

2023 年度 修士論文

Rust への Fractional Ownership の
動的検査の導入

東北大学 大学院情報科学研究科
情報基礎科学専攻

C2IM1034 馬場 風汰

指導教員：住井 英二郎 教授

2024 年 2 月 7 日 10:50–11:50
オンライン

要旨

Rust は ownership (所有権) の概念に基づいて安全な静的メモリ管理を行うプログラミング言語である。Rust では、各オブジェクトに唯一の owner が静的に定められている。ownership は、変数への代入や関数に引数を渡す際に移動する。オブジェクトのメモリ領域は、owner のスコープが終了すると解放される。

しかし、静的な ownership のみによるメモリ管理は柔軟性に欠ける場合がある。そのため Rust には Rc (参照カウント) オブジェクトも存在する。Rc オブジェクトはコンテナであり、中身のオブジェクトへの参照を持つ。Rc オブジェクトは clone 関数により複製でき、その際共通の参照カウントを 1 増やす。複製が消滅する際には参照カウントを 1 減らし、参照カウントが 0 になったら中身のオブジェクトを解放する。そのため、意図せず複製が残っているとメモリリークのおそれがある。

そこで本研究では fractional ownership [Boyland 2003] の考え方を取り入れて、Rust の静的な ownership によるメモリ管理と、動的な参照カウントの柔軟性を組み合わせる。fractional ownership とは 0 以上かつ 1 以下の有理数であり、オブジェクト生成時には ownership として 1 を与え、参照複製時には ownership を分割し、複製された参照が消滅する際には ownership を集約する。また、オブジェクトの解放には ownership 1 が必要であり、逆に ownership が 0 より大きいオブジェクトはいずれ解放される必要がある。本来 fractional ownership は静的に検査されるが、本研究では参照カウントを動的な fractional ownership とみなす方式を提案・実装する。これにより、動的な参照カウントの柔軟性を活用した後、参照カウントが 1 になったオブジェクトは fractional ownership 1 とみなし、Rust の静的な ownership に戻すことができる。逆に、静的な ownership に戻さないまま参照が消滅したら実行時エラーとすることにより、オブジェクトの解放し忘れを検出することができる。

謝辞

本論文の執筆にあたりご指導くださった住井英二郎教授に感謝申し上げます。ゼミや発表でご指摘くださった松田一孝准教授、Oleg Kiselyov 助教、住井・松田研究室の皆様に感謝申し上げます。

目次

謝辞	iii
第 1 章 序論	1
1.1 背景	1
1.2 目的	1
1.3 本論文の構成	1
第 2 章 背景	2
2.1 Rust の ownership	2
2.2 Rust の参照カウント	5
2.3 静的な fractional ownership	6
第 3 章 問題点	7
第 4 章 本研究の提案	10
4.1 提案するインターフェース	10
4.2 実装	14
第 5 章 並列処理への応用	18
5.1 Rust の並列処理	18
5.2 並列処理の際の問題点	19
5.3 新たな参照オブジェクトの応用	20
第 6 章 関連研究	21
第 7 章 結論と今後の課題	22
参考文献	23

第 1 章

序論

1.1 背景

Rust [1] は、ownership [2] の概念に基づき静的なメモリ管理を行うプログラミング言語であり、システムプログラミング等に用いられる。Rust 登場以前は、メモリ管理はユーザが手動で管理する方式と実行時ガベージコレクタを用いた方式の 2 つが主なものであった。しかし Rust のメモリ管理は、手動のメモリ管理と異なり、解放し忘れによるメモリリークなどをコンパイル時に検出でき、さらに、実行時ガベージコレクタを用いたメモリ管理より効率が良い。そのため、ユーザはメモリ管理という低レベルの制御をこれまでの方式と比べて楽に、効率よく行うことができる。しかし、Rust のメモリ管理は静的であり、動的なメモリ管理と比べると柔軟性に欠ける場合がある。そのため Rust には、ownership の規則を無視することができるオブジェクトとして Rc オブジェクトなどが存在する。しかし、Rc オブジェクトを利用した際、意図せず複製が残っている場合があり、メモリリークのおそれがある。

1.2 目的

本研究では、fractional ownership [Boyland 2003] の考え方を取り入れて、Rust の静的な ownership によるメモリ管理と、動的な参照カウントの柔軟性を組み合わせ、ユーザの意図していないメモリリークを防ぐ方式を提案・実装することを目的とする。

