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 以下の有理数である。新たに生成されるオブジェクトには ownership として 1 が与えられる。参照複製時には ownership が分割される。複製された参照が消滅する際には ownership は集約される。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が、オブジェクトの解放には ownership 1 が必要である。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。

目次

第 1 章

序論

1.1 背景

Rust [?, ?, 5] 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、Redox OS [?] や、Dropbox [?]、Firefox [?] といったシステムプログラミング等に用いられている [?]。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、静的な ownership を無視する抜け道として unsafe な機能が存在する。unsafe な機能が内部の実装で用いられているオブジェクトの一つに `Rc` オブジェクト [?] が存在する。`Rc` オブジェクトは、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コンテナであり、動的な参照カウントにより静的な ownership を補っている。clone 関数により複製が可能であり、その際共通の参照カウントを 1 増やし、参照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 になったときに中身のオブジェクトを解放する。本来 Rust では、ownership の複製をすることはできないが、`Rc` オブジェクトの内部の実装で用いられている unsafe な機能により ownership の複製ができるようになる。なお、Rust の静的な ownership を無視する unsafe な機能が用いられているので、`Rc` オブジェクトは動的な柔軟性が必要な場合にのみ用いられるべきである。

図 1.1 は、静的な ownership のみではメモリ管理が難しい例であり、`Rc` オブジェクトの動的な参照カウントの柔軟性を活用した例である。中身の値が`"Hello"`という文字列である `String` オブジェクトを持つ `Rc` オブジェクト `h` と、中身の値が`"World"`という文字列である `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、その配列から中身の値が`"Hello"`という文字列である `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。このプログラムは、オブジェクトの生存区間が中身の値に依存するため、静的なメモリ

管理は難しい。

```
1  fn main() {
2      let mut vec: Vec<Rc<String>> = Vec::new();
3      let h = Rc::new(String::from("Hello"));
4      let w = Rc::new(String::from("World"));
5      let mut rng = rand::thread_rng();
6      for _ in 0..10 {
7          if rng.gen() {
8              vec.push(h.clone());
9          }
10         else {
11             vec.push(w.clone());
12         }
13     }
14     remove_string(&mut vec, String::from("Hello"));
15     // その他の処理
16 }
```

図 1.1　静的なメモリ管理では柔軟性に欠ける例

しかし、図 1.1 の 17 行目において、その他の処理として、vec は用いるが、h と w は用いない処理が行われる場合、h のメモリ領域は、追加した処理の最中も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、意図せず複製が残ってしまうことにより、メモリリークのおそれがあり、Rust の、メモリリークを防止できるというメリットが損なわれてしまう。このメモリリークは、後ほど第 3 章で述べるように、3 行目から 16 行目を別のスコープにすることで防ぐことができる。また、同様に第 3 章で詳しく述べているが、into_inner 関数を用いることでこのようなメモリリークを検出することもできる。しかし、ユーザのミスにより 3 行目から 16 行目を別のスコープにし忘れた場合や、into_inner 関数を用いることを忘れた場合には、メモリリークが起きているにも関わらず、静的にも動的にも検出することはできない。

また、into_inner 関数の使用忘れやスコープの切り忘れを防止するために、それらをまとめた高階関数を用意することが考えられるが、Rust のように静的な ownership に基づきメモリ管理を行う言語では、高階関数に引数として渡す関数内で用いられているオブジェクトの borrowing の lifetime の推論や指定方法は自明でないという問題がある。

1.2　本研究の概略

本研究では、柔軟な複製を許すものの、ユーザのミスによりメモリリークが起きやすい Rc オブジェクトに代わり、柔軟なエイリアスを許しつつ、メモリリークの検出が可能な仕組みを実現することを目的とする。具体的には、fractional ownership [1] 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。提案する参照オブジェクトの仕組みにより、どのオブジェクトがどこで解放されるはずかが、スコープが終了するとき、もしくは `drop` 関数が明示的に呼び出されるときのいずれかとなるため、解放のタイミングがプログラムの文面上明確になる。また、意図せず複製が残っていることで中身のオブジェクトが解放されない可能性がある `Rc` オブジェクトと異なり、提案する参照オブジェクトは実際に解放されなければ実行時エラーが起きるため、ユーザのミスによるオブジェクトの解放し忘れを検出できる。

