligned}$$

La postcondition de `tail` correspond à la partie gauche de la flèche dans la formule ci-dessus ; avec la différence que la variable quantifiée $t\ell$ doit être instanciée avec la valeur de retour du handler. Pour la récupérer nous appliquons d'abord une négation transformant l'implication en conjonction. Pour instancier correctement le quantificateur, nous ajoutons l'égalité souhaitée comme VC de continuation (que nous devons passer également sous une négation pour annuler la précédente). Ainsi, nous obtenons une conjonction entre la postcondition attendue et l'égalité qui force l'instanciation du quantificateur existentiel.

$$\begin{aligned}
\texttt{tail'post} &\equiv \lambda \ell, r\colon \texttt{list int}.\ \neg(\hat{\mathcal{A}}^{\natural}(\texttt{tail})\ \ell\ (\texttt{fun tl} \mapsto \texttt{tl} \neq r)) \\
&\equiv \lambda \ell, r\colon \texttt{list int}.\ \neg(\forall t\ell.\, (\exists h.\, \ell = \texttt{Cons}\ h\ t\ell) \to (\texttt{fun tl} \mapsto \texttt{tl} \neq t\ell)\ r) \\
&\equiv \lambda \ell, r\colon \texttt{list int}.\ \exists t\ell.\, \exists h.\, \ell = \texttt{Cons}\ h\ t\ell \wedge r = t\ell \\
&\approx \lambda \ell, r\colon \texttt{list int}.\ \exists h.\, \ell = \texttt{Cons}\ h\ r
\end{aligned}$$

Ce mécanisme d'extraction se généralise naturellement aux handlers ayant plusieurs continuations. En revanche cela ne s'applique pas aux handlers ayant des continuations

4. Un cas d'usage typique est le passage de fonction en paramètre. Par exemple pour `List.map f l`, nous voulons demander dans la précondition de `List.map` que la précondition de `f` soit vraie sur les éléments de `l`.

d'ordre supérieur. Enfin, remarquons que ce mécanisme d'extraction de spécification ne peut pas introduire d'incohérence logique, mais seulement mener à un but non prouvable. En effet, seuls des prédicats sont générés et non des lemmes ou des axiomes qui introduiraient des informations en contexte.

4 Application dans Creusot

Reprenons l'exemple de la figure 1, en gardant la spécification explicite de la clôture. Voici un fragment du code Coma produit par Creusot sur cet exemple :

```
1  module Map                (* traduction de `map` *)
2    type i type f
3    predicate f'pre  (f: f) (x: i32)
4    predicate f'post (f: f) (x: i32) (result: i32)
5    let map (i: i) (f: f) (return (r: iter_map_t i f))
6    = { forall x. emit i x → f'pre f x } ↑ ...
7      / break (result: iter_map_t i f)
8        = { forall x r. emit i x → emit result r → f'post f x r } ↑ return r
9  end
10
11 module Iter_Add_Closure (* traduction de la clôture *)
12   type clos_env = i32
13   let closure_call (f: clos_env) (x: i32) (return (r: i32))
14   = { i32_min ≤ x + f ∧ x + f ≤ i32_max } ↑ break (x + f)
15     / break (result: i32) = { result = x + f } ↑ return result
16 end
17
18 module Iter_Add          (* traduction de `iter_add` *)
19   use Iter_Add_Closure as Clos
20   type i
21   predicate precondition (f: Clos.clos_env) (x: i32)
22   = i32_min ≤ x + f ∧ x + f ≤ i32_max
23   predicate postcondition (f: Clos.clos_env) (x: i32) (result: i32)
24   = result = x + f
25
26   clone Map with type i, type f = Clos.clos_env,
27     predicate f'pre = precondition, predicate f'post = postcondition
28
29   (* `iter_add` utilise la fonction `map` chargée par le `clone` *)
30   let iter_add (i: i) (y: i32) (return (r: unit)) = ↑ ...
31 end
```

Chaque fonction Rust du programme initial est vérifiée dans un module différent. Les définitions Coma `map` (ligne 5), `closure_call` (ligne 13) et `iter_add` (ligne 30) doivent être vérifiées. Notons que le handler `map` est cloné par Creusot dans le module `Iter_Add` (lignes 26-27) en instanciant les prédicats abstraits `f'pre` et `f'post` avec `precondition` et `postcondition`. Ceci correspond à la monomorphisation de la fonction en Rust, et permet comme promis une vérification modulaire.