1.3 本論文の構成

本論文では、まず第 2 章で Rust の ownership、Rc オブジェクト、fractional ownership について述べた後、第 3 章で Rc オブジェクトを用いる際の問題点について述べた後、第 4 章で本研究で提案する新たな参照オブジェクトについて述べる。その後第 5 章で新たな参照オブジェクトの並列処理への応用について述べ、第 6 章で関連研究、第 7 章で結論と今後の課題を述べる。

第 2 章

背景

2.1 Rust の ownership

2.1.1 ownership

Rust は ownership という概念に基づいて安全な静的メモリ管理を行っている。Rust では、各オブジェクトに唯一の owner が静的に定められており、オブジェクトのメモリ領域は、owner のスコープが終了すると開放される。実際の例を以下の図 2.1 に示す。3 行目で変数 `s` は Hello という文字列を保持する String オブジェクトの owner となり、5 行目で変数 `t` は World という文字列を保持する String オブジェクトの owner となる。その後 6 行目で、`t` のスコープが終了するため、World という文字列を保持する String オブジェクトのメモリ領域が先に解放される。7 行目で、`s` のスコープが終了するため、World という文字列を保持する String オブジェクトのメモリ領域が解放される。

```
1 fn main() {  
2     {  
3         let s = String::from("Hello");  
4         {  
5             let t = String::from("World");  
6         }  
7     }  
8 }
```

図 2.1 ownership によるメモリ管理

2.1.2 move

Rust では、変数への代入や関数に引数を渡す際に ownership が移動する。これを move と呼ぶ。変数への代入の際には以下の図 2.2 のように move が起きる。2 行目で、変数 `s` が String オブジェクトの owner となり、3 行目で `s` から `new_s` に ownership が move している。ownership が

move したことで `s` は owner でなくなるため、4 行目で `s` を利用しようとするコンパイルエラーが発生する。

```
1 fn main() {
2     let s = String::from("Hello");
3     let new_s = s;
4     s.push_str("World"); // Compile error!
5 }
```

図 2.2 変数への代入で move が起こる例

また、関数に引数を渡す際には以下の図 2.3 のように move が起きる。6 行目で、変数 `s` が String オブジェクトの owner となり、7 行目で `s` から関数 `f` の引数 `x` に ownership が move している。ownership が move しているため、`x` は String オブジェクトの owner となり、2 行目の処理を行うことができる。関数 `f` の処理が終了した後、3 行目で引数 `x` の指す String オブジェクトが解放される。そのため、8 行目の時点で `s` は owner でないためコンパイルエラーが発生する。さらに、8 行目ですでに解放されている String オブジェクトに対して処理を行おうとしているため、実際に危険な処理となっている。

```
1 fn f(mut x: String) {
2     x.push_str("World");
3 }
4
5 fn main() {
6     let mut s = String::from("Hello");
7     f(s);
8     s.push_str("Oops!"); // Danger!
9 }
```

図 2.3 関数に引数を渡す際に move が起こる例

図 2.3 のプログラムは、以下の図 2.4 のように関数 `f` が引数 `x` を ownership ごと返却するように変更することで危険な処理を避けることができる。しかし、ownership の返却を手動で行うことは、返却を忘れてしまうと危険な処理に繋がるおそれがあり、不便である。

```
1 fn f(mut x: String) -> String{
2     x.push_str("World");
3     return x
4 }
5
6 fn main() {
7     let mut s = String::from("Hello");
8     let mut return_s = f(s);
9     return_s.push_str("Oops!"); // Danger!
10 }
```

図 2.4 危険な処理を避けた例

2.1.3 borrowing

move によって生じる不便を解消するため、Rust には ownership を一時的に借り、自動で返却する仕組みがある。ownership を借りる行為は borrowing と呼ばれ、Rust に特有の「参照」を作成することで借りることができる。borrowing は、読み書き可能である mutable な borrowing と読み取りのみ可能である immutable な borrowing に区別されており、変数名の前に `&mut` とつけることで mutable な borrowing が、`&` とつけることで immutable な borrowing ができる。ownership を borrowing している例を以下の図 2.5 に示す。6 行目で、変数 `s` が String オブジェクトの owner となり、7 行目で `s` の ownership が一時的に関数 `f` の引数 `x` に貸しだされる。ownership が貸しだされているため、`x` は String オブジェクトの owner となり、2 行目の処理を行うことができる。関数 `f` の処理が終了した後、`x` に貸しだされていた ownership は自動で変数 `s` に返却される。そのため、8 行目で `s` は owner であり、`s` に対して処理が可能である。

```
1 fn f(x: &String) {  
2     println!("{}", &x);  
3 }  
4  
5 fn main() {  
6     let mut s = String::from("Hello");  
7     f(&s);  
8     s.push_str("World");  
9 }
```