Fractional ownership とは、0 以上かつ 1 以下の有理数である。新たに生成されるオブジェクトには ownership として 1 が与えられる。参照複製時には ownership が分割される。複製された参照が消滅する際には ownership は集約される。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が、オブジェクトの解放には ownership 1 が必要である。本来 fractional ownership [1] は静的に検査されるが、我々の参照オブジェクトは、参照カウントの逆数を動的な fractional ownership とみなしている。例えば、参照カウントが 2 であるとき、fractional ownership は 1/2 とみなす。

図 1.2 は、図 1.1 のプログラムを本研究の参照オブジェクトで表したプログラムである。本研究の参照オブジェクトは、本来の Rust の ownership に従う mutable な参照オブジェクトと、Rust の ownership に違反して複製が可能な immutable な参照オブジェクトに区別されている。動的な参照カウントによる柔軟性を活用する際には、6 行目、7 行目の `to_immut` 関数を用いて mutable な参照から immutable な参照に変換し、活用後、参照カウントが 1 であれば 20 行目、23 行目の `back_to_mut` 関数を用いて mutable な参照に戻すことができる。

またこの参照オブジェクトは、図 1.3 のように mutable な参照に戻さないまま消滅した場合、実行時エラーが起きる。これにより、意図せず参照の複製が残っていることによるオブジェクトの解放し忘れを実行時に検出することができる。

このように我々の参照オブジェクトは、mutable な参照の時に中身のオブジェクトのメモリ領域が解放されていることを動的に検査している。この参照オブジェクトは、必要な時に動的な参照カウントの柔軟性を利用することができるため、静的な ownership のみでメモリ管理行うことが難しい場合に用いることも可能である。本研究では、動的な参照カウントの柔軟性が必要な場合として、特に並列処理に我々の参照オブジェクトを用いた場合について述べている。詳しくは、第 5 章で述べる。

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。

```
fn main() {
    let mut vec: Vec<FRefImmut<String>> = Vec::new();
    let mut h = FRefMut::new(String::from("Hello"));
    let mut w = FRefMut::new(String::from"World"));
    let mut rng = rand::thread_rng();
    let h_immut = h.to_immut();
    let w_immut = w.to_immut();
    for _ in 0..10 {
        if rng.gen() {
            let h_immut1 = h_immut.clone_immut();
            vec.push(h_immut1);
        }
        else {
            let w_immut1 = w_immut.clone_immut();
            vec.push(w_immut1);
        }
    }

    remove_string(&mut vec, String::from("Hello"));
    let mut h_mut = h_immut.back_to_mut();
    drop(h_mut);
    drop(vec);
    let mut w_mut = w_immut.back_to_mut();
}
```

図 1.2　本研究の方式を用いた具体例

```
fn main() {
    let mut vec: Vec<FRefImmut<String>> = Vec::new();
    let mut h = FRefMut::new(String::from("Hello"));
    let mut w = FRefMut::new(String::from("World"));
    let mut rng = rand::thread_rng();
    let h_immut = h.to_immut();
    let w_immut = w.to_immut();
    for _ in 0..10 {
        if rng.gen() {
            let h_immut1 = h_immut.clone_immut();
            vec.push(h_immut1);
        }
        else {
            let w_immut1 = w_immut.clone_immut();
            vec.push(w_immut1);
        }
    }

    remove_string(&mut vec, String::from("Hello"));
    // Mutableに見忘れ!
}
```

図 1.3　メモリリークを検出する例

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た後、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"); // sがStringオブジェクトのowner
4     {
5       let t = String::from("World"); // tがStringオブジェクトのowner
6     } // tが指すStringオブジェクトが解放
7   } // sが指すStringオブジェクトが解放
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とコンパイルエラーが発生する。

```
1 fn main() {
2   let mut s = String::from("Hello"); // sがStringオブジェクトのowner
3   let new_s = s; // sからnew_sにownershipがmove
4   s.push_str("World"); // コンパイルエラー
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"); // xはStringオブジェクトのowner
3 } // xの指すStringオブジェクトが解放
4
5 fn main() {
6   let mut s = String::from("Hello"); // sがStringオブジェクトのowner
7   f(s); // sから関数fの引数xにownershipがmove
8   s.push_str("Oops!"); // コンパイルエラー
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、図 2.4 のように 3 行目で関数 f が引数 x を ownership ごと返却するように変更することで、コンパイルエラーを発生させずに実行することができる。しかし、関数の引数に ownership が move するたびに、ownership を返却するようにプログラムを書くことは面倒である。

