

漸進的借用検査を導入した Rust 言語処理系の検討

両角 颯 山崎 徹郎 千葉 滋

本論文では、静的検査と実行時検査を併用して強力なメモリ安全性を保証する漸進的借用検査と、それを Rust に導入した新しい漸進的型付けの言語処理系を提案する。Rust における借用検査は、借用が複製できない可変借用であるか、あるいはエイリアス可能な不変借用であるかを静的に指定することで、プログラム中のデータ競合を検出する。漸進的借用検査は、部分的に可変性の指定がないプログラムの借用検査を実行時まで遅延し、参照の可変性を動的に決定できるようにすることで、強力なメモリ安全性とプログラミングの容易さを両立させる。一部の型検査を実行時に行う漸進的型付けを Rust に導入するにあたっては、静的な型検査の結果を前提とする従来の静的借用検査器を利用することができないため、新たに必要となる漸進的借用検査を本論文で提案する。

1 はじめに

近年、低レベルなシステムプログラミングに適した言語として Rust が注目を集めており、既存のアプリケーションを Rust に移植するという取り組みが広まりつつある。Rust の特徴は、借用検査と呼ばれる静的検査によって、実行するプログラムがデータ競合を発生させないというメモリ安全性を保証することである。Rust コンパイラによるメモリ安全性の保証を得るためには、可変性の異なる二種類の参照型を適切に使い分ける必要があるが、これは Rust に親しみのない多くのプログラマにとって一般的な概念ではなく、新しく Rust を導入する際の障害となることがある。

そこで我々は、漸進的型付けの考え方に倣い、可変性の指定を省略したプログラムに段階的に可変性の指定を加えるための漸進的借用検査と、それを Rust に導入した新しい言語処理系を提案する。この処理系は、他の言語で書かれた既存のアプリケーションを、参照の可変性という概念を意識することなく Rust に

移植することを支援すると同時に、可変性の指定が省略されたプログラムを実行した場合においても、実行時検査を行うことで Rust が保証するメモリ安全性の保証と同等の保証を与える。

本論文の主たる貢献は以下のようにまとめられる。

- 漸進的借用検査が満たすべき振る舞いを示した (3.1 節)。
- 上記の振る舞いを実現するための動的借用検査器の設計と、静的な型検査器と借用検査器の拡張の概要を示した (3.2 節, 3.3 節)。
- 漸進的借用検査を既存の Rust コンパイラと統合する手法の概要を示した (3.5 節)。

2 可変性の指定を省略した参照

Rust は、並列プログラミングにおいてデータ競合を発生させないなどの強力なメモリ安全性を保証することから、低レベルなシステムプログラミングに適した言語として近年注目を集めている。既存のアプリケーションを Rust に移植することでメモリ安全性の保証を得るという取り組みが広まったことで、他の言語を主に扱うプログラマが新しく Rust を学ぶ機会も増えている。

Rust プログラミングにおいて正しくメモリ安全性の保証を得るためには、可変性の異なる二種類の参照

Examining a Rust Language Implementation with Gradual Borrow Checking

So Morozumi, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

```

1 struct User { name: String }
2 fn rename(user: &mut User,
3           new_name: &shr str) {
4     user.name = new_name.to_string();
5 }
6 fn main() {
7     let mut user = User {
8         name: "Alice".to_string() };
9     let new_name = "Bob";
10    rename(&mut user, &shr new_name);
11 }

```

図 1 参照の変性を指定したプログラムの例

型を適切に使い分けることが求められる。参照を用いる関数や変数の宣言には必ず変性の指定が必要となるが、変性の概念は Rust に親しみのない多くのプログラマにとっては一般的ではなく、この課題を回避する方策が求められている。

例えば図 1 は、参照型を引数に取る関数を定義して呼び出す Rust プログラムの例である。Rust では、参照型を引数に取る関数を定義する際にはその変性を明示する必要がある。rename 関数の引数である user (2 行目) と new_name (3 行目) の型注釈には、参照型の変性を指定するキーワード mut と shr が用いられている^{†1}。また、Rust では変数への参照を生成する場合にもその変性を指定する必要があり、main 関数の中では変数 user と new_name への参照を生成するために、キーワード mut と shr を用いて変性を指定している (10 行目)。