図 2.5 borrowing の例

2.1.4 lifetime

borrowing によって貸し出された ownership は、lifetime という仕組みを利用して自動で返却される。lifetime とは、参照を用いることができる範囲のことであり、基本的に自動で推論される。lifetime は、以下の図 2.6 のように、明示的に注釈をつけることもできる。1 行目のように書くことで、引数 `x` の lifetime が関数 `f` の本体と一致していると注釈でき、関数 `f` の処理が終了したとき、`x` の lifetime も終了するため、その時点で ownership が返却される。

```
1 fn f<'a>(x: &'a String) {  
2     println!("{}", &x);  
3 }  
4  
5 fn main() {  
6     let mut s = String::from("Hello");  
7     f(&s);  
8     s.push_str("World");  
9 }
```


2.2 Rust の参照カウント

Rust の静的な ownership のみによるメモリ管理は柔軟性に欠ける場合があるため、Rust には Rc (参照カウント) オブジェクトも存在する。Rc オブジェクトはコンテナであり、中身のオブジェクトへの参照と、共通の参照カウントを持つ。また、clone 関数により複製でき、その際共通の参照カウントを 1 増やす。複製が消滅する際には参照カウントを 1 減らし、参照カウントが 0 になったら中身のオブジェクトを解放する。Rc オブジェクトの実装には、Rust の静的な ownership を無視する unsafe な機能が用いられている。

Rc オブジェクトは、以下の図 2.7 のように用いられる。このプログラムは、中身に Hello という String オブジェクトを持つ Rc オブジェクト h と、中身に World という String オブジェクトを持つ Rc オブジェクト w の複製をランダムに 10 回配列に入れ、その配列から Hello という String オブジェクトを持つ Rc オブジェクトを削除するプログラムである。配列の要素は実行する度に变化するため、静的な ownership のみではメモリ管理が難しい例である。12 行目と 16 行目では clone 関数により、Rc オブジェクトの複製を生成し、参照カウントが 1 増えている。20 行目で h の複製は 1 つを残してすべて消滅し、h の参照カウントは 1 まで減少する。最後に 21 行目で h、w の参照カウントは 0 となり、中身のオブジェクトが解放される。

```

1  fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
2      vec.retain(|x| **x != s);
3  }
4
5  fn main() {
6      let mut vec: Vec<Rc<String>> = Vec::new();
7      let h = Rc::new(String::from("Hello"));
8      let w = Rc::new(String::from("World"));
9      let mut rng = rand::thread_rng();
10     for _ in 0..10 {
11         if rng.gen() {
12             let h_clone = h.clone();
13             vec.push(h_clone);
14         }
15         else {
16             let w_clone = w.clone();
17             vec.push(w_clone);
18         }
19     }
20     remove_string(&mut vec, String::from("Hello"));
21 }

```

図 2.7 Rc オブジェクトの利用例

2.3 静的な fractional ownership

一方、Rust の ownership とは別に fractional ownership という概念が存在する。fractional ownership とは 0 以上かつ 1 以下の有理数であり、オブジェクト生成時には ownership として 1 を与え、参照複製時には ownership を分割し、複製された参照が消滅する際には ownership を集約する。また、オブジェクトの解放には ownership 1 が必要であり、逆に ownership が 0 より大きいオブジェクトはいずれ解放される必要がある。

第3章

問題点

図 3.1 のプログラムは、中身に Hello という String オブジェクトを持つ Rc オブジェクト `h` と、中身に World という String オブジェクトを持つ Rc オブジェクト `w` の複製をランダムに 10 回配列に入れ、その配列から Hello という String オブジェクトを持つ Rc オブジェクトを削除した後、他の処理を行うプログラムである。21 行目にその他の処理として、`vec` は用いるが、Rc オブジェクト `h` と Rc オブジェクト `w` を用いない処理が行われているとする。このとき、2 つの Rc オブジェクトの中身が解放されるのは 22 行目で `main` 関数のスコープが終了したときであり、追加した新たな処理で `h` も `w` もどちらも利用していないにもかかわらず `h` のメモリ領域が確保されたままとなってしまう。このように、Rc オブジェクトを用いた場合、意図せずメモリリークが発生するおそれがある。

```
1 fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<Rc<String>> = Vec::new();
7     let h = Rc::new(String::from("Hello"));
8     let w = Rc::new(String::from("World"));
9     let mut rng = rand::thread_rng();
10    for _ in 0..10 {
11        if rng.gen() {
12            let h_clone = h.clone();
13            vec.push(h_clone);
14        }
15        else {
16            let w_clone = w.clone();
17            vec.push(w_clone);
18        }
19    }
20    remove_string(&mut vec, String::from("Hello"));
```

```

21 // その他の処理
22 }

