

東京大学
情報理工学系研究科 創造情報学専攻
修士論文

簡便な Rust プログラミングのための漸進的借用検査
Gradual Borrow Checking for Simplifying Rust Programming

両角 颯
So Morozumi

指導教員 千葉 滋 教授

2026年1月

概要

本論文では、Rust プログラムにおける参照の可変性の指定を省略可能にしながらも、静的検査と実行時検査を組み合わせることによってメモリ安全性を保証することができる、漸進的借用検査のアルゴリズムを提案する。Rust は、借用検査と呼ばれる静的検査によって、プログラムがデータ競合を引き起こさないという強力なメモリ安全性を保証する。Rust プログラムには参照の可変性を明示的に指定する必要があるが、借用検査の概念は Rust に特有のものであり、新しく Rust に入門するプログラマにとって、正しく可変性を指定してコンパイルを成功させることは困難である。漸進的借用検査は、漸進的型付けの考え方に基づき、Aliasing XOR Mutability の規則が満たされることを実行時に検査することで、メモリ安全性を保証する。本手法を導入した言語では、実行時にメモリ安全性を侵害しない場合に限り、静的検査を通過しない箇所を含むプログラムであっても安全に実行することができる。これにより、迅速にプロトタイピングを行うことが可能となり、さらに、段階的に可変性の指定を追加することで性能と安全性を向上させることができる。

Abstract

In this paper, we propose an algorithm for Gradual Borrow Checking that guarantees memory safety by combining static and runtime checks, while allowing the omission of mutability annotations for references in Rust programs. Rust ensures strong memory safety, specifically freedom from data races, through a static analysis known as borrow checking. Since Rust requires explicit mutability annotations for references and the concept of borrow checking is unique to Rust, it is challenging for programmers new to the language to correctly annotate mutability and successfully compile their code. Based on the principles of gradual typing, Gradual Borrow Checking guarantees memory safety by enforcing the “Aliasing XOR Mutability” rule at runtime. With a language implementing this approach, programs that do not fully pass static checks can be executed safely, provided they do not violate memory safety at runtime. This facilitates rapid prototyping, and allows performance and safety to be improved by gradually adding mutability annotations.

目次

第 1 章	はじめに	1
第 2 章	Rust と漸進的型付け	4
2.1	データ競合を防ぐための Aliasing XOR Mutability の規則	4
2.2	Rust と借用検査	6
2.3	漸進的型付け	10
2.4	Rust への移行における可変性の指定の難しさ	14
第 3 章	漸進的借用検査	18
3.1	漸進的借用検査による可変性の指定の省略	18
3.2	実行時検査のアルゴリズム	18
3.3	漸進的借用検査の Rust 言語への統合	26
3.4	Gradual Guarantee を満たすための設計	30
第 4 章	評価	31
4.1	実験設定	31
4.2	実験結果	33
4.3	Rust コンパイラの拡張の実現可能性	42
第 5 章	おわりに	45
	発表文献と研究活動	47
	参考文献	48

第 1 章

はじめに

近年、様々なソフトウェアの分野で既存のプログラミング言語を置き換える新しい言語として Rust^{*1}が注目を集めている。例えば、特に高いパフォーマンスが求められる Web アプリケーションサーバの開発においては、従来は Go が広く用いられてきたが、近年では Rust を採用する事例が増えている。Go と Rust はいずれも静的型付けを採用し、事前にコンパイルを行うことで高い実行性能を実現している点で共通している。しかしながら、Go と比較して Rust は、ガベージコレクションを用いずにメモリ管理を行うことで予測可能な実行性能を提供する点や、パターンマッチングなどの高度な言語機能を備え、プログラミング体験が優れている点で評価されている。

同様に Rust への関心が高まっている分野として、オペレーティングシステムや組み込みシステムをはじめとする、低レベルなシステムソフトウェアの開発が挙げられる。これらの分野では、実行性能やメモリ消費量に厳しい制約が課されることから、ガベージコレクションを用いず、プログラマが手動でメモリ管理を行う C 言語を用いることが一般的である。しかしながら、優秀なプログラマにとっても手動で正しくメモリ管理を行うことは困難であり、多くのプログラマのレビューを受けたオープンソースソフトウェアにおいても、誤ったメモリ管理に起因する脆弱性が日常的に報告されている。加えて、現在広く用いられている他のプログラミング言語と比較して、C 言語には高度な言語機能が不足しており、プログラミング体験が劣るという問題がある。特に、このような言語機能の不足を補うために利用されるマクロやプリプロセッサは、プロジェクト固有の複雑なコード生成を引き起こし、プログラムの可読性や保守性を低下させる傾向がある。

システムソフトウェア開発において Rust が注目されている背景には、C 言語と同等の実行性能を提供するにも関わらず、プログラマによる手動でのメモリ管理を必要とせず、高度な言語機能を用いて保守性の高いプログラムを記述することができるという特徴がある。このような特徴から、新規のシステムソフトウェア開発に Rust を採用するだけでなく、C 言語で記述された既存のソフトウェアを Rust に移植する試みが進められている。既存のソフトウェアを Rust に移植することで、メモリ管理に起因する未知の脆弱性を排除しつつ、プログラムの保守

^{*1} <https://rust-lang.org/>

2 第1章 はじめに

性を向上させることが期待される。また、C 言語との間の FFI (Foreign Function Interface) が整備されている点や、必要に応じて `unsafe` ブロックを用いることによって C 言語と同等の低レベルなメモリ操作を行うことができる点も、C 言語から Rust への移行を促進している。

他のプログラミング言語には見られない Rust の特徴は、借用検査と呼ばれるプログラムの静的検査を行うことである。借用検査はメモリ領域に対する所有権と借用の概念に基づいており、借用検査を通過した Rust プログラムは、並行プログラミングにおいて実行時にデータ競合を引き起こさないという強力なメモリ安全性が保証される。また、借用検査の結果を用いることでレジスタ割り当ての最適化やキャッシュ効率化を行うことができ、高い実行性能を実現することにも寄与している。

Rust でプログラムを記述する際には、すべての参照に可変性を明示的に指定する必要がある。可変性とは、参照を通じてデータを書き換えることができるかどうかを示す属性である。借用検査は、プログラムがデータ競合を引き起こさないために、参照に指定された可変性が `Aliasing XOR Mutability` と呼ばれる規則を満たしていることを要求する。しかしながら、この規則は他の著名なプログラミング言語においては一般的でなく、Rust に親しみのない多くのプログラマにとって、新しく Rust を導入する際の障害となることがある。Rust の導入を試みたプログラマの多くが、借用検査によって引き起こされる数多のコンパイルエラーに直面し、そのエラーの原因を理解することに苦労するようである。

そこで我々は、漸進的型付け (Gradual Typing) の考え方を Rust に導入し、Rust プログラミングを簡便にすることを目指す。漸進的型付けとは、静的型付けと動的型付けの双方の利点を活かすため、型注釈が部分的に省略されたプログラムを実行することを可能にする手法である [1]。漸進的型付けでは、型注釈が記述された部分を静的に型検査し、型注釈が省略された部分については実行時に型検査を行うことで、静的型付けと同等の型安全性を保証する。漸進的型付けを採用したプログラミング言語では、動的型付けのように型注釈を記述することなく効率よくプロトタイピングを行った後、必要な部分に段階的に型注釈を追加することによって実行性能と安全性を向上させることができる。

本論文では、Rust の参照における可変性の指定を省略可能にし、静的な借用検査を通過しない箇所を含むプログラムであっても、実行時にメモリ安全性を侵害しないことが保証される限り安全に実行できるようにすることを目指す。参照における可変性の指定が省略できるようになることで、C 言語で記述されたプログラムを単に Rust に移植しただけでも、メモリ安全性の保証を得ながらプログラムを実行することが可能となる。これによって、既存の C 言語で記述されたプログラムには含まれない参照の可変性を新しく検討することなく、容易に C 言語で記述されたプログラムを Rust へと移行できるようになることが期待される。さらに、段階的に可変性の指定を追加していくことでプログラムの実行性能と安全性を向上させ、最終的には静的な借用検査を通過する完全な Rust プログラムへと移行することができる。

このような目的を達成するため、部分的に可変性の指定が省略された参照を含むプログラムに対して、実行時に借用検査を行うことで `Aliasing XOR Mutability` の規則を満たすことを保証する、漸進的借用検査 (Gradual Borrow Checking) のアルゴリズムを提案する。漸進的借用検査は、静的な借用検査と実行時検査を組み合わせることで、部分的に参照の可変性の指

定が省略された Rust プログラムを実行した際に Aliasing XOR Mutability の規則を満たすことを保証する。

本論文の主たる貢献は次のようにまとめられる。

- 静的な借用検査と実行時検査を組み合わせる漸進的借用検査のアルゴリズムを示した (3.2 節)。漸進的借用検査は、参照の木構造に基づいてそれぞれの参照の権限を計算することで、プログラムが実行時に Aliasing XOR Mutability の規則を満たすことを検査する。
- 漸進的借用検査を Rust に統合する手法を提案した (3.3 節)。参照における可変性の指定を省略することができる Rust 言語は、Rust コンパイラの拡張と、通常の Rust 言語で実装したライブラリの組み合わせによって実現される。
- 実行時検査器を Rust ライブラリとして実装し、性能への影響を評価した (第 4 章)。参照の可変性の指定を削除するに従って実行時間が増加する傾向が見られたが、メモリ消費量への影響は多くの場合で小さいことが分かった。

本論文は次のように構成される。第 2 章では、Aliasing XOR Mutability の規則と借用検査、漸進的型付けについて説明し、漸進的借用検査を提案する背景を示す。第 3 章では、参照の木構造に基づいた権限の計算による漸進的借用検査のアルゴリズムを詳述する。また、Rust に漸進的借用検査を統合する手法についても説明する。第 4 章では、実行時間とメモリ消費量の観点から実装した実行時検査器の性能への影響を評価する。また、プログラム変換を行うための Rust コンパイラの拡張の実現可能性についても議論する。第 5 章では、本論文のまとめと今後の課題について述べる。

第 2 章

Rust と漸進的型付け

2.1 データ競合を防ぐための Aliasing XOR Mutability の規則

複数のプログラム片が共有された変数にアクセスする場合に、それらのアクセスが干渉するかどうかを検査することは、プログラムの安全性や性能を向上させるために重要である。並列実行されるプログラムが非決定的な結果を導くことをデータ競合と呼ぶ。データ競合を防ぐためには、並列実行される複数のプログラム片が非干渉でなければならない。加えて、ある複数のプログラム片が非干渉であるという情報は、コンパイラが命令の並べ替えやキャッシュの利用などの最適化を行う際にも役立つ。

複数のプログラム片による共有変数に対するアクセスが干渉するかを検査する際には、それぞれのプログラム片が共有変数に対して行う操作の種類（読み取りまたは書き込み）を考慮する必要がある。読み取り操作は他の読み取り操作と干渉しないが、書き込み操作は他のあらゆる操作と干渉する。そのため、読み取り専用のアクセスを提供する参照を複製すること（Aliasing）と、書き込み可能なアクセスを唯一の参照に制限すること（Mutability）は排他的な関係にある。このような関係性を保つ規則は、Aliasing XOR Mutability の規則として知られている。

複数のプログラム片が干渉するかどうかを検査する手法はいくつか存在する。その一つは、それぞれのプログラム片の作用を検査する手法である。作用とはプログラム片が共有変数に対して行う読み取りや書き込みの操作の集合であり、それぞれのプログラム片の作用が含む操作が互いに干渉しないことを検査することで、プログラム片同士が非干渉であるかどうかを検査する。例えば、図 2.1 に示すプログラムの干渉を検査することを考える。図 2.1a の作用は、変数 x の読み取りと変数 y への書き込みであり、図 2.1b の作用は、変数 x の読み取りと変数 z への書き込みである。

これらの操作が干渉するかどうかを検査するためには、“MayEqual”問題と呼ばれる課題を解決する必要がある [2]。“MayEqual”問題とは、ある変数が指し示す可能性のあるメモリ上の領域の集合が、他の変数が指し示す可能性のあるメモリ上の領域の集合と交差するかどうかを判断する課題である。先程の図 2.1 の例では、図 2.1a と図 2.1b の作用の双方に含まれる変数 x の読み取り操作同士が干渉しないことは明らかである。一方、他の操作の組み合わせが干

<pre> 1 ... 2 print(*x); 3 *y = 42; 4 ... </pre>	<pre> 1 ... 2 print(*x); 3 *z = 10; 4 ... </pre>
(a) x を読み取り y に書き込むプログラム片	(b) x を読み取り z に書き込むプログラム片

図 2.1: 干渉を検査されるプログラムの例

干渉するかどうかを判断するためには、それぞれの変数が同じメモリ領域を指し示す可能性があるかどうかを判断する必要がある。例えば、変数 y と変数 z が同じメモリ領域を指し示しているとき、図 2.1a の書き込み操作と図 2.1b の書き込み操作は干渉する。

“MayEqual”問題を解決するためには、他の検査手法を組み合わせる必要がある。型検査を用いる手法では、型に追加の情報を付与した上で、異なる型情報を持つ変数が同じメモリ領域を指し示すことはないという前提に基づいて干渉を検査することができる。別の手法として、変数が指し示す可能性があるメモリ領域を追跡する特別な型システムを用いる、Points-to Analysis と呼ばれる手法がある [3]。しかしながら、いずれの手法においても、本来許容されるべき非干渉なプログラム片が誤って干渉すると判断され、プログラミング言語の表現力が制限される可能性がある。

このような問題を解決するための手法として、権限を用いる手法がある。それぞれの権限は特定のメモリ領域に対する書き込みアクセスを許可する資源であり、複数の権限が同時に存在することは即ち、それらの権限が指し示すメモリ領域が異なることを意味する。権限はメモリ割り当てによってのみ生成され、メモリ解放によってのみ破棄される。また、権限は線形の資源として扱い、権限を複製したり、メモリ解放を伴わずに破棄したりすることはできない。例えば、図 2.1 の例における書き込み操作を許可するためには、変数 y と変数 z のそれぞれに対応する権限が一つずつ必要である。もしも変数 y と変数 z が同じメモリ領域を指し示している場合、そのメモリ領域に対応する権限は一つしか存在し得ないため、どちらかの書き込み操作は許可されない。

線形の権限を用いる手法における課題は、互いに干渉しない複数の読み取り操作を表現できないことである。あるメモリ領域に対応する権限は一つしか存在し得ないため、同じメモリ領域を読み取る複数のプログラム片の双方に対応する権限を与えることができない。プログラムが扱うメモリ領域を排他的に分割して干渉を検査する分離論理においても、複数の読み取り操作が干渉しないことを表現することができない [4]。

この問題を解決するために、プログラムの特定の部分においてのみ、一時的に書き込み権限を読み取り権限に変換する手法が存在する。この手法では、書き込み権限から変換された読み取り権限を非線形に扱い、複製や破棄を許容することで読み取り同士が干渉しないことを表現する。例えば、図 2.1 の例では、これらのプログラムが実行される前に、変数 x に対応する書

6 第2章 Rust と漸進的型付け

書き込み権限を読み取り権限に変換し、その読み取り権限を複製してそれぞれのプログラム片に与えることで、双方のプログラム片が変数 x を読み取ることが可能になる。このとき、図 2.1 のプログラムの実行後には、再び変数 x に対応する書き込み権限を復元することができる。

非線形の読み取り権限を導入する方法にも、プログラミング言語の表現力を制限する問題がある。クロージャを持つ言語の場合、クロージャの内部に読み取り権限をキャプチャした状態で書き込み権限を復元することを防ぐため、クロージャはあらゆる権限をキャプチャしないという制限を課す必要がある。

このような課題を解決し、複雑な静的検査を組み合わせた、プログラミング言語の表現力を制限したりすることなくプログラムが非干渉であることを検査する手法が、Fractional Permissions である [2]。Fractional Permissions では、読み取り権限を複製可能にする代わりに分割可能にすることによって、非線形性を導入することなく読み取り操作同士が干渉しないことを表現する。Fractional Permissions における権限は、操作の対象となるメモリ領域を示す識別子の他に、権限の大きさを表す分数値を持つ。分数値は 0 より大きく 1 以下の値を取り、分数値が 1 である権限は書き込み権限を表し、分数値が 1 に満たない権限は読み取り権限を表す。