既存のアプリケーションを Rust に移植するにあたって、これらの変性の指定は Rust プログラミングにおいて避けることのできない問題であるが、変数や関数の変性を指定するためにはプログラムの構造を慎重に検討する必要があり、迅速な移植を行うた

^{†1} Rust には変性が不変であることを示す明示的なキーワードは存在しないが、本稿ではわかりやすさのため、Shared Reference に倣ってキーワード shr を明記することとする

めの妨げとなる場合がある。

3 漸進的借用検査

我々は、漸進的型付け (Gradual Typing) の考え方に倣い、変性の指定がないプログラムに段階的に変性の指定を加えるための、漸進的借用検査 (Gradual Borrow Checking) を導入した言語処理系を提案する。漸進的借用検査は、部分的に可変性の指定を省略したプログラムを実行した場合でも、実行時に借用検査を行うことによって、Rust によって保証されるものと同等のメモリ安全性を保証する仕組みである。

Rust が保証するメモリ安全性とは、複数のスレッドが同時に同じメモリ領域にアクセスする場合に、そのうち少なくとも一つが書き込みを行うことによって発生するデータ競合を引き起こさないというものがある。Rust コンパイラは借用検査と呼ばれる静的検査によって、あるメモリ領域に書き込みを行うことができるスレッドがある場合に、他のスレッドがそのメモリ領域にアクセスできないようにすることによってこのメモリ安全性を保証する。

借用検査が検査のために利用するのが、参照型に指定する必要がある二種類の変性 (Mutability) である。変性とは、あるメモリ空間に対して変更を加えることができるかどうかを決定するプロパティであり、不変を表す shr と可変を表す mut の二つがある。可変であると宣言されたメモリ空間が読み取りと書き込みの双方の操作を受け付けるのに対して、不変であると宣言されたメモリ空間は読み取ることのみが許容され、書き込みを行うことはできない。

Rust にはこれらの変性に対応した参照型がそれぞれ存在し、異なる制約が与えられている。可変参照 (Mutable Reference, &mut T と表記する) は、その参照が指し示す先のメモリ空間を読み書きすることができる参照であるが、ある瞬間に同じメモリ空間を指す参照が他に存在することはできないという制約を持つ。変性が不変であるような参照は共有参照 (Shared Reference, &shr T と表記する) である。これは参照が指し示す先のメモリ空間に対して読み取りのみが許容され、書き込みを行うことができない

参照であるが、可変参照とは異なり、同じメモリ空間を指す共有参照が同時に複数存在することが許容される。

従来の漸進的型付けと異なり、漸進的借用検査による実行時の借用検査においては、同じメモリ領域を参照する複数の参照の状態が相互に影響を与えることがある。漸進的型付けでは、実行時の型キャストによって影響を受けるのは操作の対象となる値のみであり、一度型キャストを行った値の型が実行中に変化することはない。一方、漸進的借用検査では、同じメモリ領域を参照する他の参照の状態によって自身に対して許容される操作が変化するため、より複雑な状態管理が必要となる。加えて、一度型キャストを通過した参照であっても、その後に他の参照に対して行った操作によって参照が無効化される可能性がある。

漸進的借用検査においては、実行時の借用検査の他にも、静的な借用検査と型検査を拡張する必要がある。漸進的型付けにおいても必要となる型キャストの挿入に加えて、漸進的借用検査においては実行時の借用検査のために参照の有効期間を通知する処理を挿入しなければならない。また、従来の漸進的型付けの型システムは静的型と動的型の二種類の型を持つが、可変性の指定が省略された参照型はこれらの型のいずれにも該当しない。参照型における可変性の指定は変性 (Variance) に影響を与えることが知られており、可変参照 `&mut T` は型パラメータ `T` に対して不変であるが、共有参照 `&shr T` は共変である。我々は可変性の指定が省略された参照型に対する変性を定義し、さらなる実行時検査の必要性を指摘する。

3.1 漸進的借用検査の振る舞い

漸進的借用検査を導入した処理系では、部分的に可変性の指定を省略したプログラムを実行することができる。図 2 は漸進的借用検査を導入した言語のプログラムの例である。`ref1` や `ref2` の型注釈のように、特定の可変性を示すキーワードである `shr` や `mut` の代わりにキーワード `unk` を用いることで、可変性の指定を省略し、漸進的借用検査による可変性の決定を

```
1 let root: &mut i32 = &mut 42;
2 let ref1: &unk i32 = root;
3 *ref1 = 13;
4 let ref2: &unk i32 = root;
5 println!("{}", *ref2);
```