```

図 3.1 Rc オブジェクトを利用する際の問題点

以下の図 3.2 のプログラムは、22 行目で `h` と `w` のスコープが終了するため `h` のメモリ領域が解放されるため、図 3.1 のような意図していないメモリリークを防いだプログラムである。

```

1 fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<Rc<String>> = Vec::new();
7     {
8         let h = Rc::new(String::from("Hello"));
9         let w = Rc::new(String::from("World"));
10        let mut rng = rand::thread_rng();
11        for _ in 0..10 {
12            if rng.gen() {
13                let h_clone = h.clone();
14                vec.push(h_clone);
15            }
16            else {
17                let w_clone = w.clone();
18                vec.push(w_clone);
19            }
20        }
21        remove_string(&mut vec, String::from("Hello"));
22    }
23    // その他の処理
24 }

```

図 3.2 メモリリークを防いだ例

しかし、図 3.3 のように、スペルミスにより `Hello` ではなく `Hell` という文字列を持つ `String` オブジェクトを削除してしまった場合、21 行目で配列から `Hello` という文字列を保持する `String` オブジェクトが削除されず、22 行目で `h` のメモリ領域が解放されないためメモリリークが起きてしまう。

```

1 fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<Rc<String>> = Vec::new();

```

```

7      {
8          let h = Rc::new(String::from("Hello"));
9          let w = Rc::new(String::from("World"));
10         let mut rng = rand::thread_rng();
11         for _ in 0..10 {
12             if rng.gen() {
13                 let h_clone = h.clone();
14                 vec.push(h_clone);
15             }
16             else {
17                 let w_clone = w.clone();
18                 vec.push(w_clone);
19             }
20         }
21         remove_string(&mut vec, String::from("Hell"));
22     }
23     // その他の処理
24 }

```

図 3.3 Rc オブジェクトを利用する際の問題点

このように、ユーザが意識すればメモリリークを防ぐことも可能ではあるが、メモリリークが起きないことを保障することはできないため、ユーザのミスにより意図していないメモリリークがおきてしまうことがあり得る。

第 4 章

本研究の提案

本研究では、Rust の静的な ownership によるメモリ管理と、動的な参照カウントの柔軟性を組み合わせた新たな参照オブジェクトを提案する。この参照オブジェクトは、fractional ownership の考え方を取り入れている。

4.1 提案するインターフェース

Rc オブジェクトを用いた際に、メモリリークが起き得るという問題を解決するために、以下の図 4.1 のように用いることができる参照カウントを提案する。このプログラムは、中身に Hello という String オブジェクトを持つ mutable な参照オブジェクト h と、中身に World という String オブジェクトを持つ mutable な参照オブジェクト w をどちらも immutable な参照オブジェクトに変換した後、複製をランダムに 10 回配列に入れ、その配列から Hello という String オブジェクトを持つ immutable な参照オブジェクトを削除するプログラムである。7、8 行目で生成した mutable な参照を、10、11 行目で immutable な参照に変換し、14、19 行目で immutable な参照を複製している。remove_string が終了した後、26 行目、32 行目で mutable な参照に戻している。

```
1 fn remove_string(vec: &mut Vec<RefImmut<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<RefImmut<String>> = Vec::new();
7     let mut h = RefMut::new(String::from("Hello"));
8     let mut w = RefMut::new(String::from("World"));
9     let mut rng = rand::thread_rng();
10    let h_immut = h.to_immut();
11    let w_immut = w.to_immut();
12    for _ in 0..10 {
13        if rng.gen() {
14            let h_immut1 = h_immut.clone_immut();
```

```

15         println!("{}", h_immutable);
16         vec.push(h_immutable);
17     }
18     else {
19         let w_immutable = w_immutable.clone_immutable();
20         println!("{}", w_immutable);
21         vec.push(w_immutable);
22     }
23 }
24
25 remove_string(&mut vec, String::from("Hello"));
26 let mut h_mut = h_immutable.back_to_mutable();
27 h_mut.push_str("hoge");
28 println!("{}", h_mut);
29 drop(h_mut);
30
31 drop(vec);
32 let mut w_mut = w_immutable.back_to_mutable();
33 }