書き込み権限を複数の読み取り権限に変換するためには、権限を分割する。権限 π の分割は次の式のように表現され、権限は何度でも繰り返し分割することができる。

$$\pi \equiv \epsilon\pi, (1 - \epsilon)\pi \quad (0 < \epsilon < 1) \quad (2.1)$$

また、一度分割した書き込み権限を復元するためには、分割した読み取り権限の全てを回収しなければならない。これによって、複数の読み取り操作が非干渉でありながら、書き込み操作が他の操作と干渉することを自然に表現できる。

2.2 Rust と借用検査

Rust は、特に低レベルなシステムソフトウェア開発に適したプログラミング言語として近年注目を集めている。C 言語を始めとする従来のシステムプログラミング言語は、プログラマによる低レベルなメモリ管理を可能にする代わりに、誤ったメモリ操作に起因するデータ競合やセキュリティ上の問題を引き起こすという問題を抱えていた。Rust はプログラマに高レベルな抽象化と低レベルなリソースの管理の双方を提供し、安全性と実行性能の両立を実現する [5]。

Rust は、メモリ安全性を保証するために、Aliasing XOR Mutability の規則に基づいて干渉がないことを検査する。干渉を検査するにあたって Rust は所有権 (Ownership) と借用 (Borrowing) の概念を採用しており、この静的検査は借用検査 (Borrow Checking) と呼ばれている。所有権は、あるメモリ領域に対して読み取り及び書き込みのアクセスが許される唯一の変数に与えられる。プログラム中で変数の間を所有権が移動することは許可されるが、同じメモリ領域を指し示す所有権が複数の変数に同時に存在することは許されない。例えば、図 2.2 に示す Rust のプログラムでは、行 2 において文字列 "hello" を指し示す所有権が変数

```

1 let s1 = String::from("hello");
2 let mut s2 = s1;
3 s2.push_str(", world!");
4 println!("{}", s1); // error: borrow of moved value

```

図 2.2: Rust における所有権の移動の例

```

1 let mut v = vec![7, 2, 4];      1 let mut v = vec![7, 2, 4];
2 sort(v);                       2 sort(&mut v);
3 println!("{:?}", v);          3 println!("{:?}", v);

```

(a) 借用を用いない関数呼び出しの例

(b) 借用を用いる関数呼び出しの例

図 2.3: Rust の関数呼び出しにおける借用の使用

s1 から変数 s2 に移動している。そのため、行 3 における変数 s2 への書き込み操作は許可される。一方、変数 s1 はもはや文字列"hello"を指し示す所有権を持たないため、行 4 における変数 s1 の読み取り操作は許可されない。

借用とは、あるメモリ領域に対するアクセス許可を、所有権を持つ変数から一時的に借り受けるための仕組みである。借用を行うことで、所有権を移動することなく一時的にメモリ領域にアクセスするための、参照を生成することができる。Rust には、許容されるアクセスの種類が異なる、二種類の参照が存在する。可変参照 (Mutable References) は `&mut T` という型で表され、参照を通じた読み取りと書き込みの双方の操作を許容する。一方、共有参照 (Shared References) は `&shr T` という型で表され^{*1}、参照を通じた読み取り操作のみを許容する。

一時的な参照を生成する借用の操作は、特に関数を呼び出す際に有用である。借用を行わない場合、ある変数を引数として関数を呼び出すためにはその所有権を関数に移さなければならず、その変数は元のスコープにおいてはもはや使用することができなくなる。しかし、借用を用いることで、関数が必要とする一定の期間に限ってそのメモリ領域へのアクセスを許可し、操作が終了した後は元の変数を引き続き使用することが可能となる。例えば、図 2.3 に示す Rust のプログラムでは、変数 `v` に格納された配列をソートするために、`sort` 関数を用いている。もしも `sort` 関数が配列の所有権を引数として受け取る場合、図 2.3a に示すように `sort` 関数を呼び出すこととなる。このとき、行 2 の関数呼び出しによって、配列の所有権は `v` から `sort` 関数の内部に移動し、変数 `v` はもはや使用できなくなる。そのため、行 3 における変数 `v` の読み取り操作は許可されない。一方、`sort` 関数が配列への可変参照を引数として受け取る場合、図 2.3b の行 2 に示すように、変数 `v` への可変参照を取得して `sort` 関数を呼び出す

^{*1} Rust においては、共有参照は暗黙的に `&T` と表記されるが、本稿ではわかりやすさのために `shr` というキーワードを明記することとする

8 第2章 Rust と漸進的型付け

```
1 let mut x = 10;
2 let r1 = &shr x;
3 let r2 = &mut x;
4 *r2 += 5;
5 println!("{}", r1);
```

図 2.4: Rust におけるライフタイム重複の例

```
1 { let mut v = vec![7, 2];      1 { let mut v = vec![7, 2];
2   let r1 = &mut v;           2   { let r1 = &mut v;
3   sort(r1);                 3     sort(r1); }
4   println!("{:?}", v); }    4   println!("{:?}", v); }
```

(a) 実行できないプログラムの例

(b) 実行可能なプログラムの例

図 2.5: Rust における Lexical Lifetimes の振る舞い

ことができる。このとき、配列の所有権を変数 `v` から移動することなく、関数 `sort` による配列への一時的なアクセスが許可される。したがって、行 3 における変数 `v` の読み取り操作は許可され、ソートされた配列が出力される。

参照は一つの変数に対して複数生成することができるが、同じ変数を指し示す複数の参照のライフタイムが重複する場合には、それらの参照には `Aliasing XOR Mutability` の規則が適用される。参照のライフタイムとは、少なくとも、その参照が生成される時点と、その参照が最後に使用される時点を含む期間であり、その参照を通じたアクセスが許可される期間と概ね一致する。例えば、図 2.4 に示す Rust のプログラムでは、参照 `r1` のライフタイムは少なくとも行 2 から行 5 までの期間を含み、参照 `r2` のライフタイムは少なくとも行 3 から行 4 までの期間を含む。

ある参照を通じて変数にアクセスを行う際、同じ変数を指し示し、そのアクセスの瞬間が自身のライフタイムに含まれる参照が複数ある場合には、`Aliasing XOR Mutability` の規則に基づいて、次の規則が適用される。該当する複数の参照のうち少なくとも一つが共有参照である場合、それらの参照を通じた書き込み操作は許可されない。該当する複数の参照がすべて可変参照である場合、最も短いライフタイムを持つ参照のみがアクセスを許可され、それ以外の参照を通じたすべての操作は許可されない。このとき、最も短いライフタイムを決定できないような可変参照を通じた操作も許可されない。図 2.4 に示す Rust のプログラムでは、行 4 における参照 `r2` を通じた書き込み操作の瞬間に、同じ変数 `x` を指し示す参照 `r1` と参照 `r2` のライフタイムが重複している。このとき、参照 `r1` が共有参照であるため、参照 `r1` 及び参照 `r2` を通じた書き込み操作は許可されない。したがって、このプログラムはエラーとなる。

参照のライフタイムの定め方には、処理系の実装に応じて複数の方法が存在する。どのようなライフタイムの定め方を用いてもメモリ安全性の保証は維持されるが、ライフタイムを広く定めるほど解析が単純になる反面、許容されるプログラムの表現力が制限される。対して、ライフタイムを厳格に狭く定めるほど、多くのプログラムが検査を通過するようになる反面、ライフタイムの解析が困難になる。かつての Rust の処理系では、ブロックにおいて囲まれたスコープをライフタイムの期間として扱う、Lexical Lifetimes と呼ばれる手法を採用していた。しかしながらこの手法では、検査を通過するために明示的にスコープを分割する必要があるなど、非直感的な記述を強いられる場合があった。例えば、図 2.5a のプログラムを Lexical Lifetimes に基づいて検査する場合を考える。参照 `r1` のライフタイムは少なくとも生成される行 2 から使用される行 3 までの期間を含まなければならない。そのため、最も近いスコープである行 1 から行 4 までが参照 `r1` のライフタイムとなる。この間、変数 `v` への可変参照である参照 `r1` が存在しているため、内部的に共有参照を必要とする `println!` の呼び出しは許可されない。このプログラムを実行可能にするためには、図 2.5b に示すように、明示的に行 2 から行 3 までを含む新しいスコープを作成し、参照 `r1` のライフタイムをそのスコープ内に限定する必要がある。

このような課題を解決するため、近年の Rust 処理系では Non-Lexical Lifetimes と呼ばれるライフタイムの定め方を採用している [6]。Non-Lexical Lifetimes では、参照が生成される文からその参照が最後に使用される文までの期間を静的に解析してライフタイムを定める。Lexical Lifetimes では許容されなかった図 2.5a のプログラムを Non-Lexical Lifetimes に基づいて検査する場合、参照 `r1` のライフタイムは行 2 から行 3 までの期間と定められる。そのため、図 2.5b のように明示的にスコープを分割することなく、プログラムを実行することが可能になる。

Rust の借用検査は、このように所有権の移動と借用のライフタイムを検査することで、並行プログラミングにおけるデータ競合を防止し、安全なメモリ操作を保証している。加えて、プログラムが非干渉であることが検査されることを利用して、コンパイラによる最適化を促し、高い実行性能を実現している。

借用検査によって Rust が提供するメモリ安全性の保証について、形式化を行ったり、その安全性を証明する試みがいくつか存在する。RustBelt[5] は、分離論理 [4] に基づく並列計算のための証明フレームワークである Iris[7] を用いて、Rust のサブセットである λ_{Rust} をモデル化する。 λ_{Rust} は、Rust 及び Rust を用いて書かれたライブラリのメモリ安全性を証明するために利用できる。RustSEM[8] は実行可能な Rust の形式的モデルであり、より多くの Rust プログラムの安全性を証明することができる。Stacked Borrows[9] 及び Tree Borrows[10] は、Rust プログラムが未定義動作を引き起こさないことを検証するインタプリタである Miri^{*2}において用いられる借用モデルである。線形型システムの拡張によって Rust の借用をモデル化し、その停止性とメモリ安全性を保証する試みも存在する [11]。

^{*2} <https://github.com/rust-lang/miri>

2.3 漸進的型付け

漸進的型付け (Gradual Typing) とは、静的型付けと動的型付けの双方の利点を活かすため、型注釈が部分的に省略されたプログラムを実行可能にする手法である [1]。静的型付けは、プログラムに型注釈を記述することで、実行時に型エラーが発生して意図しない動作を引き起こすことを防ぐと同時に、実行効率を向上させるための最適化を行うことができる。一方、動的型付けでは、型注釈を記述することなく迅速にプロトタイピングを行うことが可能である。漸進的型付けを採用したプログラミング言語では、単一の言語の中で、静的に型付けされた部分と動的に型付けされた部分をプログラマが選択的に組み合わせることができる。

漸進的型付けの目的は、動的型付けから静的型付けへの段階的な移行を可能にすることである。動的型付けと同様に迅速なプロトタイピングを行った後に、プログラムの重要な部分に型注釈を追加することで、型安全性と実行効率を向上させることができる。この目的を達成するために、漸進的型付けの言語が満たすべき性質を形式的に定義したものが、Criteria for Gradual Typing である [1]。Criteria for Gradual Typing では、型注釈の有無のみが異なる二つのプログラムがどのように振る舞うべきかを定義する。

プログラムの振る舞いを議論するにあたっては、次の三つの種類のエラーを区別する。

Type Error	プログラムが静的な型検査に失敗する
Trapped Error	プログラムが実行時エラーを発生させ、言語仕様に定められた方法で停止する
Untrapped Error	プログラムが実行時エラーを発生させるが、その振る舞いが言語仕様に定められていない

静的型付け及び動的型付けの言語は、いずれも強力に型付けされており、そのため Untrapped Error は発生しない。加えて、静的型付けの言語で型検査を通過したプログラムは、Trapped Error も発生させない。

Criteria for Gradual Typing においては、漸進的型付けのプログラミング言語は、静的型付けと動的型付けの双方の言語を包含していなければならない。まず、漸進的型付けの言語は強力に型付けされているため、Untrapped Error は発生しない。加えて、漸進的型付けにおいて全ての型が静的に記述されたプログラムは、静的型付けの言語における同一のプログラムと同様に振る舞わなければならない。このようなプログラムのことを完全に注釈されたプログラムと呼び、漸進的型付けにおいては完全に注釈されたプログラムは Trapped Error を発生させない。例えば、図 2.6a に示すプログラムは、漸進的型付けの言語における完全に注釈されたプログラムである。静的型付け言語においてこのプログラムは、行 1 における関数 `max` の定義に対して、行 5 の関数呼び出しの第一引数の型が合致しないため、型検査に失敗し、Type Error を発生させる。よって、漸進的型付けの言語においても同様の理由で Type Error を発生させる。

同様に、漸進的型付けにおいて型注釈を全く含まないプログラムは、動的型付けの言語にお

```

1 fn max(a: int, b: int) -> int {
2   if (a > b) { a } else { b }
3 }
4 let x: string = "hello";
5 let m: int = max(x, 10);

```

(a) 完全に注釈されたプログラムの例

```

1 fn max(a: int, b: int) -> int {
2   if (a > b) { a } else { b }
3 }
4 let x: * = "hello";
5 let m: * = max(x, 10);

```

(b) 型注釈が部分的に省略されたプログラムの例

```

1 fn max(a: *, b: *) -> * {
2   if (a > b) { a } else { b }
3 }
4 let x: * = "hello";
5 let m: * = max(x, 10);

```

(c) 型注釈を含まないプログラムの例

図 2.6: 型注釈の有無が異なる漸進的型付けのプログラム

ける同一のプログラムと同様に振る舞わなければならない。動的型付けの言語では、特定の型の値のみに適用可能な操作を実行する前に、与えられた値が期待する型であることを確認するための動的な型検査が行われる。例えば、図 2.6c に示すプログラムは、漸進的型付けの言語における型注釈を含まないプログラムである。この言語では、型注釈の省略を表す特別な型として `*` 型を明記することとする。このプログラムは、図 2.6a に示す完全に注釈されたプログラムと、型注釈の有無を除いて完全に同一である。しかしながら、動的型付けの言語では実行前に型検査を行わないため、このプログラムは実行することができる。この言語は文字列型に対する比較操作を定義していないため、行 2 における比較操作の時点で、動的な型検査に失敗し、Trapped Error を発生させる。よって、漸進的型付けの言語においても同様の理由で Trapped Error を発生させる。

型注釈が部分的に省略されたプログラムについては、プログラムのうち型注釈が記述された静的な部分と、型注釈が省略された動的な部分について、異なる振る舞いをするのが期待

12 第2章 Rust と漸進的型付け

```
1 fn apply(f: (float -> float), x: float) -> float {
2     f(x)
3 }
4 fn error(x: *) -> * {
5     "error"
6 }
7 apply(error, 3.5);
```

図 2.7: 漸進的型付けにおいて関数を受け取る関数の例

される。型注釈が記述された静的な部分については、実行時に Trapped Error と Untrapped Error の双方を発生させないことが期待される。そのため、記述された型注釈の間で型が整合しているかどうかを、静的な型検査によって確認する。一方、型注釈が省略された動的な部分については、実行時の型検査によって Untrapped Error を発生させないことが期待される。この振る舞いを実現するため、動的な部分から静的な部分へと実行が移る際には、静的な部分が期待する型に一致することを確認するための動的な型検査を実行する。例えば、図 2.6b に示すプログラムは、行 1 における関数 `max` の定義は静的に型付けされている一方で、行 4 及び行 5 における変数宣言と関数呼び出しは型注釈が省略されているため、動的に型付けされている。このプログラムの実行にあたっては、まず静的な部分に対して型検査が行われるが、関数 `max` の定義は型検査が成功する。その後、行 5 における関数呼び出しの際に、`*` 型の引数 `x` が整数型であることを確認するための動的な型検査が行われる。変数 `x` には文字列型の値が格納されているため、この型検査は失敗し、Trapped Error を発生させる。