図 2 漸進的借用検査を通過するプログラムの例

行うことを指示する^{†2}。

漸進的借用検査では、省略されたすべての可変性を適切に補うことで静的な借用検査を通過するようなプログラムの実行を許し、可変性を補ったプログラムと同様の結果を出力する。例えば、図 2 のプログラムは、`ref1` の可変性を `mut`、`ref2` の可変性を `shr` と補うことで静的な借用検査を通過する。よって図 2 のプログラムは実行可能であり、`ref2: 13` という出力を得る。

一方で、静的な借用検査を通過するような可変性の補い方が存在しないようなプログラムでは、漸進的借用検査は違反が明らかとなる最初の操作を検出し、実行時エラーを出力して実行を停止する。例えば、図 3 は、静的な借用検査を通過するような可変性の補い方が存在しないプログラムの例である。`ref1` を通じて読み取り操作を行った時点 (4 行目) までは、`ref1` と `ref2` の可変性をともに `shr` と補うことで静的な借用検査を通過することができる。しかしながら、`ref2` を通じて書き込み操作を行う (5 行目) ためには、`ref2` の可変性が `mut` であると補う必要がある。このとき、静的な借用検査では、`ref2` を可変参照として借用した時点 (3 行目) で、共有参照 `ref1` と可変参照 `ref2` が同時に存在することをエラーとして報告する。このような場合、漸進的借用検査では、5 行目の書き込み操作を実行する前に実行時エラーを出力し、プログラムの実行を停止する。

漸進的借用検査は、実行時に通過した経路のみを考慮して可変性を補い、違反がないことを検査する。

^{†2} 可変性が明記されていない参照は Rust においては共有参照を意味するため、本稿では、可変性の省略をキーワード `unk` を用いて明記することとする

```

1 let root: &mut i32 = &mut 42;
2 let ref1: &unk i32 = root;
3 let ref2: &unk i32 = root;
4 println!("ref1: {}", *ref1);
5 *ref2 = 13;

```

図3 漸進的借用検査を通過しないプログラムの例

```

1 let root: &mut i32 = &mut 42;
2 let ref1: &unk i32 = root;
3 let ref2: &unk i32 = root;
4 if rand::random_bool(0.5) {
5     println!("{}", *ref1);
6 } else {
7     *ref2 = 13;
8 }

```

図4 静的借用検査は通過しないが、漸進的借用検査を通過するプログラムの例

例えば、図4は、1/2の確率で5行目か7行目のどちらかの操作のみを行うプログラムである。このプログラムに対して静的な借用検査を行うと、図3のプログラムと同じ理由で違反が報告される。しかしながら、漸進的借用検査においては、`ref1`を常に`shr`と補い、`ref2`については、5行目が実行された場合には`shr`、7行目が実行された場合には`mut`と補うことができる。このような場合、漸進的借用検査は実行時エラーを出力することなくプログラムの実行を許す。

3.2 動的借用検査器による実行時の借用検査

漸進的借用検査では、静的な借用検査を通過するような可変性の補い方が存在することを動的借用検査器によって検査する。本節では、この動的借用検査器の概要を述べる。具体的な可変性がプログラム上で指定されている参照を静的な参照、可変性が省略され静的に定まっていない参照を動的な参照と呼ぶ。動的な参照の可変性を実行時に決定する動的借用検査器には、新しい参照を生成する操作と、参照を解放して権限を返却する操作の二つの操作が存在する。

動的な参照を通じて読み書きを行うためには、目的の操作を行うのに十分な可変性を持つ静的な参照を生成し、その静的な参照を通じて操作を行う。通常、読み取りを行うためには共有参照へのキャスト、書き込みを行うためには可変参照へのキャストを試みる。