```

図 4.1 新たな参照オブジェクトを用いた柔軟なメモリ管理の例

この参照オブジェクトは、読み書き可能である `mutable` な参照オブジェクトと読み取りのみ可能である `immutable` な参照オブジェクトの 2 つがあり、静的に区別されている。`mutable` な参照は Rust の通常の `ownership` に従い、同時に一つしか存在しない。`to_immutable` 関数を用いると、`mutable` な参照を `immutable` な参照に変換でき、その際 Rust の静的な `ownership` の `move` により `mutable` な参照は消滅する。`immutable` な参照は、Rust の通常の `ownership` に違反して `clone_immutable` 関数により複製が可能であり、複製の数を動的にカウントする。参照カウントが 1 つであるときのみ、`back_to_immutable` 関数を用いて `immutable` な参照から `mutable` な参照へ変換でき、その際 `move` により `immutable` な参照は消滅する。`immutable` な参照から `mutable` な参照に戻し忘れると、実行時エラーが発生するため、オブジェクトの解放し忘れが検出できる。

以下の図 4.2 は、図 4.1 とほぼ同じプログラムであるが、`immutable` な参照を `mutable` な参照に戻していないプログラムである。この場合、`mutable` な参照に戻さないまま参照が消滅してしまっているため実行時エラーが発生し、オブジェクトの解放し忘れを検出することができる。

```

1 fn remove_string(vec: &mut Vec<RefImmutable<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<RefImmutable<String>> = Vec::new();
7     let mut h = RefMutable::new(String::from("Hello"));
8     let mut w = RefMutable::new(String::from("World"));
9     let mut rng = rand::thread_rng();

```

```

10     let h_immutable = h.to_immutable();
11     let w_immutable = w.to_immutable();
12     for _ in 0..10 {
13         if rng.gen() {
14             let h_immutable1 = h_immutable.clone_immutable();
15             println!("{}", h_immutable1);
16             vec.push(h_immutable1);
17         }
18         else {
19             let w_immutable1 = w_immutable.clone_immutable();
20             println!("{}", w_immutable1);
21             vec.push(w_immutable1);
22         }
23     }
24
25     remove_string(&mut vec, String::from("Hello"));
26 } // error!

```

図 4.2 オブジェクトの解放し忘れを検出する例

以下の図 4.3 は、図 4.1 とほぼ同じプログラムであるが、参照カウントが 1 でないにも関わらず immutable な参照を mutable な参照に戻そうとしているプログラムである。この場合、mutable な参照に戻すことができずに実行時エラーが発生するため、mutable な参照でないと消滅させることはできないようユーザに強制することができている。

```

1 fn remove_string(vec: &mut Vec<RefImmutable<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<RefImmutable<String>> = Vec::new();
7     let mut h = RefMutable::new(String::from("Hello"));
8     let mut w = RefMutable::new(String::from("World"));
9     let mut rng = rand::thread_rng();
10    let h_immutable = h.to_immutable();
11    let w_immutable = w.to_immutable();
12    for _ in 0..10 {
13        if rng.gen() {
14            let h_immutable1 = h_immutable.clone_immutable();
15            println!("{}", h_immutable1);
16            vec.push(h_immutable1);
17        }
18        else {
19            let w_immutable1 = w_immutable.clone_immutable();
20            println!("{}", w_immutable1);
21            vec.push(w_immutable1);

```



```

22     }
23 }
24
25 remove_string(&mut vec, String::from("Hello"));
26 let mut w_mut = w_immutable.back_to_mut(); // error!
27 w_mut.push_str("hhhhh");
28 println!("{}", w_mut);
29 }

```

図 4.3 mutable な参照へ戻せないことを検出する例

以下の図 4.4 は、図 4.1 とほぼ同じプログラムであるが、文字列を削除する際にスペルミスをしてしまっているプログラムである。この場合、mutable な参照に戻ることができずに実行時エラーが発生するため、メモリリークを防ぐことができています。

```

1  fn remove_string(vec: &mut Vec<RefImmut<String>>, s: String) {
2      vec.retain(|x| **x != s);
3  }
4
5  fn main() {
6      let mut vec: Vec<RefImmut<String>> = Vec::new();
7      let mut h = RefMut::new(String::from("Hello"));
8      let mut w = RefMut::new(String::from("World"));
9      let mut rng = rand::thread_rng();
10     let h_immutable = h.to_immutable();
11     let w_immutable = w.to_immutable();
12     for _ in 0..10 {
13         if rng.gen() {
14             let h_immutable1 = h_immutable.clone_immutable();
15             println!("{}", h_immutable1);
16             vec.push(h_immutable1);
17         }
18         else {
19             let w_immutable1 = w_immutable.clone_immutable();
20             println!("{}", w_immutable1);
21             vec.push(w_immutable1);
22         }
23     }
24
25     remove_string(&mut vec, String::from("Hell"));
26     let mut h_mut = h_immutable.back_to_mut(); // error!
27     h_mut.push_str("hoge");
28     println!("{}", h_mut);
29     drop(h_mut);
30
31     // その他の処理