漸進的型付けにおいて関数型を扱う場合には、関数型の値が期待する型に一致することを確認することは困難である。任意の関数がどのような型の引数を受け取り、どのような型の戻り値を返すかを解析することは、停止性問題に帰着されるためである。そこで、関数型の値の型検査はその関数が呼び出される時点まで遅延し、関数が呼び出される際に、引数の型と戻り値の型が期待する型に一致することをそれぞれ確認する。このとき、プログラムの静的に型付けされた部分で実行時エラーが発生しないことを保証するために、実行時エラーが生じる原因となった関数呼び出しまで遡ってエラーを報告する必要がある。この振る舞いを実現するために、Blame Tracking[12] と呼ばれる手法が存在する。例えば、図 2.7 に示すプログラムを考える。関数 `apply` は、引数として `float -> float` 型の関数を受け取るように静的に型付けされている。一方、関数 `error` は型注釈が省略されているが、任意の型の引数を受け取って文字列型の値を返す関数である。この関数 `error` は、行 7 において `apply` 関数の引数として渡されている。関数 `error` は関数 `apply` が期待する型に一致しないため、行 7 における関数呼び出しの時点で実行時エラーが発生することが期待されるが、この時点では関数 `error` の型が一致しないことは検出できない。代わりに、行 2 における関数 `f` の呼び出しの際に、引数と戻り値の型検査が行われる。この例では、引数の型検査は成功するが、戻り値の型が `float`

```
1 fn id(x: *) -> * { x }
2 id(42);
```

(a) 元となる型注釈を含まないプログラム

```
1 fn id(x: int) -> * { x }
2 id(42);
```

(b) 引数に正しく追加したプログラム

```
1 fn id(x: *) -> int { x }
2 id(42);
```

(c) 戻り値に正しく追加したプログラム

```
1 fn id(x: int) -> int { x }
2 id(42);
```

(d) 引数と戻り値に正しく追加したプログラム

```
1 fn id(x: *) -> str { x }
2 id(42);
```

(e) 戻り値に誤って追加したプログラム

```
1 fn id(x: str) -> str { x }
2 id(42);
```

(f) 引数と戻り値に誤って追加したプログラム

図 2.8: 漸進的型付けにおける型注釈の変化の例

型に一致しないため、実行時エラーが発生する。Blame Tracking では、この実行時エラーの原因が行 7 における関数呼び出しにあることを特定し、その箇所でエラーを報告する。

漸進的型付けが動的型付けのプログラムから静的型付けのプログラムへの段階的な移行を支援するためには、型注釈の追加や削除によって意図せずプログラムの振る舞いに変化しないことが重要である。Gradual Guarantee[1] は、型注釈の有無のみを変更したプログラムがどのように振る舞いを変化させるかを形式的に定義する。正しく型付けされたプログラムに型注釈を追加する場合には、追加した型注釈が正しいものである限り、プログラムは正しく型付けされ、同じように振る舞う。例えば、図 2.8 に示すようにプログラムに型注釈を追加する場合を考える。元となるプログラムは、図 2.8a に示す型注釈を全く含まないプログラムである。これに対して、図 2.8b に示すように、引数に整数型の型注釈を追加した場合を考える。この場合、追加した型注釈は正しいものであるため、プログラムは正しく型付けされ、図 2.8a と同じように振る舞う。加えて、図 2.8d に示すように、さらに戻り値に整数型の型注釈を追加した場合にも、型注釈が正しいものであるため、同様に振る舞う。

一方で、誤った型注釈を追加した場合には、静的に型検査に失敗したり、実行時に Trapped Error を生じさせたりするなど、プログラムの振る舞いに変化する場合がある。例えば、図 2.8e に示すように、戻り値に文字列型の型注釈を追加した場合を考える。この場合、追加した型注釈は誤ったものであるが、これは実際に関数 id を評価するまで判明しない。そのため、プログラムは実行することができるが、実行時に Trapped Error を発生させる。さらに、図 2.8f に示すように、引数と戻り値の両方に文字列型の型注釈を追加した場合を考える。この場合、引数に追加された誤った型注釈が、行 2 における関数呼び出しの引数と整合しないことが静的に検出されるため、型検査に失敗し、Type Error を発生させる。

型注釈を削除する場合については、より強力な保証を与えることができる。正しく型付けさ

14 第2章 Rust と漸進的型付け

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

図 2.9: Rust における可変性の指定の例

れたプログラムから型注釈を削除した場合、プログラムは引き続き正しく型付けされ、同じように振る舞う。例えば、図 2.8d に示すプログラムは正しく型付けされている。そのため、引数と戻り値の型注釈をそれぞれ削除した図 2.8b 及び図 2.8c に示すプログラムも引き続き正しく型付けされ、同じように振る舞うことが保証される。両方の型注釈を削除した図 2.8a に示すプログラムについても同様である。

2.4 Rust への移行における可変性の指定の難しさ

Rust プログラムにおいて参照を用いる場合には、参照ごとに可変性 (Mutability) を明示的に指定する必要がある。可変性とは、参照を通じて指し示される値を変更できるかどうかを表す性質であり、参照型の宣言や参照の生成において必ず記述しなければならない。可変性を指定するキーワードは `shr` と `mut` のいずれかであり、それぞれ共有参照と可変参照に対応する。例えば、図 2.9 に示す Rust のプログラムでは、行 2 において、関数 `rename` の引数として宣言されている参照 `user` は可変参照であり、参照 `new_name` は共有参照である。これらの参照の型を宣言するために、それぞれ `mut` と `shr` のキーワードが用いられている。また、行 8 において関数 `rename` を呼び出す際に、変数から参照を生成するための演算子に、`mut` と `shr` のキーワードが用いられている。

一般に広く用いられているプログラミング言語には、Rust の可変性に相当する概念が存在しない。C 言語の `const` 修飾子や Java の `final` 修飾子など、変数や参照が指し示す値の変更を防止するための手段は存在するが、これらは主にプログラムの高速化や保守性の向上などの目的で用いられ、省略してもプログラムの結果に影響を与えない。例えば、図 2.10 に示す C 言語のプログラムでは、関数 `rename` の引数のうち `new_name` に対して `const` 修飾子が付与されている。この修飾子は、関数 `rename` の内部で引数 `new_name` を通じて指し示される値が変更されないことを示しているが、この修飾子を省略したとしても、プログラムの実行結果

```

1 typedef struct { const char* name; } User;
2 void rename(User* user, const char* new_name) {
3     user->name = new_name;
4 }

```

図 2.10: C 言語における const 修飾子の例

<pre> 1 void swap(int* a, 2 int* b) { 3 int temp = *a; 4 *a = *b; 5 *b = temp; 6 } 7 void main() { 8 int v[2] = {4, 2}; 9 swap(&v[0], 10 &v[1]); 11 } </pre>	<pre> 1 fn swap(a: &mut i32, 2 b: &mut i32) { 3 let temp = *a; 4 *a = *b; 5 *b = temp; 6 } 7 fn main() { 8 let mut v = vec![4, 2]; 9 swap(&mut v[0], 10 &mut v[1]); 11 } </pre>
---	--

(a) C 言語のプログラム

(b) Rust のプログラム

図 2.11: Rust への移行が困難になる C 言語のプログラムの例

に影響はない。

一方で、Rust においては、可変性の異なる二種類の参照はそれぞれ許容される操作や借用検査によって課される制約が異なり、すべての参照について適切な可変性を選択しなければ、プログラムを実行することはできない。そのため、どちらかの可変性を既定のものとして選択することができない。例えば、図 2.9 に示す Rust のプログラムにおいて、関数 `rename` の引数 `user` として共有参照を指定した場合、コンパイルエラーが発生する。引数 `new_name` として可変参照を指定した場合には、この例においてはコンパイルエラーは発生しないが、一般に共有参照を可変参照に置き換えることができるとは限らない。

よって、既存のソフトウェアを Rust へ移行する際には、記法の異なる言語を翻訳するだけでなく、参照の可変性を検討するという追加の作業が必要になる。この作業は、仮に移行元のプログラムに `const` 修飾子などが適切に付与されていたとしても、回避することができない。ましてや、`const` 修飾子などが適切に付与されていない場合には、作業はさらに困難となる。例えば、図 2.11a に示す C 言語のプログラムを Rust に移行する場合を考える。このプログラ

16 第2章 Rust と漸進的型付け

```
1 void rename(User* user, char* new_name) {
2     user->name = new_name;
3 }
```

(a) 移行元となる C 言語のプログラム

```
1 fn rename(user: &unk User, new_name: &unk str) {
2     user.name = new_name.to_string();
3 }
```

(b) 移行後の Rust のプログラム

図 2.12: 可変性の指定を省略した Rust への移行の例

ムにはこれ以上 `const` 修飾子を記述することができる箇所が存在しないため、適切に `const` 修飾子が付与されているとすることができる。そこで、`const` 修飾子を持たないポインタを可変参照に対応させることとし、Rust に移植したプログラムを図 2.11b に示す。この Rust プログラムは、行 9 における関数 `swap` の呼び出しにおいて、同じ変数に対する可変参照を二度借用しようとしているため、コンパイルエラーが発生する。このような問題を解決するためには Rust の借用検査の振る舞いを正しく理解しなければならず、新しく Rust を導入しようとするプログラマにとって大きな障壁となる。

そこで本論文では、Rust プログラム中における参照の可変性の指定を省略したまま、プログラムを実行可能にする手法を提案する。この手法を用いることによって、C 言語をはじめとする従来のプログラミング言語から Rust への移行において、可変性を検討する作業を回避することが可能になる。本稿では、可変性の指定を省略する際には、可変性を指定するためのキーワードである `mut` 及び `shr` の代わりに `unk` というキーワードを明記することとする。例えば、図 2.12a に示す C 言語のプログラムを Rust に移行する場合を考える。このプログラムには適切に `const` 修飾子が記述されていないため、参照型を利用するためには適当な可変性を選択しなければならない。ここで、提案手法を用いて可変性の指定を省略した Rust プログラムを図 2.12b に示す。このプログラムでは、行 1 において参照型の引数 `user` 及び `new_name` を宣言する際に、その可変性が省略されている。これによって、移行元となる C 言語のプログラムには記述されていない、可変性という新しい概念を検討する必要を回避することができる。

可変性の指定を省略した場合においても、Rust に移行したプログラムを実行する限りにおいては、Rust が保証するメモリ安全性が維持されることが期待される。そのため、漸進的型付けの考え方に倣い、静的な借用検査と実行時検査を組み合わせることによって、実行されるプログラムが意図しない振る舞いをしないことを保証する。可変性が静的に記述された部分については、従来の借用検査によって静的に検査を行い、Aliasing XOR Mutability の規則に違反する場合には実行前に静的にエラーを発生させる。可変性の指定が省略された部分について

は、実行時に追加の検査を行うことで Aliasing XOR Mutability の規則に違反しないことを検査し、違反が検出された場合には実行時エラーを発生させる。

Rust コンパイラは、参照に指定された可変性の情報を利用してプログラムの最適化を行う。参照の可変性の指定を省略した場合については、実際にプログラムが実行されるまで可変性を決定することができない。そのため、可変性の指定を省略した参照に対しては、不適切な最適化が行われないようにする。最適化が制限されるため、参照の可変性の指定を省略したプログラムは性能が低下する可能性がある。

提案手法は、漸進的型付けと同様に、参照の可変性の指定を段階的に追加し、静的な Rust プログラムへと進化させることを支援する。そのため、参照の可変性を省略して Rust プログラムを記述した場合においても、特にプログラムの重要な部分については、追って可変性を検討し、静的に記述することが望ましい。可変性を静的に記述することによって、実行時エラーが発生しないという強力な保証が得られるようになる。加えて、実行時検査が不要となり、プログラムの性能が向上することが期待される。

第 3 章

漸進的借用検査

3.1 漸進的借用検査による可変性の指定の省略

漸進的借用検査 (Gradual Borrow Checking) は、参照の可変性の指定が部分的に省略されたプログラムを、メモリ安全性を保証しながら実行することを可能にする手法である。Rust と同等のメモリ安全性を保証するためには、同じ変数を指し示す複数の参照の可変性が Aliasing XOR Mutability の規則を満たしている必要がある。漸進的借用検査は、参照の可変性の指定が省略された部分を含むプログラムについても、実行時検査によって Aliasing XOR Mutability の規則を満たすことを保証する。

可変性の指定が省略された参照が Aliasing XOR Mutability の規則を満たすことを検査するためには、参照のライフタイムが開始する時点と終了する時点で実行時の検査を行う必要がある。参照のライフタイムが開始するときには、新しく生成されようとする参照が既存の参照と競合しないことを確認すると同時に、既存の参照を通じて許容される操作を必要に応じて制限する。参照のライフタイムが終了するときには、その参照によって制限されていた他の参照の操作を再び許可する。

漸進的借用検査による Aliasing XOR Mutability の規則の検査は、静的な借用検査と実行時検査を組み合わせることで実現される。静的な借用検査は、可変性が静的に指定された参照の間で Aliasing XOR Mutability の規則が満たされていることを検査するほか、可変性の指定が省略された参照についても、主にそのライフタイムを解析する目的で使用される。実行時検査は、プログラムの実行中に参照を通じて行われた操作に基づいて、可変性の指定が省略された参照の間で Aliasing XOR Mutability の規則が満たされていることを検査する。

3.2 実行時検査のアルゴリズム

3.2.1 可変性の指定が省略された参照の権限の範囲

漸進的借用検査における実行時検査器は、可変性の指定が省略された参照に対して、現在それぞれの参照が持ちうる権限の範囲を計算する。権限の範囲は「権限の下限」と「権限の上限」の組によって表現され、その計算は、同じ変数を指し示しているいずれかの参照の、ライフタ

<pre>1 let mut x = 10; 2 let r1 = &unk x; 3</pre>	<pre>1 let mut x = 10; 2 let r1 = &mut x; 3 let r2 = &unk *r1;</pre>
(a) 変数の借用による参照の生成	(b) 参照の再借用による参照の生成

図 3.1: Rust における参照の生成

<pre>1 let r1 = &unk 10; 2 *r1 = 20; 3</pre>	<pre>1 let r1 = &unk 10; 2 let r2 = &mut *r1; 3 *r2 = 20;</pre>
(a) 参照を直接使用した書き込み	(b) 可変参照の再借用を経由した書き込み

図 3.2: 可変性の指定が省略された参照を通じた操作

タイムが開始する時点と終了する時点で行われる。ライフタイムの開始は、再借用によって新しい参照が生成される時点とする。再借用とは、既存の参照を複製して新しい参照を生成する操作であり、参照を変数に代入したり、参照を用いて関数を呼び出したりする際に行われる。このとき、参照が指し示す変数を変更することはできないが、参照の可変性は変更することができる。例えば、図 3.1b では、再借用の演算子 `&mut *` を用いて可変参照 `r1` を再借用して、可変性の指定が省略された参照 `r2` を生成している。

参照を新しく生成する操作には、再借用のほかに、変数から新しく参照を生成する借用があるが、簡単のため、可変性の指定が省略された参照の生成は再借用のみによって行われると仮定する。可変性の指定を省略した参照を変数から直接生成したい場合には、一度可変性が静的に指定された参照を生成し、その参照を再借用することができるため、この仮定はプログラムの表現力を制限しない。例えば、図 3.1a に示すような借用による参照の生成は、図 3.1b に示すような再借用による参照の生成と等価である。

加えて、可変性の指定が省略された参照を通じて、直接値を読み書きする操作は行われないと仮定する。可変性の指定が省略された参照を通じて値を読み書きする場合には、一度可変性が静的に指定された参照を生成し、その参照を通じて値を読み書きすることができるため、この仮定もプログラムの表現力を制限しない。例えば、図 3.2a に示すような可変性の指定が省略された参照を直接使用した書き込み操作は、図 3.2b に示すような可変参照の再借用を経由した書き込み操作と等価である。これらの仮定により、漸進的借用検査は、可変性の指定が省略された参照の再借用のみに注目して検査を行うことができる。

権限は、Fractional Permissions[2] における分数値の表現に類似する概念であり、直観的にはその参照を通じて許容される操作を示す。権限は三値で表現され、それぞれ次のように表記される。

20 第3章 漸進的借用検査

0 参照を通じて値を読み書きすることはできない。

ϵ 参照を通じて値を読み取ることができるが、書き換えることはできない。

1 参照を通じて値を読み書きすることができる。

ϵ は 0 より大きく 1 より小さい微小の値を表す。したがって、権限には大小関係があり、 $0 < \epsilon < 1$ が成り立つ。また、それぞれの権限に許容される操作から、権限と可変性の間に表 3.1 のような対応関係を考えることができる。権限には加減算が定義されており、その結果は表 3.2 に示す通りである。特に注意が必要な演算については赤字で示している。