Rustにおいて参照を生成する操作には、メモリ空間を所有する変数から参照を生成する借用と、ある参照から同じメモリ空間を指す別の参照を生成する再借用の二種類がある。簡単のため、これ以降の議論では、動的な参照を生成する操作が、既存の参照から新しい参照を生成する再借用のみによって行われることとする。変数から借用によって動的な参照を生成する操作は、一度静的な参照を借用によって生成し、その参照から動的な参照を再借用することで実現できるため、この制限が自由度を損なうことはない。

動的借用検査器において新しい参照を生成する操作は、元となる参照と新しい参照の可変性の指定の有無によってさらに三種類に分類することができる。なお、静的な参照から静的な借用を新しく生成する操作は、静的な借用検査によって完全に検査されるため、動的借用検査器では検査しない。

静的から動的 静的に可変性が注釈された参照から可変性が省略された参照を生成する。元となった参照の可変性によって新しく生成される参照の可変性の上限が決定される。

動的から静的 参照先のメモリ空間に対して何らかの操作を行うため、必要な可変性を要求し、静的な可変性を持つ参照を生成する。要求する可変性がメモリ安全性を侵害する可能性がある場合には、実行時エラーを生じさせる。

動的から動的 動的な参照を複製する。複製された参照は元の参照とは異なる可変性を持つことができる。

動的借用検査器によって生成される動的な参照はそれぞれ、自身が持つ権限の下限 l と上限 u に加えて、他の参照から再借用されているために一時的に無効化されている権限 b の三つの権限値を管理する。権限は Fractional Permissions [1] に類似する概念で

表 1 動的な参照の権限の初期化。
 a_p は元となる参照の有効な権限を表す

元の参照	新しい参照	下限 l	上限 u	無効 b
&shr	&unk	0	ϵ	0
&mut	&unk	0	1	0
&unk	&unk	0	a_p	0
&unk	&shr	ϵ	ϵ	0
&unk	&mut	1	1	0

あり、0 と 1 の他に、 $0 < \epsilon < 1$ を満たす十分小さい実数値を表す ϵ を用いて表記され、0 以上 1 以下の実数値を取る。動的な参照が管理する三つの権限値は、常に $b \leq l \leq u$ という関係を保つ必要がある。

静的借用検査を通過するような可変性の選び方は、動的な参照の上限 u によって決定できる。 $u = 1$ であるような参照は、shr と mut のどちらを補っても静的な借用検査を通過することができる。一方、 $u = \epsilon$ であるような参照は、shr と補うことで静的な借用検査を通過することができるが、mut と補うことはできず、実行時エラーを発生させる。 $u = 0$ であるような参照はどの可変性を補っても静的な借用検査を通過することができないため、この参照を通じていかなる操作も行うことはできない。

表 1 に示すのは、再借用によって参照を生成する際に新しく生成される参照の権限の初期値である。静的な参照から動的な参照を生成する際には、元となる静的な参照の可変性に対応する権限を上限とする。動的な参照から動的な参照を生成する際には、元となる参照の上限 u_p から無効化されている権限 b_p を引いた値を上限とする。この値のことを有効な権限 $a_p = u_p - b_p$ と呼ぶ。元となる参照の種類に関わらず、動的な参照の権限の下限は 0 で初期化される。動的な参照から静的な参照を生成する場合は、可変性に対応する権限を下限として新しい参照を生成する。このとき、新しい参照の権限の下限 l_n は元となる参照の有効な権限 a_p 以下でなければならない、特に静的な参照を生成する場合は ϵ として $0 < \epsilon < a_p$ を満たす十分に小さい値を取る。

ある参照の権限の下限 l_c が l'_c へと増加したか、ま

たは $l_c = 0$ より大きい下限 l'_c を持って生成された場合、その参照の元となった参照について、無効化された権限 b_s を $b'_s = b_s + (l'_c - l_c)$ へと更新する必要がある。この更新によって無効化された権限 b'_s が自身の権限の下限 l_s を上回った場合、権限の大小関係 ($b'_s \leq l_s \leq u_s$) が保たれるように自身の権限の下限 l_s を更新しなければならない。ここで自身の権限の下限 l_s を更新したならば、さらにその親の参照においても同様の規則に従って無効化された権限を更新する。

また、ある参照の権限が更新された場合、その参照の子にあたる参照の権限の上限 u_c を更新する必要がある。子の権限の上限 u_c は、子の権限の下限 l_c 、その親の権限の上限 u_p と無効化された権限 b_p を用いて、 $u_c = u_p - b_p + l_c$ である。この更新によってある参照の権限の上限が変化した場合、その参照のすべての子についても同様の規則に従って権限の上限を更新する必要がある。