```

```

32
33     drop(vec);
34     let mut w_mut = w_immut.back_to_mut();
35 }

```

図 4.4 mutable な参照へ戻せないことを検出する例

4.2 実装

mutable な参照オブジェクト、immutable な参照オブジェクトは以下の図 4.5 のように実装した。mutable な参照オブジェクトは、オブジェクトを保持するコンテナとして実装した。immutable な参照オブジェクトは、Rust の ownership を無視する特別なポインタである NonNull オブジェクトを保持するオブジェクトとして実装した。NonNull オブジェクトにより、参照の複製が可能になる。immutable な参照内のポインタは NewRefInner オブジェクトへのポインタであり、このオブジェクトは参照カウントと中身のオブジェクトを保持する。

```

1 pub struct RefMut<T: ?Sized> {
2     data: T,
3 }
4 pub struct RefImmut<T: ?Sized> {
5     ptr: NonNull<NewRefInner<T>>
6 }
7 struct NewRefInner<T: ?Sized> {
8     ref_count: atomic::AtomicUsize,
9     data: T
10 }

```

図 4.5 新たな参照オブジェクトの実装

mutable な参照オブジェクトを生成する new 関数と mutable な参照オブジェクトを引数として受け取り、immutable な参照オブジェクトに変換して返す to_immut 関数は以下の図 4.6 ように実装した。4 行目で to_immut 関数の引数は、borrowing でなく move が起きるようにすることで、変換した後引数の mutable な参照が消滅するように実装した。

```

1 pub fn new(data: T) -> RefMut<T> {
2     Self { data: data }
3 }
4 pub fn to_immut(self: RefMut<T>) -> RefImmut<T> {
5     let mut this = ManuallyDrop::new(self);
6     let inner =
7         unsafe{ptr::read(Self::get_mut_unchecked(&mut this))};
8     let x: Box<_> = Box::new(NewRefInner {
9         ref_count: AtomicUsize::new(1),
10         data: inner,

```

```

11     });
12     RefImmut { ptr: Box::leak(x).into() }
13 }

```

図 4.6 関数 new と関数 to_immutable の実装

immutable な参照オブジェクトを引数として受け取り、複製した参照を返す clone_immutable 関数は次の図 4.7 ように実装した。clone_immutable 関数の引数を borrowing にすることで、複製元の参照が消滅せずに利用できるように実装した。また、2 行目の fetch_add 関数によって immutable な参照の内部の参照カウントが 1 増加するように実装した。

```

1 pub fn clone_immutable(&self: &RefImmut<T>) -> RefImmut<T> {
2     let old_size = self.inner().ref_count.fetch_add(1, Release);
3     if old_size > MAX_REFCOUNT {
4         abort();
5     }
6     RefImmut { ptr: self.ptr }
7 }

```

図 4.7 関数 clone_immutable の実装

immutable な参照オブジェクトを引数として受け取り、mutable な参照オブジェクトに変換し返す back_to_mutable 関数は次の図 4.8 のように実装した。back_to_mutable 関数の引数を borrowing でなく move するようにし、変換した後引数の immutable な参照が消滅するように実装した。また、2 行目で immutable な参照の参照カウントが 1 でなければ実行時エラーが起きるようにし、immutable な参照のままでは消滅できないように実装した。

```

1 pub fn back_to_mutable(self: RefImmut<T>) -> RefMut<T> {
2     if self.inner().ref_count.load(Acquire) != 1 {
3         panic!("cannot back to mutable");
4     }
5     let mut this = ManuallyDrop::new(self);
6     let inner =
7         unsafe { ptr::read(Self::get_mutable_unchecked(&mut this)) };
8     unsafe {
9         dealloc(this.ptr.cast().as_mutable(),
10             Layout::for_value_raw(this.ptr.as_ptr()))
11     }
12     RefMut { data: inner }
13 }

```

図 4.8 関数 back_to_mutable の実装

immutable な参照を引数として受け取り、その参照を消滅させる drop 関数は以下の図 4.9 のように実装した。参照カウントが 2 以上である場合は、参照カウントを 1 減少させ、参照カウントが 1 である場合は実行時エラーが起きるようにすることで、immutable な参照を mutable な参照に