権限の下限は、現時点までにその参照が親の参照から借り受けた権限の大きさを表す値である。この値は、プログラム中に可変性の指定を省略せず静的に記述した場合に、その参照のために選択しなければならない可変性を決定するために使用される。そのため、権限の下限は、その参照を通じて読み書きの操作が行われる際に増加し、参照が生成されてからライフタイムが終了するまでの間に使用した最大の権限を記憶する。したがって、あるプログラムの実行中に計算された権限の下限の値に基づいて、プログラムに対応する可変性を静的に記述すると、Aliasing XOR Mutability の規則を満たすことが期待される。ただし、権限の下限はプログラム中でそれまでに実行された操作に基づいて決定されるため、実行中に通過したプログラムの部分に限っては規則を満たすことが保証されるが、プログラム全体にわたって静的に規則を満たすことは保証されない。

権限の下限は、可変性の指定が省略された参照が再借用によって新しく生成される時、0 に初期化される。本手法では、可変性の指定が省略された参照を通じて読み書きの操作を行うためには、可変性が静的に指定された参照への再借用を経由しなければならない。そのため、

表 3.1: 権限と可変性の対応関係

権限	可変性	読み取り	書き込み
0	該当なし	×	×
ϵ	shr	○	×
1	mut	○	○

表 3.2: 権限の加減算

(a) 権限の加算 ($A + B$)

A \ B	B		
	0	ϵ	1
0	0	ϵ	1
ϵ	ϵ	ϵ	1
1	1	1	1

(b) 権限の減算 ($A - B$)

A \ B	B		
	0	ϵ	1
0	0	0	0
ϵ	ϵ	ϵ	0
1	1	1	0

```

1 let r1 = &mut 10;
2 let r2 = &unk *r1;
3
4 { let r3 = &shr *r2;
5   let r4 = &shr *r2;
6   println!("r3: {}, r4: {}", *r3, *r4); }
7
8 let r5 = &mut *r2;
9 *r5 = 20;

```

図 3.3: 実行中に権限の下限が増加するプログラムの例

ある参照の権限の下限は、その参照及びその子孫にあたる参照が再借用され子に権限を貸し出した際に、次の式に従って更新される。

$$\begin{aligned}
 \text{権限の下限} &= \text{peak}(\text{同時に子に貸し出している権限の総和}) \\
 &= \text{peak}(\sum_{\text{すべての子の参照}} \text{権限の下限})
 \end{aligned}
 \tag{3.1}$$

ただし、関数 $\text{peak}(E)$ は、参照が生成された時点から現在までの間における式 E の最大値を返す関数であり、権限の和は表 3.2a に従って計算される。また、可変性の指定が省略された参照から可変性が静的に指定された参照を再借用する場合には、その可変性に対応する権限を親となる参照から借り受け、自身の権限の下限とする。例えば、図 3.3 に示すプログラムにおいて、可変性の指定が省略された参照 $r2$ の権限の下限は、次のように変化する。参照 $r2$ は行 2 で生成される。このとき、参照 $r2$ の権限の下限は 0 に初期化される。その後、行 4 にて参照 $r3$ が共有参照として再借用される。このとき、参照 $r2$ は権限 ϵ を子に貸し出すため、参照 $r2$ の権限の下限は ϵ に更新される。続いて、行 5 にて参照 $r4$ が共有参照として再借用される。このとき、参照 $r2$ は、参照 $r3$ と参照 $r4$ に対してそれぞれ権限 ϵ を同時に貸し出しており、その和は $\epsilon + \epsilon = \epsilon$ である。したがって、参照 $r2$ の権限の下限は ϵ のままとなる。その後、参照 $r3$ と参照 $r4$ のライフタイムは行 6 で終了する。これによって、参照 $r2$ がこの時点で同時に子に貸し出している権限の総和は 0 となるが、参照 $r2$ の権限の下限は ϵ のままである。最後に、行 8 にて参照 $r5$ が可変参照として再借用される。このとき、参照 $r2$ は権限 1 を子に貸し出すため、参照 $r2$ の権限の下限は 1 に更新される。これらを踏まえると、参照 $r2$ のライフタイムが終了するまでに計算された権限の下限は 1 であるため、参照 $r2$ の可変性を静的に記述する場合には `mut` を選択するのが妥当である。

権限の上限は、現時点でその参照が親の参照から借りることができる最大の権限の大きさを表す値である。実行時検査器は、可変性の指定が省略された参照を通じて、この値よりも大きな権限を必要とする操作が行われようとした場合には、その操作が実行される直前に実行時エラーを発生させなければならない。権限の上限は、その参照が生成されてから現在までの間

22 第3章 漸進的借用検査

```
1 let r1 = &mut 10;
2 let r2 = &unk *r1;
3
4 let r3 = &unk *r2;
5
6 { let r4 = &shr *r2;
7   println!("r4: {}", *r4); }
8
9 let r5 = &mut *r3; // error
```

図 3.4: 漸進的借用検査によって実行時エラーが生じるプログラムの例

に、その参照の傍系^{*1}にあたる参照へと一度も貸し出されていない権限の大きさに一致する。言い換えれば、親の参照からある大きさの権限を借りるためには、その大きさの権限を、その参照が生成された時点から現在までの間継続して借りていたと仮定しても他の参照が同時に借りている権限と矛盾を生じないことを、過去を遡って確認する。このような制限を課すことで、権限の下限の値に基づいてプログラムに対応する可変性を静的に記述した際に、Aliasing XOR Mutability の規則を満たすことが期待される。

ある参照の権限の上限は次の式に従って計算され、自身の傍系にあたる参照が権限を借り受けた際に更新される。

$$\begin{aligned} \text{権限の上限} &= \text{bottom}(\text{親の参照の権限の上限} - \text{兄弟の参照に貸し出されている権限の総和}) \\ &= \text{bottom}(\text{親の参照の権限の上限} - \sum_{\text{すべての兄弟の参照}} \text{権限の下限}) \end{aligned} \quad (3.2)$$

ただし、関数 $\text{bottom}(E)$ は、参照が生成された時点から現在までの間における式 E の最小値を返す関数であり、権限の和及び減算はそれぞれ表 3.2a 及び表 3.2b に従って計算される。また、可変性が静的に指定された参照から可変性の指定が省略された参照を再借用する場合には、その可変性に対応する権限を親となる参照から借り受け、自身の権限の上限とする。例えば、図 3.4 に示すプログラムにおいて、可変性の指定が省略された参照 $r3$ の権限の上限は、次のように変化する。参照 $r3$ は行 4 で生成される。このとき、参照 $r3$ の権限の上限は、親である参照 $r2$ の権限の上限に一致する。参照 $r2$ は行 2 において可変参照 $r1$ から権限 1 を借り受けて生成されているため、この時点での参照 $r2$ 及び参照 $r3$ の権限の上限はともに 1 である。その後、行 6 にて、参照 $r2$ を親として参照 $r4$ が生成される。このとき、参照 $r4$ は参照 $r3$ の傍系にあたるため、参照 $r3$ の権限の上限を更新する。ここで、参照 $r2$ の権限の上限は 1 であり、参照 $r3$ の兄弟である参照 $r4$ の権限の下限は ϵ であるため、参照 $r3$ の権限の

^{*1} ある参照の傍系にあたる参照とは、その参照と同じ変数を指し示す参照のうち、その参照の先祖及び子孫に相当しない参照を指す。

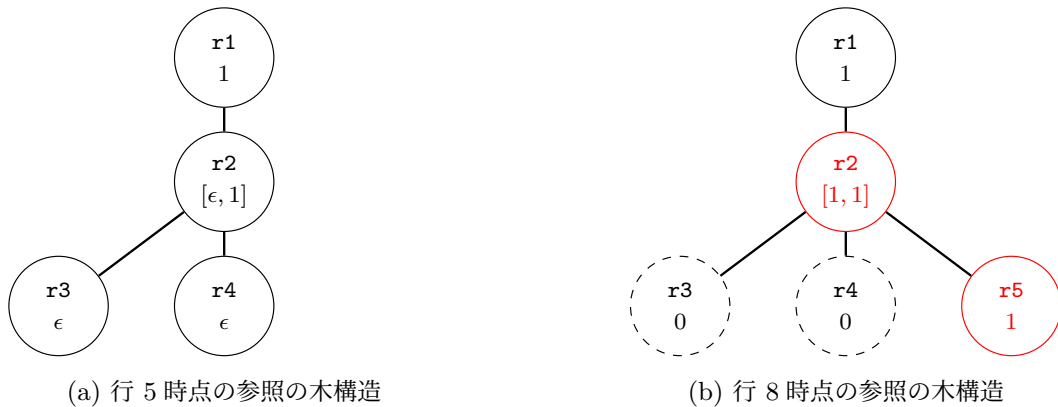


図 3.5: 図 3.3 のプログラムに対応する参照の木構造

上限は $1 - \epsilon = \epsilon$ に更新される．続いて，行 7 にて参照 $r4$ のライフタイムが終了し，兄弟の参照に貸し出されている権限の総和が 0 となるが，参照 $r3$ の権限の上限は ϵ のままである．最後に，行 9 にて可変参照 $r5$ を生成することを試みる．そのためには，参照 $r3$ は参照 $r5$ に対して権限 1 を貸し出す必要があるが，参照 $r3$ の権限の上限は ϵ であるため，この操作は実行時エラーとなる．

3.2.2 参照間での権限の貸し借り

漸進的借用検査における実行時検査器は，静的検査によって予め解析された参照のライフタイムに基づき，参照が再借用によって生成される時点及びライフタイムが終了する時点で実行時検査を行う．参照が再借用によって生成される時点では，次の二種類の処理を行う．一つは，可変性の指定が省略された参照から可変性が静的に指定された参照を再借用する場合に行われる．このとき，実行時検査器は，それぞれの参照の権限の範囲に基づいて権限の貸し借りが許容されるかを検査し，Aliasing XOR Mutability の規則に違反する矛盾が生じる場合には実行時エラーを発生させる．また，権限の貸し借りが許容された場合には，それぞれの参照の権限の下限及び上限を更新する．

参照の生成時における実行時検査のもう一つの処理は，可変性の指定が省略された参照から可変性の指定が省略された参照を再借用する場合に行われる．可変性の指定が省略された参照は，生成された時点では権限を持たないが，その後の操作に応じて親の参照から権限を借り受ける可能性がある．そのため，実行時検査器は再借用が行われるたびに，権限を貸し出す参照と貸し出される参照の間の親子関係を記憶する．漸進的借用検査では，Tree Borrows[10] と同様に，同じ変数を指し示す複数の参照の間の親子関係を木構造で管理する．これ以降，この木構造のことを単に「参照の木構造」と呼ぶこととする．参照の木構造は，木の根及び葉にあたる参照を可変性の指定が静的に指定された参照とし，その他の中間ノードのすべてを可変性の指定が省略された参照とすることで構成される．このとき，根にあたる参照はその木構造における最大の権限を提供し，葉にあたる参照はその参照を通じて行われる操作に必要な権限を

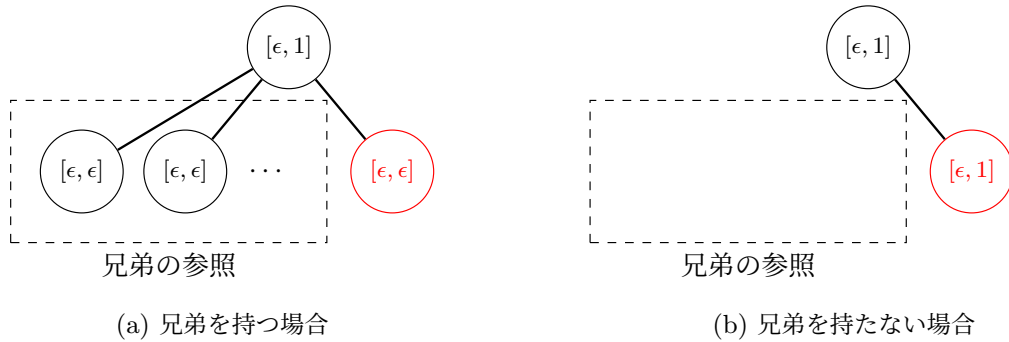


図 3.6: 兄弟の参照の有無による権限の上限の計算の違い

消費する。例として、図 3.3 のプログラムに対応する参照の木構造を図 3.5 に示す。参照の木構造には、再借用によって新しい参照が生成されるたびにノードが追加される。可変性が静的に指定された参照に対応するノード、すなわち根及び葉にあたるノードには、指定された可変性に対応する権限値が記録される。例えば、可変参照 r_1 は 1 であり、共有参照 r_3 及び r_4 はいずれも ϵ である。可変性の指定が省略された参照については、その参照の権限の下限及び上限が記録される。例えば、行 5 の時点では、参照 r_2 の権限の下限及び上限はそれぞれ ϵ 及び 1 であることが、図 3.5a に示されている。プログラムの実行が進むに従って、参照の木構造は変化する。ここでは、行 8 まで実行を進めた場合の参照の木構造を図 3.5b に示す。参照 r_3 及び参照 r_4 のライフタイムが終了したため、それらに対応するノードは点線で示され、権限値は 0 に更新されている。また、参照 r_5 が可変参照として再借用されたため、そのノードが追加され、対応する権限値は 1 に設定されている。これによって、参照 r_2 の権限の下限が 1 に更新されていることがわかる。

参照の権限の下限及び上限の計算にあたっては、現時点で子にあたる参照に貸し出している権限の大きさを記憶しておくことが望ましい。この情報のことを「貸出中の権限」と呼ぶこととする。式 (3.1) を単純に利用して権限の下限を計算する場合、自身の権限の下限の値は子の参照の権限の下限の総和に依存するため、ある参照の権限の下限を更新するためには、その参照の子孫にあたる部分木を全探索して、葉にあたる参照の権限の総和を計算しなければならない。一方、貸出中の権限を記憶する場合、新しく権限を借り受ける参照は、その祖先にあたる参照に対して自身が借り受ける権限の大きさを通知し、それを受けて権限の下限及び貸出中の権限を更新すればよい。計算量を大幅に削減することができる。すなわち、式 (3.1) は次のように書き換えられる。

$$\begin{aligned}
 \text{権限の下限} &= \text{peak}\left(\sum_{\text{すべての子の参照}} \text{権限の下限}\right) \\
 &= \text{peak}(\text{貸出中の権限})
 \end{aligned}
 \tag{3.3}$$

貸出中の権限として記憶する情報には、権限の大きさだけでなく、同時に権限を貸し出している子の参照の数を含まなければならない。この情報が含まれない場合、貸出中の権限を用いて権限の上限を計算する際に問題が生じる。貸出中の権限を用いて権限の上限を計算するよう

に式 (3.2) を書き換えると、次のようになる。

$$\begin{aligned}
 \text{権限の上限} &= \text{bottom}(\text{親の参照の権限の上限} - \sum_{\text{すべての兄弟の参照}} \text{権限の下限}) \\
 &= \text{bottom}(\text{親の参照の権限の上限} - (\text{親の参照の貸出中の権限} \ominus \text{権限の下限}))
 \end{aligned}
 \tag{3.4}$$

この式では、貸出中の権限が子の参照の権限の下限の総和であることを踏まえ、貸出中の権限から自身の権限の下限を差し引くことで他の兄弟の参照の権限の下限の総和を計算する。このとき、自身が親から借りている権限の大きさが ϵ である場合には、図 3.6 に示すように、兄弟の参照が存在する場合としない場合を区別する必要がある。これらの例において、強調された参照に着目すると、兄弟の有無に関わらず、親の参照の権限の上限は 1 であり、貸出中の権限の大きさは ϵ である。しかしながら、期待される権限の上限の値は、図 3.6a に示すように兄弟の参照が存在する場合には ϵ であり、図 3.6b に示すように兄弟の参照が存在しない場合には 1 である。このような振る舞いを実現するため、兄弟の参照の権限の下限の総和を表す (親の参照の貸出中の権限 \ominus 権限の下限) の値は、図 3.6a に示すように兄弟の参照が存在する場合には ϵ である一方、図 3.6b に示すように兄弟の参照が存在しない場合には 0 となるように定義されなければならない。このような権限の減算は、表 3.3 に示すように定義される。特に、貸出中の参照の数 c が 1 である場合に限り $\epsilon - \epsilon = 0$ となる点に注意されたい。なお、貸出中の権限の大きさが 0 または 1 である場合には、貸出中の参照の数はそれぞれ 0 または 1 であることが自明であるため、これを考慮する必要はない。