ある参照の権限が変化した場合に同時に権限が変化する可能性がある、共通の祖先を持つ参照の集合を記憶するため、動的借用検査器は再借用の操作によって生成される参照を参照グラフと呼ばれるグラフ構造で管理する。参照グラフ中のある参照を示すノードは、自身を生成する元となった参照を唯一の親として持ち、自身を元として新しく生成された複数の参照を子として持つ。よって参照グラフは、静的な参照から生成された動的な参照を根ノード、動的な参照から生成された動的な参照を中間ノードとし、動的な参照から生成される静的な参照を葉ノードとするような木構造となる。

参照を解放して権限を返却する操作を行うと、すべての権限が 0 へと更新され ($l = u = b = 0$)、以降その参照を通じていかなる操作も行うことができなくなる。自身の直接の子が権限を返却した場合、返却前の子の権限の下限 l_c を用いて、自分の無効化された権限 b_s を $b'_s = b_s - l_c$ へと更新する。

動的借用検査器の実装上、必ずしも権限 ϵ を具体的な実数値として扱う必要はなく、権限を 0 か 1 か ϵ かの三値として扱うことが可能である。その場合、無効化された権限 b が ϵ であるような参照では、すべて

の子が権限を返却し $b = 0$ になったことを検出できるように、現在権限 ϵ を貸し出している参照の数を記憶するためのカウンタを持つ必要がある。

3.3 静的検査と動的検査の協調

漸進的借用検査では、動的借用検査器が持つ参照の生成と解放の操作をプログラム中の適当な位置に挿入するよう、静的な型検査器と借用検査器を拡張する必要がある。

静的型検査器の主な役割は、可変性が異なる複数の参照型の間のキャストを挿入することである。同じ型の参照型の間のキャストは、共有参照 `&shr T` から可変参照 `&mut T` へのキャストを除いてすべてが許容され、再借用として実装される。このとき、動的な参照 `&unk T` が関与する参照型のキャストについては、動的借用検査器による追加の処理を挿入する必要がある。

また、静的型検査器は可変性が省略された参照を含んだプログラムの静的型付けを行う必要がある。可変性が省略された参照を含むシステムでは、サブタイプ関係 $T <: U$ を持つ型 T と U について、可変性が省略された参照型 `&unk T` と `&unk U` の間のサブタイプ関係を定義する必要がある。可変性が定まっている参照型においては、共有参照が共変である (`&shr T <: &shr U`) のに対して、可変参照は不変である (`&mut T <: &mut U` only if $T = U$) 。

漸進的借用検査では、可変性が省略された参照が実行時検査によって共有参照であると決定される可能性があるため、可変性が省略された参照型は共変である (`&unk T <: &unk U`) と定義する必要がある。しかしながら、共変性を用いて型キャストを行った参照の可変性が実行時に可変であると判定された場合、その参照を通じて行われる書き込み操作は型安全性を侵害する可能性があるため、共変性を用いて実行された型キャストを追跡し、実行時エラーを生じさせる必要がある。

漸進的借用検査における静的借用検査器の主な役割は、それぞれの参照が有効になる期間を静的に解析し、動的借用検査器に提供することである。簡単には、参照の有効期間はその参照を生成した時点から始

まり、その参照が最後に使用された時点で終わる [4]。プログラムの実行中に動的借用検査器が単独で参照の最後の使用を解析することは難しいため、静的借用検査器によって静的に解析し、参照を解放する処理を挿入することによって動的借用検査器に通知する。

また、再借用が行われると、新しい参照の有効期間が開始すると同時に、元となった参照は一時的に無効化される。無効になった参照は、新しく生成された参照の有効期間が終了した時点で再び有効になる。再借用による参照の無効化は、新しく生成される参照の生成と解放の操作によって行われるため、特別な処理を挿入する必要はない。

3.4 実行時のパフォーマンス改善

静的な借用検査と同等のメモリ安全性を保証することができる動的借用検査器の実装は複数考えられるが、動的な参照を用いることによるオーバーヘッドを抑えるため、パフォーマンスの高い実装を選択することが望ましい。オーバーヘッドは実行時間とメモリ使用量の双方の観点から考慮する必要があるが、ここでは双方を同時に改善する手法の一つを提案する。