```
1  fn f(mut x: String) -> String{
2    x.push_str("World"); // xはStringオブジェクトのowner
3    return x // ownershipを返却
4  }
5
6  fn main() {
7    let mut s = String::from("Hello"); // sがStringオブジェクトのowner
8    let mut return_s = f(s); // sから関数fの引数xにownershipがmove
9    return_s.push_str("World");
10 }
```

図 2.4　Ownership を返却する例

2.1.3　Borrowing

Move の際の不便な点を解消するため、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。変数名の前に&とつけることで、Rust に特有の「参照」を作成し、ownership を borrowing できる。

Ownership を borrowing している例を図 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); // xはStringオブジェクトのowner
3  } // sにownershipが自動で返却
4
5  fn main() {
6    let mut s = String::from("Hello"); // sがStringオブジェクトのowner
7    f(&s); // sのownershipが一時的に関数fの引数xに貸し出される
8    s.push_str("World"); // sはStringオブジェクトのowner
9  }
```

図 2.5　Borrowing の例

2.1.4 Mutable な borrowing

上で述べた borrowing は、読み取りのみ可能である immutable な borrowing であり、読み書きが可能である mutable な borrowing も存在する。変数名の前に`&mut` とつけると mutable な borroiwng ができる。Mutable な borrowing をしている例を図 2.6 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: &mut String) {
2   x.push_str("hoge") // xはStringオブジェクトのowner
3 } // sにownershipが自動で返却
4
5 fn main() {
6   let mut s = String::from("Hello"); // sがStringオブジェクトのowner
7   f(&mut s); // sのownershipが一時的に関数fの引数xに貸し出される
8   s.push_str("World"); // sはStringオブジェクトのowner
9 }
```

図 2.6　Mutable な borrowing の例

Immutable な borrowing は、すでに mutable な borrowing が行われていなければ複数回行うことができる。逆に mutable な borrowing は、mutable な borrowing も immutable な borrowing も行われていないときにしか行うことはできない。

2.1.5 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、図 2.7 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。

図 2.8 は、スコープと lifetime が一致していない例である。このとき、7 行目以降では `s_borrow` は用いられていないため、`s_borrow` のライフタイムは 4 行目から 6 行目と推論され、`t_borrow` のライフタイムは 7 行目から 8 行目で `main` 関数のスコープが終了するまでであると推論される。このように、ライフタイムはブロックにより推論されているわけではなく、スコープとライフタイムが一致しない場合もある。この推論は、Rust の Non-lexical lifetimes (NLL) [?] により可能となっている。

```
1 fn f<'a>(x: &'a String) { // xのライフタイムと関数本体が一致
2   println!("{}", &x);
3 } // sにownershipが自動で返却
4
5 fn main() {
6   let mut s = String::from("Hello"); // sがStringオブジェクトのowner
7   f(&s); // sのownershipが一時的に関数fの引数xに貸し出される
8   s.push_str("World"); // sはStringオブジェクトのowner
9 }
```

図 2.7　Lifetime 注釈

```
1 fn main () {
2     let mut s = String::from("Hello");
3
4     let s_borrow = &mut s;
5     println!("{}", s_borrow);
6     // s_borrowのライフタイム終了
7     let t_borrow = &mut s; // Mutableなborrowingが可能
8 } // t_borrowのライフタイム終了
```

図 2.8　スコープと lifetime が一致していない例

2.2　Rust の参照カウント

後述の例のように、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、図 2.9 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。オブジェクトの生存期間が中身の値に依存しているため、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。