実行時検査器は、参照のライフタイムが終了する際に、自身の権限の下限及び上限を 0 へとリセットし、それ以降その参照を通じて操作を行うことができないようにする。このとき、親の参照から借りていた権限を返却し、親の参照の貸出中の権限を更新する。権限の返却は、ライフタイムが終了した参照の直接の親に対してのみ行われ、親の参照よりも祖先の参照には伝播しない。例えば、図 3.5b に示す参照の木構造においては、すでにライフタイムが終了している参照 r3 及び参照 r4 の権限が 0 に更新されている。このとき、参照 r3 及び参照 r4 が親である参照 r2 に返却した権限 ϵ は、参照 r2 が新しく参照 r1 から借り受けた権限と合わせて、参照 r5 へと再び貸し出されている。

ライフタイムの終了時における権限の返却は、可変性の指定が省略された参照のみならず、可変性の指定が省略された参照を再借用することによって生成された、可変性が静的に指定さ

表 3.3: 貸出中の参照の数を考慮した権限の減算 ($A \ominus B$)

		B		
		0	ϵ	1
A	0	0	0	0
	$c = 1$	ϵ	0	0
	$c > 1$	ϵ	ϵ	0
	1	1	ϵ	0

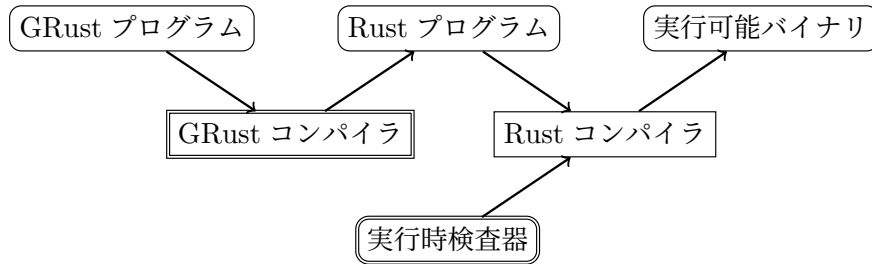


図 3.7: GRust プログラムのコンパイル

れた参照についても行う必要がある。通常の静的な借用検査においては、ライフタイムの解析結果は実行時のプログラムの振る舞いに影響を与えず、参照のライフタイムが終了した後は、参照は単にメモリ上から解放されるだけである。しかしながら、漸進的借用検査においては、可変性が静的に指定された参照もライフタイムの終了時に親の参照に権限を返却することで、他の参照が新しく権限を借り受けることができるようにしなければならない。これによって、プログラムの中で可変性の指定が省略された部分や、可変性が静的に記述された部分との境界だけでなく、可変性が静的に記述された部分の内部であったとしても、その参照が可変性の指定が省略された参照を経由して生成されている場合には、実行時検査器の処理を挿入しなければならない。

3.3 漸進的借用検査の Rust 言語への統合

漸進的借用検査を導入し、参照における可変性の指定を省略することができる Rust 言語は、Rust コンパイラの拡張と、通常の Rust 言語で実装したライブラリの組み合わせによって実現できると考えられる。これ以降、この漸進的借用検査を導入した Rust 言語を、GRust と呼ぶこととする。GRust は、Rust 言語のサブセットに可変性の指定が省略された参照を導入した言語である。GRust がサポートする主な Rust の言語機能は、次のとおりである。

- 変数宣言
- 変数への参照
- 構造体宣言
- 代入
- 関数宣言
- 構造体アクセス
- 制御構造
- 関数呼び出し
- Unsafe
- 例外処理
- 再帰呼び出し

図 3.7 に示すのは、GRust プログラムを実行するためのコンパイルの流れである。可変性の指定が省略された参照を含む GRust のプログラムは、GRust コンパイラによって、可変性の指定の省略を含まない通常の Rust プログラムに変換される。この Rust プログラムを通常の Rust コンパイラでコンパイルすることで、実行可能なバイナリが生成される。このとき、通常の Rust 言語で実装された実行時検査器が、Rust ライブラリとしてバイナリに組み込まれる。

GRust プログラムのコンパイルにおける GRust コンパイラの主な役割は、Rust ライブラ

りとして実装された実行時検査器の呼び出しを、プログラム中の適切な位置に挿入することである。実行時検査器の呼び出しが挿入されるのは、参照が再借用によって生成される時点と、参照のライフタイムが終了する時点の二つである。参照のライフタイムは、Rust コンパイラによって MIR[13] を静的に解析することによって得ることができる。MIR とは、制御フロー解析のために単純化された Rust 言語の中間表現であり、Rust コンパイラによって Rust プログラムから生成される。

参照が再借用によって生成される際には、親となる参照の可変性と新しく生成される参照の可変性の双方を、静的解析の結果として実行時検査器に渡す必要がある。再借用による参照の生成は、明示的に再借用の演算子を用いた場合だけでなく、関数呼び出しや変数への代入においても暗黙的に行われる場合がある。このような暗黙的な再借用は、Rust コンパイラが型検査を行う過程で自動的に検出され、明示的な再借用の演算子として MIR に出力される。したがって、再借用による参照の生成はすべて、MIR 上の再借用の演算子を検出することで特定することができる。MIR 上の再借用の演算子に指定された可変性と、親となる参照の型情報に含まれる可変性を実行時検査器に渡すことで、参照の再借用時に適切な実行時検査を行うことができる。

参照のライフタイムが終了する時点では、実行時検査器による権限の返却を行う必要がある。この実行時検査器の呼び出しを挿入する位置は、参照のライフタイムの定め方によって異なる。参照のライフタイムをスコープによって定める Lexical Lifetimes を採用する場合には、Rust コンパイラによって自動挿入される、デストラクタの呼び出しを利用することができる。Rust コンパイラは、変数のスコープが終了する際に自動的にデストラクタを呼び出す。多くの場合、参照のライフタイムはその参照が束縛されている変数のスコープに一致するため、参照のデストラクタとして実行時検査器の呼び出しを挿入することで、参照のライフタイムの終了時に権限の返却を行うことができる。

一方、参照が最後に使用される文を参照のライフタイムの終了と定める Non-Lexical Lifetimes[6] を採用する場合には、変数のスコープが終了するよりも前に参照のライフタイムが終了する可能性がある。そのため、変数のスコープよりも短いライフタイムを持つ参照については、ライフタイムが終了する時点で明示的にデストラクタの呼び出しを挿入する必要がある。例えば、図 3.8 に示すプログラムにおいては、行 6 における参照 r4 の使用をもって参照 r3 及び参照 r4 のライフタイムが終了する。しかしながら、これらの参照のスコープは行 11 まで続いており、スコープの終了よりも前にライフタイムが終了している。そのため、GRust コンパイラは、ライフタイムの終了時点より後ろであって可能な限り早い位置に、参照 r3 及び参照 r4 に対する `drop` 関数の呼び出しを挿入する。`drop` 関数は、引数に与えられた値に対するデストラクタを明示的に呼び出し、それ以降その変数を使用できないようにする Rust 標準ライブラリの関数である。このようにして挿入されたデストラクタの呼び出しによって権限の返却が行われることで、行 10 における可変参照 r5 の生成が可能となる。なお、参照 r5 のライフタイムは行 11 における使用をもって終了するが、これは変数のスコープの終了と一致しているため、明示的にデストラクタの呼び出しを挿入する必要はない。

権限の返却をデストラクタによって行う場合には、参照のライフタイムが変数のスコープよ

```

1 { let r1 = &mut 10;
2   let r2 = &unk *r1;
3
4   let r3 = &unk *r2;
5   let r4 = &shr *r3;
6   println!("r4: {}", *r4);
7   drop(r4);
8   drop(r3);
9
10  let r5 = &mut *r2;
11  *r5 = 20; }
```

図 3.8: Non-Lexical Lifetimes におけるデストラクタ呼び出しの挿入の例

```

1 fn copy(source: &unk i32) -> &unk i32 {
2   let dest = &unk *source;
3   dest
4 }
5 fn main() {
6   let r1 = &mut 10;
7   let r2 = &unk *r1;
8   let r3 = copy(r2);
9   let r4 = &shr *r3;
10  println!("r4: {}", *r4);
11 }
```

図 3.9: 権限の返却が遅延されるプログラムの例

りも長い場合に細心の注意を払う必要がある。このような参照は、引数で受け取った参照を複製し、新しい参照を返すような関数によって生成される場合がある。このような関数を宣言するためには、引数で受け取る参照のライフタイムを関数のスコープよりも長く指定する必要があるため、ライフタイムが終了していないにも関わらずデストラクタが呼び出される。このような問題を回避するため、デストラクタによって実行時検査器が呼び出された際には、まずその参照に削除フラグを立てる。削除フラグが立てられた参照における権限の返却は、その参照のすべての子孫にあたる参照に削除フラグが立てられるまで遅延する。例えば、図 3.9 に示すプログラム中の `copy` 関数は、渡された参照を再借用によって複製し、新しい参照を返す関数



(a) 行 9 時点の参照の木構造

(b) 行 10 時点の参照の木構造

図 3.10: 権限の返却が遅延される場合の参照の木構造の変化

であるが、`copy` 関数のスコープを抜ける際に、引数として渡した参照 `source` のデストラクタが呼び出される。そのため、このプログラムの `main` 関数を実行すると、行 8 における関数 `copy` の呼び出し後、引数として渡した参照 `r2` のデストラクタが呼び出され、削除フラグが立てられる。その後、参照 `r4` が生成される行 9 の時点における参照の木構造を図 3.10a に示す。参照 `r2` に削除フラグが立てられているが、その子孫である参照 `r3` には削除フラグが立てられていないため、参照 `r2` の権限の返却は遅延される。その後、行 10 における参照 `r4` の使用をもって、参照 `r3` 及び参照 `r4` のライフタイムが終了し、それらの参照のデストラクタが呼び出される。この時点における参照の木構造を図 3.10b に示す。参照 `r3` 及び参照 `r4` に削除フラグが立てられたため、参照 `r4`、参照 `r3`、参照 `r2` の順に権限の返却が行われ、最終的に参照 `r2` の権限の下限が 0 に更新されていることがわかる。

GRust コンパイラは、実行時検査器による権限の返却をデストラクタとして実装するため、可変性の指定が省略された参照と、それを經由して生成された可変性が静的に指定された参照の双方を、独自のデータ構造に置き換える。Rust においては、純粋な参照型には独自のデストラクタを定義することができないが、`Deref` トレイトや `DerefMut` トレイトを実装したデータ構造を宣言することによって、参照のように振る舞いながら独自のデストラクタを持つ型を定義することができる。これらの参照を表すデータ構造には、参照の指し示す先を表すポインタの他に、参照の木構造における自身のノードへのポインタを含める。これによって、デストラクタが呼び出された際に、親の参照から借り受けていた権限の返却先を特定することができる。参照の木構造へのポインタを保持しなければならないため、可変性が静的に指定された参照を表すデータ構造は、通常の参照型よりも大きなメモリ領域を消費し、異なるデータ型として扱われる。そのため、通常の参照型を受け取るように実装された既存の Rust ライブラリを利用できるようにするためには、通常の参照型の代わりに独自のデータ構造による参照型を受け取るようにライブラリを修正する必要がある。

実行時検査器は、通常の Rust ライブラリとして実装される。実行時検査器による参照の操作は、Rust の `unsafe` ブロックによって実装される。`unsafe` ブロックは、借用検査をはじめとする Rust の静的検査による安全性の保証を無効化し、プログラマが任意の低レベルな操作を行うことを可能にする機能である。`unsafe` ブロックを利用するには、記述するプログラムがメモリ安全性を侵害しないことをプログラマが保証する責任を負う。漸進的借用検査においては、通常の借用検査によって静的に安全性を保証することができない参照の操作を伴うため、ライブラリ内部には `unsafe` ブロックを用いる必要がある。しかしながら、実行時検査によってメモリ安全性が保証されるため、この `unsafe` ブロックの影響はライブラリの内部に限定され、ライブラリの利用者は安全にプログラムを記述することができる。

3.4 Gradual Guarantee を満たすための設計

漸進的借用検査は、Gradual Guarantee[1] を満たすように設計することが望ましい。Gradual Guarantee の主要な性質の一つは、正しく型付けされたプログラムの型注釈を削除しても、プログラムは正しく型付けされ、同じように振る舞わなければならないというものである。この性質を保証するために、漸進的借用検査では、可変性の指定が省略された参照に関するサブタイプ関係に特別な注意を払う必要がある。Rust においては、それぞれの参照に許容される操作に基づいて、共有参照 `&shr T` における被参照型 `T` の変性 (Variance) は共変 (Covariant) であり、可変参照 `&mut T` における被参照型 `T` の変性は不変 (Invariant) であると定められている。すなわち、以下のサブタイプ関係が成り立つ。

$$\&shr T <: \&shr U \quad \text{only if } T <: U \quad (3.5)$$

$$\&mut T <: \&mut U \quad \text{only if } T = U \quad (3.6)$$

共有参照の被参照型が共変であることを踏まえると、可変性の指定が省略された参照 `&unk T` における被参照型 `T` の変性は共変であると定義しなければならない。さもなくば、この共変性を用いて型付けされたプログラムにおいて、共有参照の可変性の指定を省略した場合に、型付けに失敗する可能性がある。

加えて、この共変性を用いた型キャストによって生成された参照には、通常の実行時検査による制約に加えて、その権限の上限が ϵ よりも大きくならないという追加の制約を課す必要がある。なぜならば、共有参照の共変性は参照の指し示す先の値が変更されないことに基づいており、共変性を用いてキャストした参照を通じて参照の指し示す先の値を変更することは、型システムの安全性を侵害するからである。

第 4 章

評価

4.1 実験設定

本論文では、漸進的借用検査による実行時性能の影響を評価するため、既存の Rust プログラムに対して実行時検査器を組み込み、その性能の変化を計測する。可変性の指定を省略した GRust のプログラムを実行する場合を想定し、既存の Rust プログラムに含まれる可変性の指定を段階的に削除する。可変性の指定が削除された参照の利用にあたっては、通常の Rust ライブラリとして実装された実行時検査器を呼び出すように実装し、Rust プログラムの実行にかかる実行時間とメモリ使用量を計測する。

性能を評価するプログラムとして、Rust で記述された代表的なベンチマークスイートからいくつかのプログラムを選択する。プログラムを選択するにあたっては、外部ライブラリの使用をはじめとする、GRust がサポートしない言語機能を用いるものを除外する。また、メンバ関数の宣言や可変長スライスなどの、実行時検査器のライブラリとの互換性がない機能を使用しているプログラムについては、プログラムの振る舞いを変化させないように注意しながら、別の表現を用いた実装に置き換える。

選択するプログラムは、次の 5 つである。まず、著名なプログラミング言語の性能を比較するために広く用いられているベンチマークスイートである The Computer Language Benchmarks Game^{*1}から、fannkuch-redux, n-body の 2 つのプログラムを選択する。加えて、言語処理系の性能を評価するために用いられるベンチマークスイートである “Are We Fast Yet?” [14] から、bounce, queens, towers の 3 つのプログラムを選択する。ただし、“Are We Fast Yet?” の Rust 実装は執筆時点で公式には提供されておらず、提案段階にあるもの^{*2*3}を用いた。

それぞれの Rust プログラムについて、可変性が静的に記述された参照の利用箇所を特定し、可変性の指定の有無の組み合わせを変化させた複数の GRust プログラムを網羅的に生成

^{*1} <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>

^{*2} <https://github.com/smarr/are-we-fast-yet/pull/66>

^{*3} <https://github.com/OvermindDL1/are-we-fast-yet/tree/78b0560c7c3b7a90b22b90c4cb27660e56478d30/benchmarks/Rust/src>