動的借用検査器では、ある参照の権限が変化した場合、変化があった参照と親子関係を持つ参照グラフ中の参照に対して権限の更新を繰り返し行う必要がある。このとき、権限が変化した参照自身や、権限が変化した参照を子孫に持つ参照は、権限の下限や無効化された権限を正確に一度だけ更新する必要があるため、権限の更新を即座に反映することが望ましい。一方、権限の上限は、権限の下限や無効化された権限が常に正しく計算されているならば、参照グラフの根ノードから順に関係式 $u_c = u_p - b_p + l_c$ を適用することで任意の時点で正しい値を得ることができる。

そこで、権限の上限の更新をその値が必要になるまで遅延することを考えると、権限の更新を即座に行う必要がある参照は、権限が変化した参照の先祖のみに限定される。ある参照の権限の上限は、その参照の先祖に権限の下限と無効化された権限を問い合わせることで計算できるため、参照グラフ上で子の参照を探索する必要がなくなる。これによって、参照グラフは Parent Pointer Tree によって実装すること

が可能となる。Parent Pointer Tree は親のノードへの参照のみを持つ木構造であり、子ノードの一覧という可変長のデータ構造を持たないため、メモリ使用量を抑えることができる。また、可変性の通知を遅延し複数回の等価な通知を取りまとめることによって、権限の変化に応じて状態変化する参照の数が減り、計算量を削減することができる。

この手法で通知の遅延を行う場合、親に記録されたすべての通知のうち、自身が反映すべき通知を識別するため、単調増加するようなタイムスタンプを用いて通知を識別する必要がある。遅延された通知を反映する際には、通知に記録されたタイムスタンプと自身がすでに反映済みのタイムスタンプを比較し、必要な通知のみを反映する。

3.5 コンパイラの実装

可変性の指定を省略することができる Rust 言語（以降、GRust とする）のプログラムは、Rust コンパイラを拡張して開発した GRust コンパイラと、従来の Rust コンパイラを組み合わせてコンパイルする。GRust コンパイラの役割は、GRust のプログラムの型解析と借用検査を行い、動的借用検査器の呼び出しを含む Rust のプログラムへと変換することである。GRust コンパイラによって生成される Rust のプログラムは、Rust ライブラリとして実装された動的借用検査器とあわせて従来の Rust コンパイラによってコンパイルすることで、実行可能なバイナリを生成する。

可変参照と共有参照はどちらも実行時には同じポインタ型の値を持つので、従来の Rust コンパイラでは再借用の処理は借用検査後に削除される。一方、GRust コンパイラは、動的な参照が関与する再借用の処理に、動的借用検査器における参照を生成する関数の呼び出しを挿入する。また、GRust コンパイラは Rust コンパイラの静的借用検査を拡張して参照が最後に使用される位置を特定し、参照を解放する関数の呼び出しを挿入する。

GRust プログラムから Rust プログラムへのプログラム変換は、Rust コンパイラの持つエラー診断機能を利用してコード変換を提案するという形で行

う。Rust コンパイラが出力するコード変換の提案は、`rustfix` と呼ばれるツールによって自動的に適用することができるため、GRust プログラムのコンパイル手続きを自動化することが可能となる。

4 関連研究

静的型付けと動的型付けを統合する Gradual Typing に関する研究は数多く行われている。Siek らは Gradual Typing に求められる要件として Criteria for Gradual Typing [3] を提案した。この要件における不明型 \star を省略された可変性に自然に拡張することで、漸進的借用検査が満たすべき性質を定義することができる。現時点での我々の漸進的借用検査は、Blame Tracking [2] によるエラー報告を行わないため、この要件を部分的に満たさない可能性がある。一方で、我々の提案は静的検査を通過しないプログラムの実行を停止し実行時エラーを出力することを要件とするが、Siek らの要件には静的検査を通過しないプログラムに対する保証は含まれない。