Cependant, nous savons que la spécification et la vérification de la clôture de notre exemple n'a pas vraiment lieu d'être : « la précondition (resp. postcondition) de la clôture est redondante avec la précondition (resp. postcondition) implicite de l'addition `x + y` » (page 2). Pour éviter d'écrire la spécification de cette clôture nous utilisons la méthode vu en section 3. Ainsi, nous enlevons la barrière opaque et les assertions de sa traduction Coma (lignes 13 à 15).

De plus, nous avons besoin de nous référer aux prédicats `precondition` et `postcondition` pour pouvoir écrire la version monomorphisée de `map`. C'est ici que le mécanisme d'extraction

Nom du programme	LoC	LoS	# Fonctions	Temps	Écart type
`bool_then.rs`	24	10	1	0,835	0,022
`bool_then.rs`+`extspec`	18	8	1	0,837	0,016
`option.rs`	52	31	11	0,975	0,032
`option.rs`+`extspec`	24	12	1	0,892	0,020
`iterator.rs`	42	15	9	2,794	0,043
`iterator.rs`+`extspec`	24	9	4	2,508	0,025
`avl.rs`	105	155	6	1,345	0,030
`avl.rs`+`extspec`	101	99	4	1,504	0,071

Table 1. Résultats de notre évaluation. La colonne « LoC » indique les lignes de code du programme que nous vérifions. La colonne « LoS » mesure les lignes de spécifications et d'assertions utilisées. La colonne « # Fonctions » mesure le nombre de fonctions devant être vérifiées dans le programme. La colonne « Temps » indique le temps moyen de vérification pour le programme entier en secondes (moyenne réalisée avec 10 exécutions). La colonne « Écart type » indique l'écart type du temps de vérification pour le programme entier en secondes.

de spécification de Coma intervient. Il offre la génération des prédicats `closure'pre` et `closure'post`, même pour un handler sans barrière.

Ainsi, le code Rust de la fonction `iter_add` (figure 1) tenant initialement sur 5 lignes peut se réduire à la simple ligne `map(i, |x: i32| x + y).sum()` et reste autant vérifié. Celui-ci est bien plus clair et concis que la version initiale.

Finalement, avec cette nouvelle version, le traitement de la fonction `iter_add` produit un seul module au lieu de deux.

```
1  module Iter_Add
2    use Iter_Add_Closure as Clos
3    let closure [@coma:extspec] (f: Clos.clos_env) (x: i32) (return (r: i32))
4    = return (x + f) (* pas de barrière *)
5    predicate precondition  (f: Clos.clos_env) (x: i32)
6    = closure'pre  f x
7    predicate postcondition (f: Clos.clos_env) (x: i32) (result: i32)
8    = closure'post f x result
9
10   (* ajouter les lignes 26-30 du programme précédent *)
11 end
```

Les conditions de vérification générées pour `precondition` et `postcondition` sont alors équivalentes à la version explicite.

5 Expérimentation et évaluation

Pour évaluer notre approche, nous mesurons la capacité de `extspec` à éliminer les spécifications utilisateur des programmes Rust. En utilisant des programmes extraits de la suite de tests de Creusot et de la bibliothèque standard de Rust, nous comparons le nombre de lignes de spécification avec et sans `extspec`, ainsi que le temps nécessaire pour vérifier ces programmes. L'évaluation est réalisée avec une machine ayant un processeur Intel Core i7-13800H, jusqu'à 5,2GHz, 14 cœurs et 32 Go de RAM.

Les résultats sont présentés dans le tableau 1. Nous vérifions quatre suites de tests : un petit programme utilisant la fonction `bool::then`, une collection de tests de l'API de `Option` dans la bibliothèque standard, une utilisation de `map`, et enfin une implémentation d'arbres binaires de recherche équilibrés [BFS16]. Les décomptes de lignes ont été obtenus après avoir exécuté le formateur `rustfmt` pour normaliser les fichiers. Pour obtenir les temps de calcul, nous utilisons le logiciel `hyperfine` [Pet23] et la commande `replay` de Why3.

L'exemple des arbres de recherche est particulièrement illustratif du gain obtenu par le mécanisme d'inférence de spécification. Autrement, les fonctions `rotate_left` et `rotate_right` auraient une spécification très longue, répétant en grande partie leur définition.

Il est également intéressant de noter que `iterator.rs+extspec` contient encore deux assertions, en raison de deux fonctions distinctes. Dans le premier cas, la spécification manuelle est nécessaire pour guider les prouveurs afin qu'ils puissent déduire l'égalité entre deux séquences. (La même spécification est présente sans inférence.)