<pre> 1 fn place_queen(2 queens: &mut Queens, 3 c: usize, 4) -> bool { ... } 5 fn get_row_column(6 queens: &shr Queens, 7 r: usize, 8 c: usize, 9) -> bool { ... } 10 fn set_row_column(11 queens: &mut Queens, 12 r: usize, 13 c: usize, 14 v: bool, 15) { ... } </pre>	<pre> 1 fn place_queen(2 queens: &unk Queens, 3 c: usize, 4) -> bool { ... } 5 fn get_row_column(6 queens: &shr Queens, 7 r: usize, 8 c: usize, 9) -> bool { ... } 10 fn set_row_column(11 queens: &unk Queens, 12 r: usize, 13 c: usize, 14 v: bool, 15) { ... } </pre>
(a) オリジナルの Rust プログラム	(b) 一部の可変性の指定を削除したプログラム

図 4.1: queens ベンチマークにおける関数宣言の一部

する。生成された複数のプログラムのそれぞれについて、オリジナルのプログラムにおける参照の個数に対する可変性の指定を削除した参照の個数の割合を計算する。例えば、図 4.1 に示すのは収集した queens ベンチマークのプログラムに含まれる関数宣言の一部である。このプログラムには可変性が静的に記述された参照が 3 つ存在するため、可変性の指定の有無の組み合わせは $2^3 = 8$ 通り存在する。このうち、図 4.1a に示すプログラムはすべての参照に可変性が指定されているため、可変性の指定を削除した参照の割合は 0% である。一方、図 4.1b に示すプログラムは、2 つの参照から可変性の指定が削除されているため、可変性の指定を削除した参照の割合は 66.6% である。

この実験においては、生成された複数の GRust プログラムを、GRust コンパイラを用いてコンパイルする代わりに、手作業にて通常の Rust プログラムに変換する。GRust プログラムを、通常の Rust コンパイラによってコンパイルすることができる Rust プログラムへと変換するためには、主に以下の三つの処理を行う必要がある。一つは、可変性の指定が省略された参照と、それらを経由して生成された可変性が静的に指定された参照の型宣言を、実行時検査器が提供する独自のデータ構造を表す型宣言に置き換えることである。二つ目は、これらの参照を生成するための再借用の操作を、実行時検査器が提供する関数を呼び出す形に置き換えることである。三つ目は、これらの参照のライフタイムが変数のスコープよりも短い場合に、必要に応じて明示的にデストラクタを呼び出す `drop` 関数の呼び出しを挿入することである。こ

これらの処理を適切に行うことで、通常の Rust コンパイラが受け付ける Rust プログラムへと変換されることが期待される。

本実験は、Azure Virtual Machines 上に確保した Ubuntu 24.04 環境で実施した。この環境は、Intel Xeon E5-2673 v4 (2.30GHz) プロセッサを搭載したマシン上に割り当てられており、8vCPU と 32GiB のメモリを利用することができる Standard_D8s_v3 インスタンスである。また、Rust コンパイラとして rustc 1.90.0 の開発版^{*4}を使用した。

4.2 実験結果

4.2.1 実行時間

それぞれのベンチマークプログラムについて、可変性の指定を削除した参照の割合を変化させた場合の実行時間の变化を測定した。実行時間を計測するために適切なパラメータをベンチマークプログラムごとに選択し、それぞれの可変性の指定の有無の組み合わせごとに、選択したパラメータで各プログラムを 10 回ずつ実行した。実行時間は time コマンドを用いて計測し、wall clock time を採用した。特に記載がない場合は、10 回の実行の平均値を結果として示す。

各ベンチマークにおける実行時間の計測結果について、一元配置分散分析及びクラスカル・ウォリス検定を用いて有意差があるかを検定したところ、有意水準 1% において全てのベンチマークプログラムで有意差が認められた。その結果を表 4.1 に示す。

The Computer Language Benchmarks Game から選択した fannkuch-redux ベンチマークの実行時間の結果を図 4.2 に示す。このベンチマークプログラムには参照が 4 つ含まれているため、16 通りの GRust プログラムが生成できる。このプログラムでは、ある参照の可変性を削除すると、配列の順序を反転させる関数 `reverse` の内部で可変性の指定が省略された参照から可変参照への再借用が繰り返し行われ、実行時検査による権限の貸し借りと返却が繰り返される。このため、ある特定の参照の可変性を削除した場合に、他の参照の可変性を削除した

表 4.1: 各ベンチマークプログラムにおける実行時間の有意差検定

ベンチマーク	一元配置分散分析		クラスカル・ウォリス検定	
	F 値	p 値	H 値	p 値
fannkuch-redux	930.6620	< 0.001	156.6574	< 0.001
n-body	2561.1048	< 0.001	628.4783	< 0.001
bounce	5776.9749	< 0.001	76.7530	< 0.001
queens	22219.5564	< 0.001	157.6450	< 0.001
towers	20274.0751	< 0.001	2529.3603	< 0.001

^{*4} <https://github.com/rust-lang/rust/tree/b63223c152212832ce37a109e26cc5f84c577532>

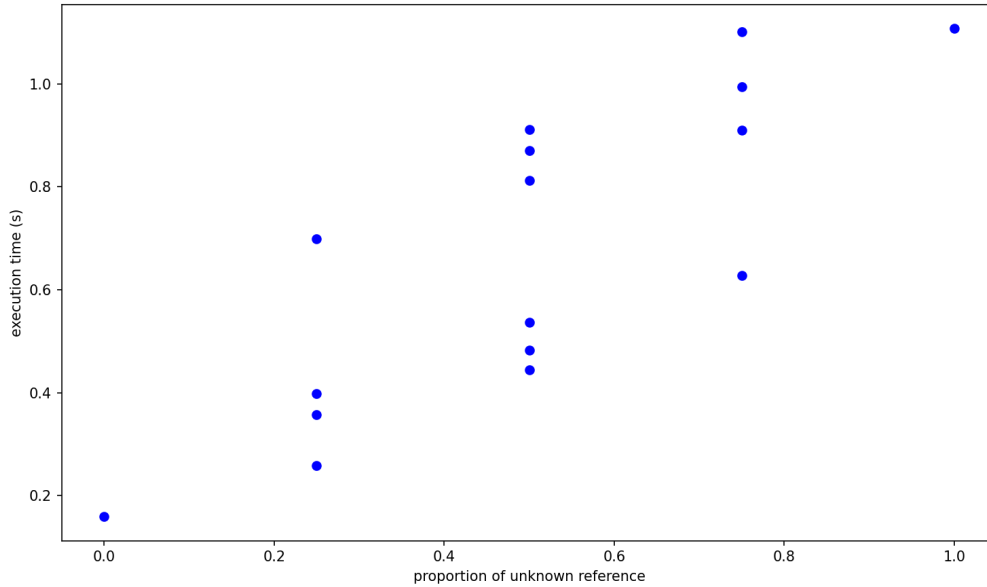


図 4.2: fannkuch-redux ベンチマークの実行時間

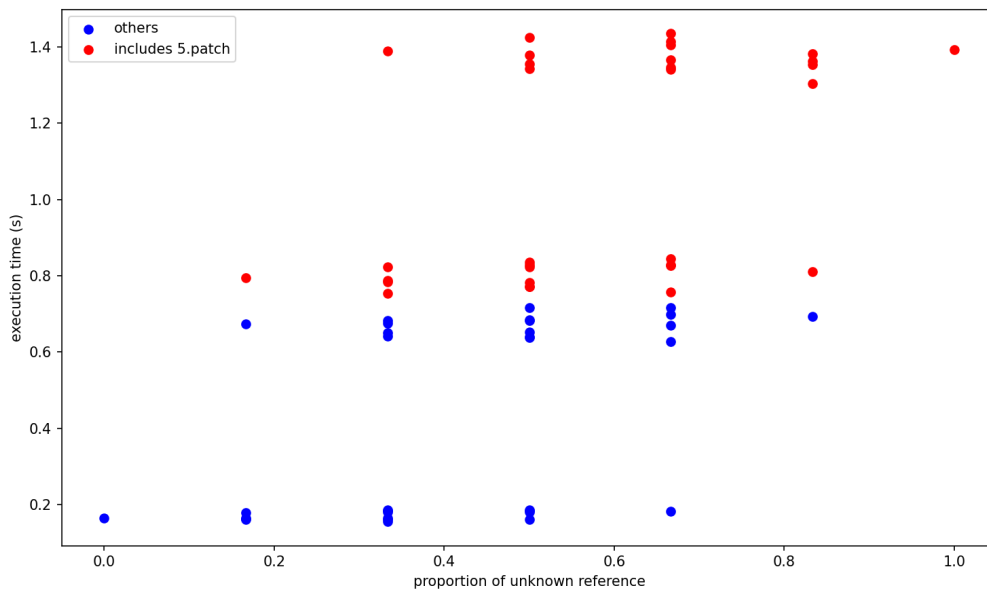


図 4.3: n-body ベンチマークの実行時間

場合と比較して実行時間が大きく増加する傾向が観測された。一方で、他の参照の変性を削除した場合も実行時間が緩やかに増加し、全体として変性の指定を削除した参照の割合が増加するにつれて実行時間が増加する傾向が見られた。

The Computer Language Benchmarks Game から選択した n-body ベンチマークの実行時間の結果を図 4.3 に示す。このベンチマークプログラムには参照が 6 つ含まれているため、64 通りの GRust プログラムが生成できる。このプログラムでは、図 4.4 に示すように、ループ

```

1 fn shift_vec_idx<T>(
2     v: &shr Vec<T>, idx: &mut usize,
3 ) -> Option<usize> {
4     if *idx >= v.len() { return None }
5     let result = *idx;
6     *idx += 1;
7     Some(result)
8 }

```

図 4.4: n-body ベンチマークにおける関数 `shift_vec_idx` の定義

の内部で繰り返し呼び出される関数 `shift_vec_idx` の引数として、2つの参照 `v` 及び `idx` が用いられている。そのため、これらの参照の可変性を削除しない場合、いずれか一方の参照の可変性を削除した場合、双方の参照の可変性を削除した場合で、実行時間の傾向に違いが見られた。

さらに、関数 `shift_vec_idx` の2つの引数の間でも、実行時間への影響に差が見られた。図 4.3 では、参照 `idx` の可変性を削除した場合の実行時間を赤く示している。0.7秒付近に分布する、2つの参照のいずれか一方の可変性を削除した場合を示す点群のうち、参照 `idx` の可変性を削除した場合の実行時間が、参照 `v` の可変性を削除した場合の実行時間よりも大きくなる傾向が観測された。これは、関数 `shift_vec_idx` の内部で、参照 `v` は1回だけ使用されるのに対し、参照 `idx` は3回使用されるためであると考えられる。また、参照の使用回数が3倍に増加するにもかかわらず、実行時間への影響がそれほど大きくないことから、実行時検査における権限の貸し借りと比較して、新しく参照の木構造を生成するオーバーヘッドが大きいことが示唆される。

“Are We Fast Yet?”から選択した `bounce` ベンチマークの実行時間の結果を図 4.5 に示す。このベンチマークプログラムには参照が3つ含まれているため、8通りの GRust プログラムが生成できる。このプログラムにおいても、`fannkuch-redux` ベンチマークと同様に、ループの内部で繰り返し呼び出される特定の関数の引数における可変性の指定を削除した場合に、実行時間が大きく増加する傾向が観測された。この関数の内部における参照の再借用の回数が `fannkuch-redux` ベンチマークと比較して非常に多いことから、`fannkuch-redux` ベンチマークよりも実行時間への影響が顕著に現れていると考えられる。

“Are We Fast Yet?”から選択した `queens` ベンチマークの実行時間の結果を図 4.6 に示す。このベンチマークプログラムには参照が4つ含まれているため、16通りの GRust プログラムが生成できる。このプログラムにおいては、他のベンチマークプログラムとは異なり、特定の2つの参照の可変性を同時に削除した場合に限って実行時間が大きく増加する傾向が観測された。これは、2つの参照の可変性が同時に削除されることによって参照の木構造の規模が大き

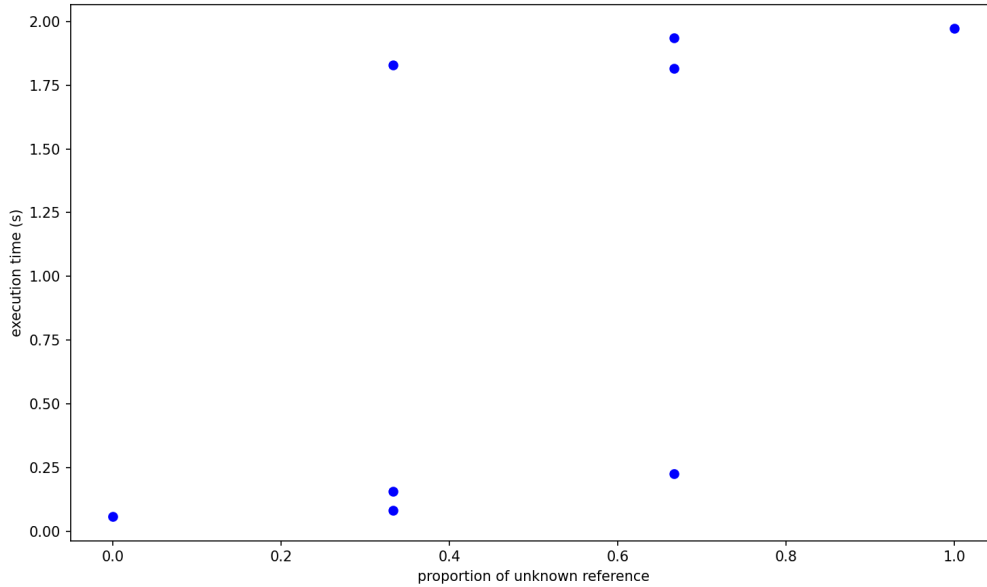


図 4.5: bounce ベンチマークの実行時間

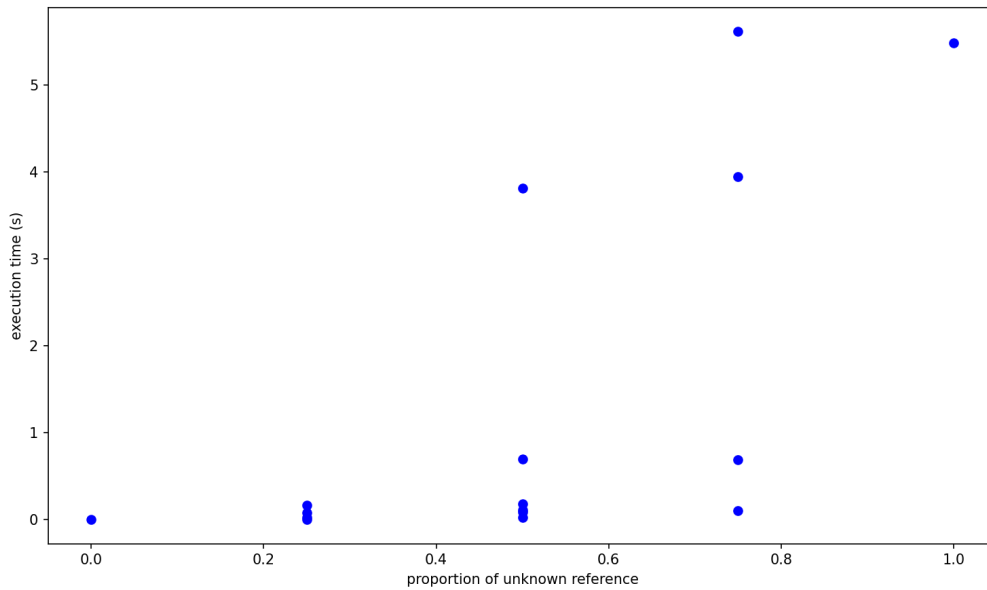
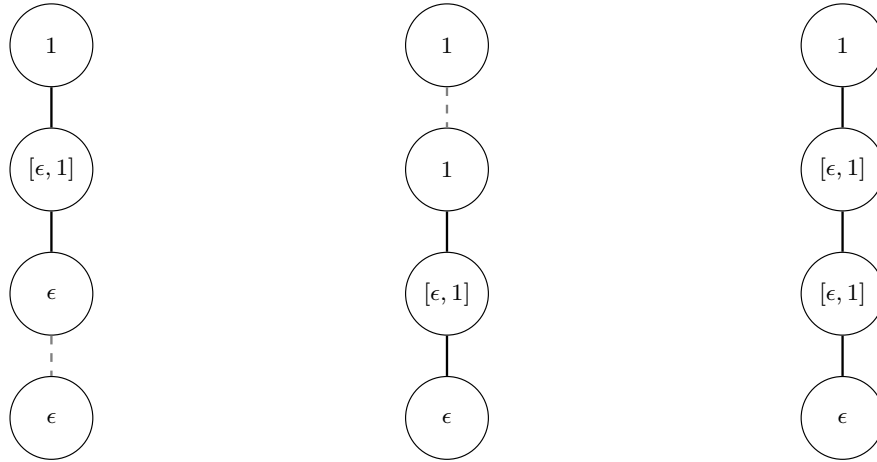