戻さずに消滅できなように実装した。これにより、オブジェクトの解放し忘れを検出することが可能である。

```
1 fn drop(&mut self) {
2     let count = self.inner().ref_count.fetch_sub(1, Relaxed);
3     if count > 1{
4         return
5     }
6     else if count == 1{
7         if panicking() {
8             return
9         }
10        unsafe {
11            dealloc(self.ptr.cast().as_mut(),
12                Layout::for_value_raw(self.ptr.as_ptr()))
13        }
14        panic!("cannot drop");
15    }
16    else {
17        abort();
18    }
19 }
```

図 4.9 関数 back_to_mut の実装

新たな参照オブジェクトを用いた図 4.1 は、Rc オブジェクトを用いることで以下の図 4.10 のように表すことができる。つまり本研究の参照オブジェクトは、単に Rc オブジェクトの使い方を強制するオブジェクトである。しかし、本研究の新たな参照オブジェクトを用いることで、意図していないメモリリークが起きているかどうかを実行時エラーによって検出でき、ユーザはメモリリークを意識せずにプログラムを書くことができる。これは、Rc オブジェクトを用いる場合と異なる点であり、新たな参照オブジェクトを利用するメリットである。

```
1 fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
2     vec.retain(|x| **x != s);
3 }
4
5 fn main() {
6     let mut vec: Vec<Rc<String>> = Vec::new();
7     let h = String::from("Hello");
8     let w = String::from("World");
9     let mut rng = rand::thread_rng();
10    let h_immut = Rc::new(h);
11    let w_immut = Rc::new(w);
12    for _ in 0..10 {
13        if rng.gen() {
```

```
14         let h_clone = h_immut.clone();
15         vec.push(h_clone);
16     }
17     else {
18         let w_clone = w_immut.clone();
19         vec.push(w_clone);
20     }
21 }
22 remove_string(&mut vec, String::from("Hello"));
23 let mut h_mut = Rc::into_inner(h_immut).unwrap();
24 h_mut.push_str("hoge");
25 println!("{}", h_mut);
26 drop(h_mut);
27 }
```

図 4.10 新たな参照オブジェクトを用いた並列処理の例

第 5 章

並列処理への応用

静的な ownership で柔軟性に欠ける例として、並列処理の例があり、本研究の新たな参照オブジェクトを応用できる。

5.1 Rust の並列処理

まず、Rust でどのように並列処理を行うことができるかを述べる。

5.1.1 通常のスレッド

Rust の通常のスレッド生成では、owner は高々一つのスレッドに move し、複数のスレッドに move することはできない。以下の図 5.1 は、Rust で通常のスレッドを生成する例であり、3 行目にある move キーワードにより move が起きる。

```
1 fn main() {  
2     let s = String::from("Hello");  
3     thread::spawn(move || {  
4         println!("{}", s);  
5     });  
6 }
```

図 5.1 Rust での通常のスレッド生成の例

Rust では、スレッドのライフタイムは無限とみなされており、スレッドが合流した際に ownership が返却されないため、共有オブジェクトに対する borrowing は不可能である。

5.1.2 scoped thread

共有オブジェクトに対する borrowing ができないことは不便であるため、Rust にはスレッドをまたいだ borrowing を可能にする scoped thread が存在する。これは、スレッドに静的スコープを付与したものであり、静的スコープに従ってスレッドの合流が起きるため、共有オブジェクトに

対する borrowing が可能となる。以下の図 5.2 は、scoped thread を利用した例である。スレッドに付与されたスコープは、10 行目で終了しスレッドの合流が起こる。

```
1 fn main () {
2     let mut s = String::from("Hello");
3     thread::scope(|sc| {
4         sc.spawn(|| {
5             println!("{}", &s);
6         });
7         sc.spawn(|| {
8             println!("{}", &s);
9         });
10    });
11    s.push_str("World");
12 }
```

図 5.2 scoped thread の例

5.2 並列処理の際の問題点

以下図 5.3 は scoped thread で表すことができない。関数 my_spawn は、引数として String オブジェクトを受け取り、スレッドを生成し、スレッドのハンドラを返す関数である。このプログラムは、関数 my_spawn により main 関数外で生成したスレッドが main 関数内で合流するというプログラムである。このプログラムは、スレッドの合流が静的スコープに従っていないため、scoped thread で表すことができない。

```
1 fn my_spawn (a: &String) -> JoinHandle<()>{
2     let t = thread::spawn(|| {
3         println!("{}", &a);
4     });
5     return t
6 }
7 fn main() {
8     let mut a = String::from("Hello");
9     let t1 = my_spawn(&a);
10    let t2 = my_spawn(&a);
11    t1.join();
12    t2.join();
13    a.push_str(" World");
14 }
```