Rust には、実行時に追加の検査を行うことを前提に、部分的にプログラムの借用検査を回避する方法として、`UnsafeCell` や `RefCell` などの参照型が存在する。`UnsafeCell` は、共有参照を通じて書き込みを行うことを許容する参照型である。プログラマは `UnsafeCell` を利用する際に、静的な借用検査とは異なる方法でデータ競合が生じないことを保証する必要がある。`RefCell` は、書き込みと競合する別の参照が存在しないことを実行時に検査することで、データ競合を防ぎながら静的な借用検査では許容されないプログラムの実行を許容する参照型である。これらの参照型を利用する際には、プログラマは利用する参照の可変性を明示的に指定する必要がある。漸進的借用検査の目的はこれらの参照型とは異なり、参照の可変性の指定を省略することができるようにすることであり、またそのようなプログラムを実行しても静的な借用検査と同様の振る舞いを実行時の検査によって保証することである。

Rust プログラムの検証を目的として Rust の振る舞いを形式化した研究も存在する。RustBelt [5] は Rust のサブセットであるような言語の振る舞いを

形式化しており, unsafe なコードを含む Rust プログラムの正しさを検証するために利用することができる. RustBelt は借用検査を型システムの一部に内包しており, 実行時検査を行うことを目的としない. RustSEM [4] は型システムとは独立に借用検査の動的意味論を形式化した研究であり, 検査にあたっては我々の動的借用検査器と同様の参照グラフを用いている. Tree Borrows [6] は unsafe なコードを含む Rust プログラムの正しさを検証するための形式化であり, RustSEM や我々の動的借用検査器と同様に参照グラフを用いた競合検査を行う. Tree Borrows の目的は借用検査を通過しないプログラムが未定義動作を引き起こすかどうかを検査することであり, 我々の漸進的借用検査が目指す, 実行時検査による静的な借用検査の振る舞いの模倣とは異なる.

5 まとめと今後の課題

借用検査によって強力なメモリ安全性を保証する Rust は低レベルなシステムプログラミングに適した言語として注目を集めているが, 新しく Rust を学ぶ多くのプログラマにとって, 可変性の異なる二つの参照型を使い分けるといった概念は馴染みがなく, 難しい. そこで我々は, 可変性の指定を省略したプログラムを実行した場合においても, 実行時にその可変性を補いながら借用検査を行うことで, Rust が保証するメモリ安全性の保証と同等の保証を与えることができる漸進的借用検査の仕組みと, それを Rust に導入した新しい言語処理系を提案した.

漸進的借用検査を導入した言語のコンパイラは現在実装の初期段階にあるが, この実装が完成した際には, 既存の Rust プログラムから可変性の指定を削除した場合に正しい可変性を補うことができ, 実行

時の挙動が変化しないことを検証することを目指している. 加えて, 提案した動的借用検査器の最適化が, 最適化を適用しない実装と等価な挙動を示すことを検証し, この最適化による実行時間とメモリ消費量の改善を計測することを予定している. また, 漸進的借用検査, 特に動的借用検査器が静的な借用検査と同様の振る舞いをするということについて, RustBelt や RustSEM などの形式化に基づく証明を与えることは, 今後の課題としている.

参考文献

- [1] Boyland, J.: Checking Interference with Fractional Permissions, *Static Analysis*, Goos, G., Hartmanis, J., Van Leeuwen, J., and Cousot, R.(eds.), Vol. 2694, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 55–72. Series Title: Lecture Notes in Computer Science.
- [2] Findler, R. B. and Felleisen, M.: Contracts for higher-order functions, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, Pittsburgh PA USA, ACM, September 2002, pp. 48–59.
- [3] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland: Refined Criteria for Gradual Typing, *LIPICs, Volume 32, SNAPL 2015*, Vol. 32(2015), pp. 274–293.
- [4] Kan, S., Chen, Z., Sanan, D., Lin, S.-W., and Liu, Y.: An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing, 2018. Version Number: 2.
- [5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer: RustBelt: securing the foundations of the Rust programming language, *Proceedings of the ACM on Programming Languages*, Vol. 2, No. POPL(2018), pp. 1–34. Publisher: Association for Computing Machinery (ACM).
- [6] Villani, N., Hostert, J., Dreyer, D., and Jung, R.: Tree Borrows, *Proceedings of the ACM on Programming Languages*, Vol. 9, No. PLDI(2025), pp. 1019–1042.