図 4.6: queens ベンチマークの実行時間

くなり、実行時検査器のオーバーヘッドが増加したためであると考えられる。図 4.7 に示すのは、可変性の指定を削除する参照の組み合わせを変化させた場合における、プログラム中の同じ時点における参照の木構造の例である。この図中において点線で示された辺は、静的に可変性が指定され、静的に借用検査が行われた参照間の関係を表している。そのため、この辺は実行時の参照の木構造には含まれず、実行時検査器のオーバーヘッドを生じさせない。いずれか一方の可変性の指定だけを削除した場合には、図 4.7a 及び図 4.7b に示すように、参照の木構



(a) 親側の参照の可変性を削除 (b) 子側の参照の可変性を削除 (c) 双方の参照の可変性を削除

図 4.7: queens ベンチマークのある時点における参照の木構造

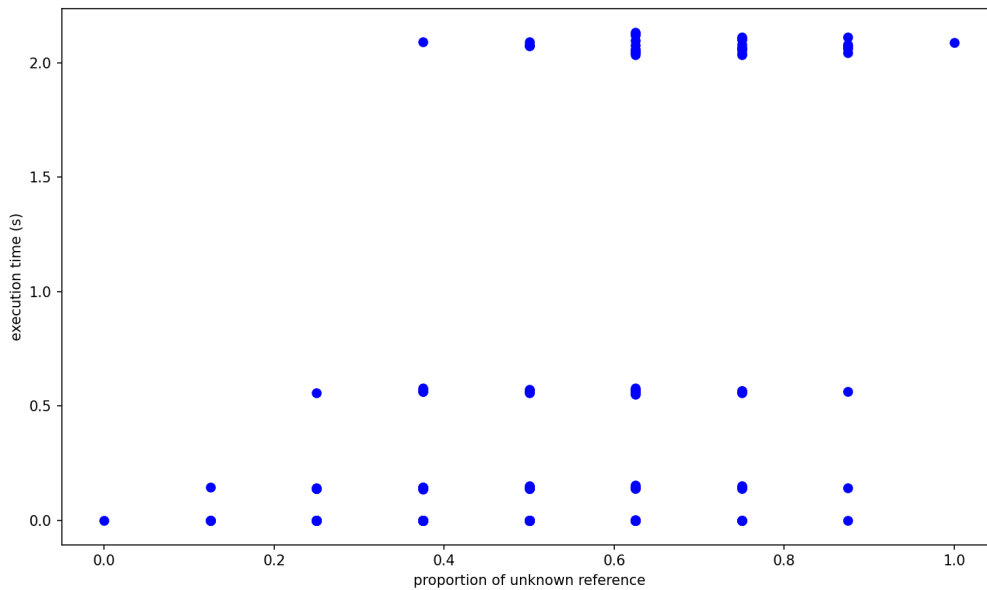


図 4.8: towers ベンチマークの実行時間

造は3つのノードから構成される。一方で、双方の可変性の指定を削除した場合には、図 4.7c に示すように、参照の木構造は4つのノードから構成され、高さが4の木となる。このように、参照の木構造の規模が大きくなることは、実行時検査器のオーバーヘッドの増加につながり、実行時間の増加を引き起こすと考えられる。

“Are We Fast Yet?” から選択した towers ベンチマークの実行時間の結果を図 4.8 に示す。このベンチマークプログラムには参照が8つ含まれているため、256通りの GRust プログラムが生成できる。他のベンチマークプログラムと同様に、可変性の指定を削除した参照の割合が増加するにつれて実行時間が増加する傾向や、特定の組み合わせで可変性の指定を削除した

場合に実行時間に大きな影響が現れる傾向が観測された。

4.2.2 メモリ消費量

メモリ消費量についても実行時間と同様に、それぞれのベンチマークプログラムについて、可変性の指定を削除した参照の割合を変化させた場合の変化を測定した。実行時間を計測するために使用したのと同じパラメータを選択し、それぞれの可変性の指定の有無の組み合わせごとに、選択したパラメータで各プログラムを10回ずつ実行した。メモリ消費量はtimeコマンドを用いて計測し、maximum resident set sizeを採用した。特に記載がない場合は、10回の実行の平均値を結果として示す。

各ベンチマークにおけるメモリ消費量の計測結果について、一元配置分散分析及びクラスカル・ウォリス検定を用いて有意差があるかを検定したところ、有意水準1%において towers ベンチマークのみに有意差が認められた。その結果を表4.2に示す。

有意差が見られなかった4つのベンチマークプログラムにおけるメモリ消費量の結果を図4.9に示す。これらのベンチマークプログラムにおいては、可変性の指定を削除した参照の割合を変化させた場合においても、メモリ消費量の変化に特別な傾向は観測されなかった。これは、プログラムの実行中に長期間に渡ってメモリ上に保持され続ける参照が少なく、オリジナルのRustプログラムが使用するメモリ量に対して、実行時検査器が追加で消費するメモリ量が相対的に小さいためであると考えられる。

実行時検査器は、プログラムの実行中に、参照の木構造に含まれる全ての参照のライフタイムが終了すると、参照の木構造をメモリから解放する。ガベージコレクションを採用せず、利用されなくなったメモリ領域を逐次的に解放するRustのメモリ管理方式により、参照の木構造に含まれる参照のライフタイムが短い場合、実行時検査器が確保したメモリ領域は早期に解放されることが期待される。したがって、特定のループの内部で参照の生成と破棄が繰り返されるベンチマークプログラムにおいては、多くのメモリ領域を一度に確保することなく、同じメモリ領域が繰り返し再利用される可能性が高い。

特に顕著な傾向が観測された towers ベンチマークのメモリ消費量の結果を図4.10に示す。

表 4.2: 各ベンチマークプログラムにおけるメモリ消費量の有意差検定

ベンチマーク	一元配置分散分析		クラスカル・ウォリス検定	
	F 値	p 値	H 値	p 値
fannkuch-redux	0.5570	0.903	7.5733	0.940
n-body	1.1129	0.265	62.4932	0.494
bounce	0.7081	0.665	4.4357	0.728
queens	1.8235	0.037	17.7525	0.275
towers	2279.2262	< 0.001	2221.8827	< 0.001

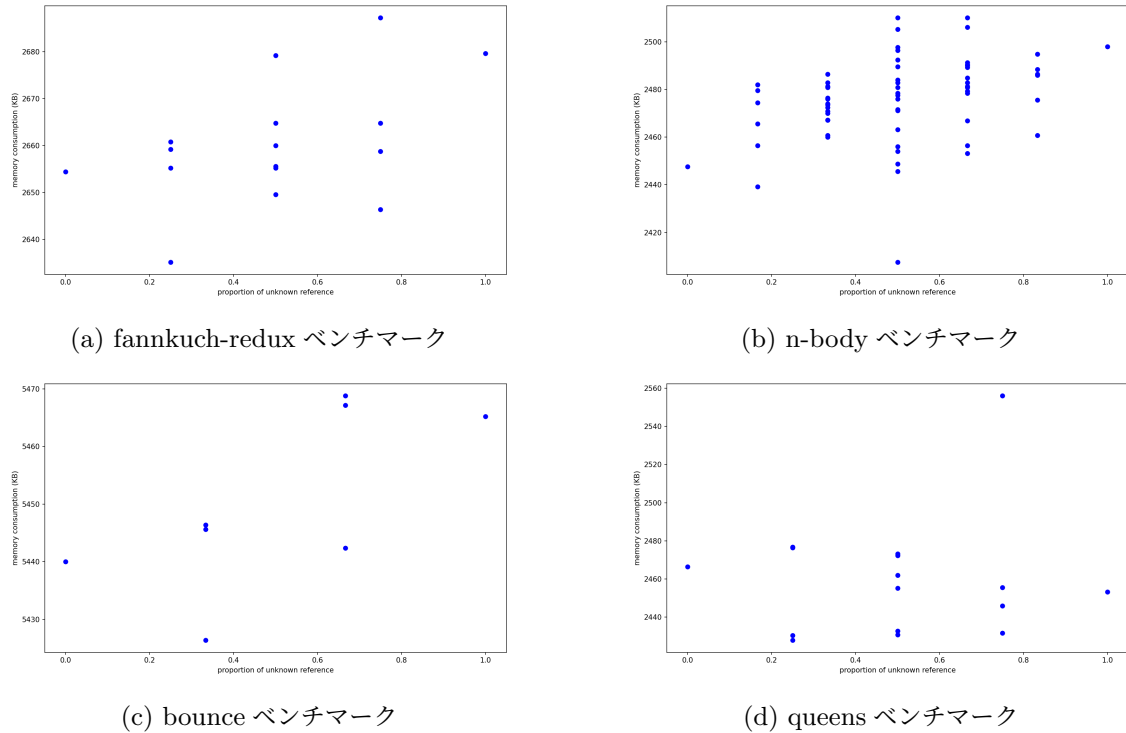


図 4.9: 各ベンチマークプログラムのメモリ消費量

このベンチマークプログラムでは、可変性の指定を削除した参照の割合が増加するにつれてメモリ消費量が増加する傾向が観測された。加えて、特定の組み合わせで可変性の指定を削除した場合にメモリ消費量の増加が顕著になる傾向が見られた。これは、towers ベンチマークプログラムにおいては、ライフタイムがプログラムの実行全体にわたる参照が多く存在し、その参照に対して繰り返し再借用が行われることによって、参照の木構造に多くのノードが生成されるためであると考えられる。多くのベンチマークでは、問題サイズを大きくするために同じ計算を繰り返すため、参照の寿命は一回のイテレーション内に収まることが多い。一方、ハノイの塔の問題を解く towers ベンチマークでは、ディスクの枚数を増やすことで問題サイズを大きくすることができるため、プログラムの実行全体を通して生存する参照が多く存在する。そのため、プログラムの実行が長くなるにつれ、参照の木構造におけるノード数が増加し、メモリ消費量の増加につながっていると考えられる。この問題を解決するためには、参照の木構造に含まれるノードのうち、すでにライフタイムが終了して利用されなくなったノードを削除する仕組みを導入することが考えられる。

4.2.3 実行時間に占める実行時検査の時間

それぞれのベンチマークプログラムについて、可変性の指定を削除した参照の組み合わせを変化させた場合の、プログラム全体の実行時間に占める実行時検査にかかる時間の変化を測定した。実行時間を計測するために使用したのと同じパラメータを選択し、それぞれの可変性

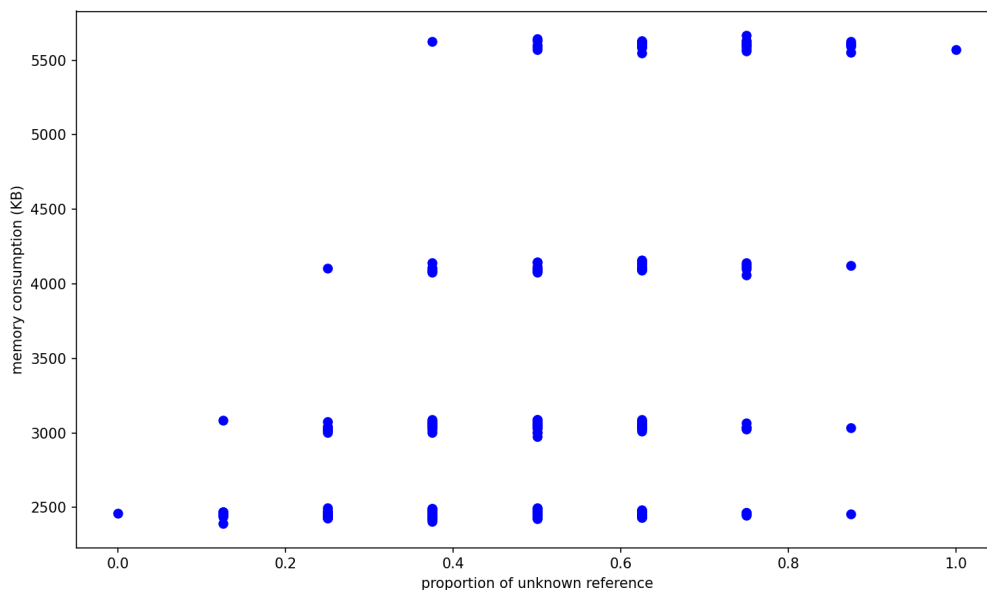


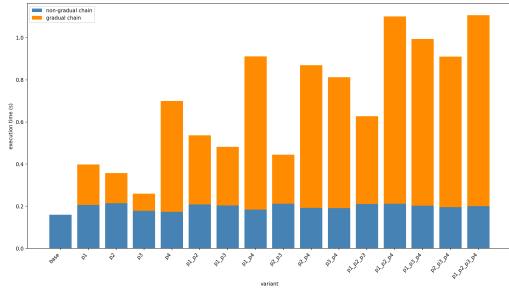
図 4.10: towers ベンチマークのメモリ消費量

の指定の有無の組み合わせごとに各プログラムを1回ずつ実行した。実行時検査にかかる時間としては、実行時検査器のライブラリに含まれる関数群に加え、実行時検査のために必要な参照の木構造が保持されるメモリ領域の確保と解放にかかる時間を計上した。perf コマンドを用いてプログラム実行中の特定の時点で実行されている関数をサンプリングし、プログラム全体の実行時間に占める割合から、実行時検査にかかる時間を推定した。

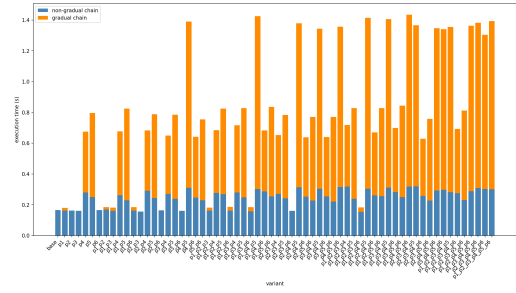
次に示すグラフ中の variants は、可変性の指定を削除する参照の組み合わせを表す。base はすべての参照に可変性が指定されているオリジナルの Rust プログラムを表し、p1 や p5 はそれぞれ1番目または5番目の参照の可変性を削除したプログラムを表す。複数の数字が含まれる場合は、それらの組み合わせの参照の可変性を削除したプログラムを表す。例えば、p2_p3_p5 は2番目、3番目、5番目の参照の可変性を同時に削除したプログラムを表す。

それぞれのベンチマークプログラムにおける、プログラムの実行時間に占める実行時検査にかかる時間の測定結果を図 4.11 に示す。4.2.1 節で述べたように、多くのベンチマークにおいて、可変性の指定を削除した参照の割合が増加するにつれて、実行時間が増加する傾向が観測された。このとき、直観的には、実行時間の増加に寄与するのは実行時検査にかかる時間であると考えられる。しかしながら、実際には実行時検査にかかる時間だけでなく、元のベンチマークプログラムの実行にかかる時間も増加していることが分かる。この傾向は、特に fannkuch-redux ベンチマーク (図 4.11a) や n-body ベンチマーク (図 4.11b) で顕著に見られるが、他のベンチマークにおいても同様の傾向が観測される。

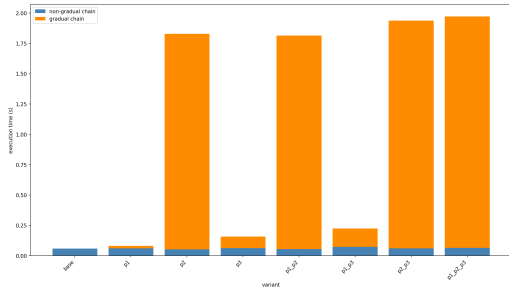
この現象には、複数の原因が考えられる。一つは、実行時検査器のライブラリ関数を呼び出すことによるオーバーヘッドである。通常、Rust における参照の再借用は、実行時にはメモリアドレスのコピーによって実装される。これは単一の命令で実行可能であり、場合によってはコンパイラの最適化により省略されることもある。一方、実行時検査を伴う参照の再借用に