Dans le second cas, l'assertion restante provient de la fonction `counter`, reproduite ci-dessous. Remarquons que nous utilisons la fonction `map_inv` plutôt que `map`. En effet, nous souhaitons spécifier l'itérateur `map` avec un paramètre supplémentaire `prod` qui correspond à la séquence des éléments déjà parcourus [FP16].

```
#[ensures(result@ == v@.len())]
pub fn counter<T>(v: Vec<T>) -> usize {
    let mut cnt: usize = 0;
    let _: Vec<_> = v.into_iter().map_inv(|x, prod| {
            proof_assert!(cnt@ == prod.len());
            cnt += 1; x }).collect();
    cnt
}
```

L'unique assertion présente dans la clôture lie les valeurs de `cnt` et la longueur de la séquence `prod`, ce qui est nécessaire pour prouver la postcondition de la fonction. Sans cette assertion, l'inférence `extspec` produirait une précondition trop faible.

6 Conclusion

Nous avons présenté l'utilisation du langage intermédiaire de vérification Coma comme backend de Creusot, un outil de vérification de programmes Rust. Nous avons mis au point une méthode permettant de vérifier des programmes utilisant des clôtures dont nous omettons la spécification, et nous l'avons vérifiée expérimentalement. Les travaux futurs incluent l'ajout des clôtures `dyn` dans Creusot et l'inférence de spécification pour des handlers d'ordre supérieur dans Coma.

Travaux connexes. Prusti [WBM+21] est un outil de preuves de programmes permettant également de vérifier du code Rust avec des clôtures qui capturent leur environnement. En revanche, il n'y a pas de mécanisme d'inférence de spécification et les clôtures doivent nécessairement être annotées avec des pré- et postconditions. D'autre méthodes telles que la bi-abduction [CDOY09, TLDC13] permettent d'inférer des propriétés de programmes impératifs. Cette dernière est principalement appliquée à des problèmes dans un cadre de logique de séparation. Coma étant basé sur une logique de premier ordre, nous pouvons utiliser un mécanisme moins complexe.

Accessibilité des sources. Les exemples discutés en section 5 sont disponibles à l'adresse `https://doi.org/10.5281/zenodo.14507773`.

Remerciements. Nous remercions Andrei Paskevich pour les nombreuses discussions, idées échangées et les conseils pour une utilisation judicieuse de Coma comme nouveau langage intermédiaire de Creusot. Nous remercions également les rapporteurs anonymes pour leurs remarques et leurs suggestions.

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

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs et K Rustan M Leino : Boogie : A modular reusable verifier for object-oriented programs. *In Formal Methods for Components and Objects : 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[BFS16] Guy E Blelloch, Daniel Ferizovic et Yihan Sun : Just join for parallel ordered sets. *In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2016.

[CCIM18] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala et Alain Mebsout : Alt-Ergo 2.2. *In SMT Workshop : International Workshop on Satisfiability Modulo Theories*, 2018.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn et Hongseok Yang : Compositional shape analysis by means of bi-abduction. *In Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

[DJ23] Xavier Denis et Jacques-Henri Jourdan : Specifying and verifying higher-order Rust iterators. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–110. Springer, 2023.

[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.

[DMB08] Leonardo De Moura et Nikolaj Bjørner : Z3 : An efficient SMT solver. *In International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich : Why3 — where programs meet provers. *In* Matthias Felleisen et Philippa Gardner, éditeurs : *Proceedings of the 22nd European Symposium on Programming*, volume 7792 de *LNCS*, pages 125–128. SV, 2013.

[FP16] Jean-Christophe Filliâtre et Mário Pereira : Itérer avec confiance. *In Journées Francophones des Langages Applicatifs*, 2016.

[MSS16] Peter Müller, Malte Schwerhoff et Alexander J Summers : Viper : A verification infrastructure for permission-based reasoning. *In Verification, Model Checking, and Abstract Interpretation : 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[Pet23] David Peter : hyperfine, 2023. `https://github.com/sharkdp/hyperfine`.

[PP24] Paul Patault et Andrei Paskevich : *Coma documentation*, 2024. `https://coma.paulpatault.fr`.

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

[TLDC13] Minh-Thai Trinh, Quang Loc Le, Cristina David et Wei-Ngan Chin : Bi-abduction with pure properties for specification inference. *In Programming Languages and Systems : 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings 11*, pages 107–123. Springer, 2013.

[WBM+21] Fabian Wolff, Aurel Bílỳ, Christoph Matheja, Peter Müller et Alexander J Summers : Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.