図 5.3 scoped thread で表せない例

Rust には、Readers-Writer Lock [3] がある。これは、複数の読み取りは許すが書き込みのアクセスの際は排他的な場合のみ許すロックである。しかし、Readers-Writer Lock を用いた際には本研

究の参照オブジェクトと異なり、不要なロックが発生するおそれがある。

5.3 新たな参照オブジェクトの応用

スレッドの合流が静的スコープに従わない並列処理は、本研究の新たな参照オブジェクトにより以下の図 5.4 のように表すことができる。関数 `my_spawn` は、引数として `String` オブジェクトへの `immutable` な参照を受け取り、スレッドを生成し、スレッドのハンドラを返す関数である。このプログラムは、関数 `my_spawn` により `main` 関数外で生成したスレッドが `main` 関数内で合流するというプログラムである。8 行目で `mutable` な参照を生成し、9、10 行目で `immutable` な参照に変換した後、複製している。その後、関数 `my_spawn` によりスレッドを生成し、13、14 行目で `main` 関数内でスレッドが合流している。その後関数 `drop` により `ownership` の集約が起き、関数 `back_to_mut` により `mutable` な参照に戻している。

```
1 fn my_spawn (a: &RefImmut<String>) -> JoinHandle<()>{
2     let t = thread::spawn(||{
3         println!("{}", a);
4     });
5     return t
6 }
7 fn main() {
8     let mut s = RefMut::new(String::from("Hello"));
9     let s1 = s.to_immut();
10    let s2 = s1.clone_immut();
11    let t1 = my_spawn(&s1);
12    let t2 = my_spawn(&s2);
13    t1.join();
14    t2.join();
15    drop(s2);
16    let mut s3 = s1.back_to_mut();
17    s3.push_str("World");
18 }
```

図 5.4 新たな参照オブジェクトを用いた並列処理の例

第 6 章

関連研究

本研究で取り入れた、fractional ownership [4] は、有理数計画法を用いて静的に検査されている。また、Rust でなく C 言語の特定のプログラムでメモリ関連のエラーがないことが fractional ownership により検証されている [5]。また、fractional ownership と Rust の静的な ownership を組み合わせた新たな所有権型が提案されており、提案した型システムにより命令型プログラムが検証されている [6]。これらはすべて動的検査でなく静的検査であり、保守的であるものの本研究のような動的検査と比べると柔軟性に欠ける検査となっている。

また、RustBelt [7] により unsafe な機能を用いた際でもメモリ安全であるか証明できる。本研究の新たな参照オブジェクトがメモリ安全であるかを RustBelt を用いて証明できるか検討した。Rc オブジェクトは実装に unsafe な機能が用いられているが、RustBelt によってメモリ安全であることが証明されている。本研究の参照オブジェクトは、Rc オブジェクトの利用方法を強制しているオブジェクトであるため、Rc オブジェクトを用いて本研究の参照オブジェクトを実装できれば、メモリ安全であることが証明できると考えた。しかし、unsafe な機能を用いなければ実装することができなかったため、本研究の参照オブジェクトがメモリ安全であるかは、RustBelt を用いて新たに証明する必要があった。

第 7 章

結論と今後の課題

fractional ownership の考え方を取り入れ、Rust の静的な ownership によるメモリ管理と、動的な参照カウンタの柔軟性を組み合わせた新たな参照オブジェクトを提案した。この参照オブジェクトにより、オブジェクトの解放し忘れを実行時に検出し、メモリリークを防ぐことができる。今後の課題は、新たな参照オブジェクトがメモリ安全であることを証明することが挙げられる。また、新たな参照オブジェクトをより具体的な例で利用し、どれほどメモリリークを防ぐことができたかを計測することも挙げられる。

参考文献

- [1] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael B. Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pp. 103–104. ACM, 2014.
- [2] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, pp. 48–64. ACM, 1998.
- [3] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, Vol. 14, No. 10, pp. 667–668, 1971.
- [4] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, Vol. 2694 of *Lecture Notes in Computer Science*, pp. 55–72. Springer, 2003.
- [5] Kohei Suenaga and Naoki Kobayashi. Fractional ownerships for safe memory deallocation. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pp. 128–143, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [6] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. Borrowable fractional ownership types for verification. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pp. 224–246, Cham, 2024. Springer Nature Switzerland.
- [7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, Vol. 2, No. POPL, pp. 66:1–66:34, 2018.