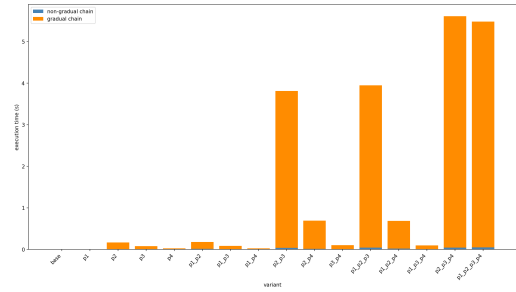
(a) fannkuch-redux ベンチマーク



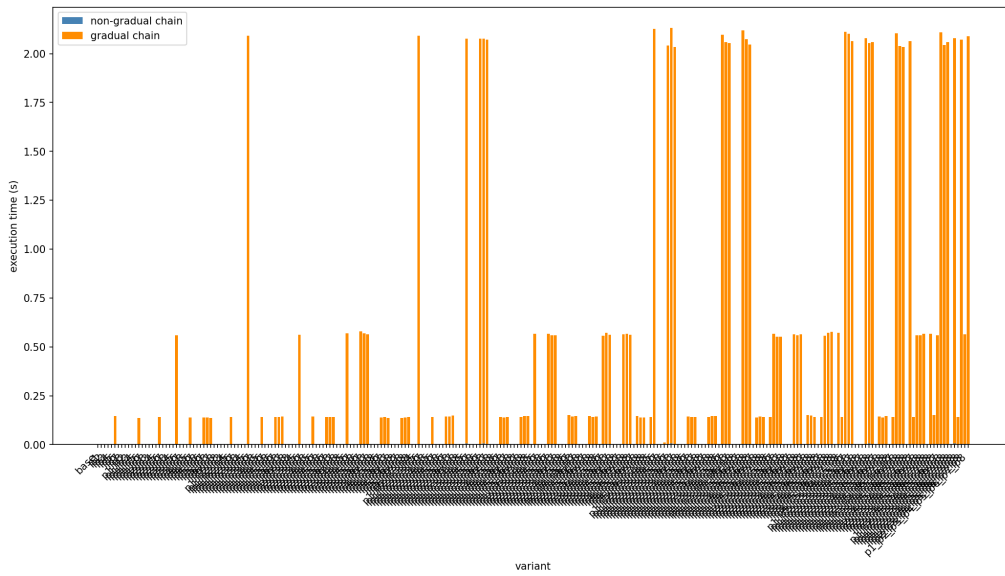
(b) n-body ベンチマーク



(c) bounce ベンチマーク



(d) queens ベンチマーク



(e) towers ベンチマーク

図 4.11: 各ベンチマークプログラムにおける実行時間に占める実行時検査の時間

は複数の関数呼び出しが必要であり、そのためには、レジスタの値をスタックに退避しなければならない場合がある。また、実行時検査は例外を発生させる可能性があるため、Rust コンパイラは例外処理のためのコードを生成する必要がある。加えて、関数呼び出しが分岐予測ミスやキャッシュミスを引き起こす可能性もあり、これらの要因が組み合わさることで、元のベンチマークプログラムの実行にかかる時間が増加することが考えられる。

もう一つの要因は、参照の可変性が削除されることによる最適化の阻害である。Rust コンパイラは、参照に指定される可変性の情報を利用して、プログラムの最適化を行う。例えば、共有参照を使用する場合、コンパイラはその参照が指し示すデータが変更されないことを前提に、高度な最適化を適用できる。漸進的借用検査では、参照の可変性の指定が省略された場合に Rust コンパイラが誤った最適化を適用しないようにするために、可変性の指定が削除された参照を可変参照と同様に扱うように指示する。これにより、コンパイラが適用できる最適化の範囲が制限され、元のベンチマークプログラムの実行にかかる時間が増加する可能性がある。

4.3 Rust コンパイラの拡張の実現可能性

漸進的借用検査を導入し、参照における可変性の指定を省略することができる Rust 言語を実現するための GRust コンパイラの実装について、その実現可能性を検討する。GRust プログラムを通常の Rust プログラムへと変換する GRust コンパイラの主な役割は、静的な型検査及び借用検査によって得られる情報を用いて、参照のライフタイムの開始と終了の位置に、実行時検査器のライブラリ関数の呼び出しを挿入することである。

Rust コンパイラを拡張することで GRust コンパイラを実装することは、Rust コンパイラに実装された静的な型検査及び借用検査の機能を再利用できることから、現実的であると考えられる。GRust コンパイラには、可変性が静的に記述された部分に対する静的な借用検査の機能や、参照のライフタイムを解析する機能が必要であるが、Rust コンパイラを拡張する場合には、これらの機能を新たに実装する必要はない。一方、独自の言語処理系を実装する場合や、他の言語処理系を拡張する場合には、これらの機能を一から実装する必要がある。しかしながら、借用検査を備えた言語処理系は非常に限られており、借用検査のための複雑なアルゴリズムを新たに実装することは困難であると考えられる。

GRust コンパイラは、Rust コンパイラの間中表現である MIR を利用して参照のライフタイムを解析し、MIR 中の行に対応する GRust プログラムのソースコードの位置を特定することで、実行時検査器のライブラリ関数の呼び出しを適切な位置に挿入する。実装した GRust コンパイラが、小規模の簡単な GRust プログラムについて自動的に実行時検査器のライブラリ関数の呼び出しを挿入し、実行可能な Rust プログラムを生成できることを確認した。しかしながら、一定の規模を超える複雑な GRust プログラムに対しては、正しく実行時検査器のライブラリ関数の呼び出しを挿入できない場合があることも確認した。

複雑な GRust プログラムに対して GRust コンパイラを正しく動作させることが難しい原因として、MIR を解析した結果を GRust プログラムのソースコードの位置に正しく対応付け

```
1 { let __tmp = EXPR; std::mem::drop(REF); __tmp }
```

図 4.12: drop 関数の呼び出しを挿入するコードスニペット

ることが困難であることが挙げられる。借用検査やコンパイラ最適化を行うために生成される MIR は、データフロー解析を行いやすいように、非常に単純かつ低レベルな手続きを表す文によって構成されている。したがって、高レベルな言語機能を持つ GRust プログラムを MIR に変換すると、単一の GRust の文から複数の MIR の文が生成されたり、GRust プログラムには含まれない多くの補助的な変数が MIR に導入されたりする。そのため、MIR の文から GRust プログラムのソースコードの位置を特定することが困難となり、適切な位置に実行時検査器のライブラリ関数の呼び出しを挿入できない場合がある。

参照の再借用を実行時検査器のライブラリ関数の呼び出しに変換する際には、Rust コンパイラによって自動的に挿入される暗黙的な再借用の扱いが問題となる場合がある。Rust コンパイラは、関数の呼び出しに参照が用いられる場合や、可変性の異なる変数間での代入が行われる場合などに、必要に応じて暗黙的な再借用を自動的に挿入する。この再借用は型検査の過程で挿入され、MIR 上では明示的な再借用の文として表現される。そのため、GRust コンパイラは MIR に含まれる再借用の文を検出することで、暗黙的な再借用を含むすべての再借用の位置を特定することができる。しかしながら、特定した再借用が GRust プログラムのソースコード上に明示的に出現するかどうかを判別することは困難であり、保守的にすべての再借用に対して実行時検査器のライブラリ関数の呼び出しを挿入すると、生成された Rust プログラム上でコンパイルエラーが発生する場合がある。コンパイルエラーが生じた場合でも、単純な型の不整合であれば Rust コンパイラによって提案される修正候補によって自動的に修正できる場合があるが、誤った修正候補が提示される場合や、修正候補が提示されない場合もある。

参照のライフタイムの終了の位置に drop 関数の呼び出しを挿入することには、さらなる課題が存在する。参照のライフタイムの終了とはその参照の最後の使用箇所を指すため、任意の文においてライフタイムの終了が発生する可能性がある。さらに、文が複数の式によって構成される場合には、文の途中であっても特定の式の評価が完了した時点でライフタイムの終了が発生する可能性がある。できる限り多くの GRust プログラムに対して正しく drop 関数の呼び出しを挿入するため、GRust コンパイラは図 4.12 に示すようなコードスニペットを生成する。このコードスニペットは、式 EXPR を評価した結果を一時変数 __tmp に格納し、参照 REF のデストラクタを呼び出した後に、一時変数 __tmp の値を返すことで、式の評価結果を保持しながら、可能な限り早く参照のライフタイムを終了させる。

このようなコードスニペットの適用を困難にするのは、参照がソースコード上で明示的な識別子を持たない場合や、複数の参照のライフタイムが単一の式によって終了する場合である。MIR を生成する過程で導入された補助変数に束縛されている参照はソースコード上で識別子を持たないため、drop 関数の引数として渡すことができない。また、複数の参照のライフタイムが単一の式において同時に終了する場合には、drop 関数の呼び出しを複数回挿入する必

要があるが、GRust コンパイラはそのような場合に適切にコードスニペットを生成できない。このような問題により、任意の GRust プログラムに対して適切なコード変換を行うことは困難である。

GRust プログラムから既存の Rust ライブラリを利用するにあたっては、Rust における通常の参照型と、実行時検査器のライブラリが提供する独自のデータ構造による参照型の間には互換性がないことが問題となる。通常の参照型を期待する Rust ライブラリの関数に対して可変性の指定が省略された参照を経由して生成された参照を渡すためには、実行時検査器のライブラリが提供する参照型を受け付けるようにライブラリの実装を変更しなければならない。ライブラリの実装を変更することなく、通常の参照型と実行時検査器のライブラリが提供する参照型を混ぜて使用した際には、Rust コンパイラによって型エラーが報告される。ライブラリの型定義を修正する作業は複雑であるため、この型エラーを自動的に修正することは困難である。この問題を解決するためには、メモリ消費量に一定のオーバーヘッドが発生することを許容した上で、漸進的借用検査を前提とした参照型のみを使用する言語設計を採用することが望ましい。

漸進的借用検査が Gradual Guarantee を満たすためには、可変性の指定が省略された参照 `&mut T` の被参照型 `T` が共変であることを用いた型キャストを行った場合に、その参照を通じて書き込みが行われないことを保証する追加の実行時検査を導入する必要がある。この型キャストは、単純な参照型だけでなく、関数型の引数や戻り値の型としても現れる可能性がある。一般に、漸進的型付けを導入した言語で同様の実行時検査を導入する場合には、多くの漸進的型付けの言語が動的プログラミング言語の特徴を持つことから、ラッパー関数を生成して型キャストを実装することが多い。しかしながら、静的な Rust プログラムを生成する GRust コンパイラにおいては、任意の型キャストに対してラッパー関数を生成することは困難である。したがって、漸進的借用検査を Rust 言語に導入する場合には、Gradual Guarantee に一定の制限を設けることが必要になると考えられる。

第 5 章

おわりに

本論文では、部分的に可変性の指定が省略された参照を含むプログラムを、データ競合を引き起こすことなく安全に実行することができる、漸進的借用検査のアルゴリズムを提案した。提案手法を用いることで、実行時にメモリ安全性を侵害しない場合に限り、静的な借用検査を通過しない箇所を含むプログラムであっても安全に実行することができる。これによって、新しく Rust を導入しようとするプログラマであっても、可変性の指定を省略しながら Rust プログラムを迅速にプロトタイピングすることが可能となる。さらに、段階的に可変性の指定を追加することでプログラムの安全性と実行性能を向上させ、最終的に完全に可変性が指定された Rust プログラムへと移行することを可能にする。

漸進的借用検査は、可変性の指定が省略された参照について、実行時に Aliasing XOR Mutability の規則が満たされることを検査する。この実行時検査は、参照の再借用の関係に基づいて構築される参照の木構造の上で、それぞれの参照が持ちうる権限の範囲を計算することによって実現される。また、この漸進的借用検査を Rust 言語に統合するためには、Rust コンパイラの拡張と、Rust ライブラリとして実装された実行時検査器が必要となる。

漸進的借用検査における実行時検査による実行時間のオーバーヘッドは、可変性の指定が省略された参照の割合が高いほど大きくなることを確認した。これは、可変性の指定を省略しながら記述したプログラムに対して、可変性の指定を段階的に追加することで実行性能を向上させることが可能であることを示している。また、Rust コンパイラの拡張によって漸進的借用検査を導入したプログラミング言語を実現することには、いくつかの技術的な課題が存在することも明らかとなった。

今後の課題としては、手動でのコード変換を行うことなく任意のプログラムを実行することができる、漸進的借用検査を導入した言語を実現することが挙げられる。特に、既存の Rust コンパイラを拡張して漸進的借用検査を導入する手法が、漸進的借用検査を前提として設計された独自の言語処理系を開発する手法と比較して合理的であるかどうかには、検討の余地がある。

また、実行時検査のアルゴリズムを最適化し、実行時間のオーバーヘッドを低減させることも重要な課題である。再借用が行われるたびに参照の木構造に含まれるすべての参照の権限を再計算する現在のアルゴリズムは、実行時間とメモリ消費量の双方の観点から非効率である。

46 第5章 おわりに

この問題を改善する方法として、参照の木構造の探索を自身の親方向のみに限定することが考えられるが、アルゴリズムの正当性を維持しつつ効率化を実現するためにはさらなる検討が必要である。

発表文献と研究活動

- (1) 両角 颯, 山崎 徹郎, 千葉 滋. 漸進的借用検査を導入した Rust 言語処理系の検討. 日本ソフトウェア科学会第 42 回大会. 2025.9.3-5.

参考文献

- [1] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. *LIPICs, Volume 32, SNAPL 2015*, Vol. 32, pp. 274–293, 2015.
- [2] John Boyland. Checking Interference with Fractional Permissions. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Radhia Cousot, editors, *Static Analysis*, Vol. 2694, pp. 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
- [3] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*, pp. 32–41, St. Petersburg Beach, Florida, United States, 1996. ACM Press.
- [4] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, Copenhagen, Denmark. IEEE Comput. Soc.
- [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, pp. 1–34, January 2018. Publisher: Association for Computing Machinery (ACM).
- [6] Nicholas D. Matsakis. Non-lexical lifetimes: introduction. <https://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/>, April 2016. Accessed: 2025-12-31.
- [7] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, Vol. 28, p. e20, 2018.
- [8] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing, 2018. Version Number: 2.
- [9] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, Vol. 4,

- No. POPL, pp. 1–32, January 2020.
- [10] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. Tree Borrows. *Proceedings of the ACM on Programming Languages*, Vol. 9, No. PLDI, pp. 1019–1042, June 2025.
 - [11] Andrew Wagner, Olek Gierczak, Brianna Marshall, John M. Li, and Amal Ahmed. From Linearity to Borrowing. *Proceedings of the ACM on Programming Languages*, Vol. 9, No. OOPSLA2, pp. 3981–4007, October 2025.
 - [12] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pp. 48–59, Pittsburgh PA USA, September 2002. ACM.
 - [13] The Rust Project Developers. The MIR (mid-level IR) - Rust Compiler Development Guide. <https://rustc-dev-guide.rust-lang.org/mir/index.html>, 2018. Accessed: 2026-01-09.
 - [14] Stefan Marr, Benoit Daloz, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS’16, pp. 120–131. ACM, November 2016.

謝辞

本研究を進めるにあたり、研究室の皆様をはじめとする多くの方々にご支援とご助言を賜りました。まずはじめに、指導教員である千葉滋教授には、個別での議論の機会を頻繁に設けていただき、研究成果を発表するための心得や論文執筆の指導など、多大なるご指導を賜りました。また、山崎徹郎先生には、アルゴリズムの設計や実装に関して多くの有益な助言をいただきました。さらに、研究室の皆様、特に同期の皆様からは、研究室での議論を通じて多くの刺激を受け、積極的に研究に取り組む意欲を得ることができました。最後に、自由な環境で研究に専念できるよう支えてくれた家族を含め、すべての方々に心より感謝申し上げます。