```
use std::rc::Rc;
use rand::Rng;
fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<Rc<String>> = Vec::new();
    let h = Rc::new(String::from("Hello"));
    let w = Rc::new(String::from("World"));
    let mut rng = rand::thread_rng();
    for _ in 0..10 {
        if rng.gen() {
            let h_clone = h.clone(); // clone関数により複製を生成
            vec.push(h_clone);
        }
        else {
            let w_clone = w.clone(); // clone関数により複製を生成
            vec.push(w_clone);
        }
    }
    remove_string(&mut vec, String::from("Hello")); // hの複製が消滅
} // hとwの中身のオブジェクトが解放
```

図 2.9　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 [1] という概念が存在する。Fractional ownership とは 0 以上かつ 1 以下の有理数である。オブジェクト生成時には ownership として 1 が与えられ、参照複製時には ownership は分割され、複製された参照が消滅する際には ownership は集約される。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が、オブジェクトの解放には ownership 1 が必要である。Ownership が 0 より大きく 1 未満のオブジェクトは、ownership の集約により ownership が 1 となれば解放できる。図 2.10 は、fractional ownership を利用した疑似コード例である。1 行目で、変数 a、b、c に ownership として 1 を与え、2 行目では、d と a で ownership を 0.5 ずつに分割している。3 行目では*b += *a と*c += *d は並列に処理されており、b、c は ownership 1 を持つため読み書き可能であり、a、d は ownership 0.5 を持つため読み取りのみ可能になる。4 行目で、分割されていた ownership は a に集約され、a は読み書き可能となる。

```
1 a := new; b := new; c := new; // Ownership 1を与える
2 d = a; // dとaでownershipを0.5ずつに分割
3 *b += *a || *c += *d // b、cは読み書き可能、a、dは読み取りのみ可能
4 *a += *b + *c // a、b、cは読み書き可能
```

図 2.10　Fractional ownership の利用例

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。23 行目にその他の処理として、配列 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 24 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 を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。

図 3.2 のプログラムは、24 行目で h と w のスコープが終了し、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。

しかし、例えば図 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、23 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、24 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。

また、Rc オブジェクトの中身のオブジェクトを取り出す into_inner 関数を用いることで、図 3.1 のプログラムは図 3.4 のように、メモリリークが起きないように表すことができる。24 行目の部分で、複製が消滅した後、25 行目で into_inner 関数により中身のオブジェクトを取り出している。複製が消滅していなければ、into_inner を用いる際にエラーが起きるため、意図していないメモリリークを検出することができる。しかし、into_inner 関数により中身のオブジェクトを取り出すことを忘れてしまうと、メモリリークが発生してしまう。

このように、Rc オブジェクトを用いた場合、ユーザによるミスが起こりやすく、結果としてメモリリークが起きる場合がある。現在、Rust でこのようなメモリリークが起きないことは静的に検査する手段が存在しないだけでなく、動的に検査する手段も存在していない。また、Rust はメモリリークを防止できるというメリットがあるが、Rc オブジェクトを用いるとそのメリットは損なわれてしまう。

```
use std::rc::Rc;
use rand::Rng;
fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<Rc<String>> = Vec::new();
    let h = Rc::new(String::from("Hello"));
    let w = Rc::new(String::from("World"));
    let mut rng = rand::thread_rng();
    for _ in 0..10 {
        if rng.gen() {
            let h_clone = h.clone();
            vec.push(h_clone);
        }
        else {
            let w_clone = w.clone();
            vec.push(w_clone);
        }
    }
    remove_string(&mut vec, String::from("Hello"));
    // その他の処理
} // hとwの中身のオブジェクトが解放
```

図 3.1　Rc オブジェクトを利用する際の問題点

```
use std::rc::Rc;
use rand::Rng;
fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<Rc<String>> = Vec::new();
    {
      let h = Rc::new(String::from("Hello"));
      let w = Rc::new(String::from("World"));
      let mut rng = rand::thread_rng();
      for _ in 0..10 {
          if rng.gen() {
              let h_clone = h.clone();
              vec.push(h_clone);
          }
          else {
              let w_clone = w.clone();
              vec.push(w_clone);
          }
      }
      remove_string(&mut vec, String::from("Hello"));
    } // hの中身のオブジェクトが解放
    // その他の処理
}
```

図 3.2　メモリリークを防いだ例

```
use std::rc::Rc;
use rand::Rng;
fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<Rc<String>> = Vec::new();
    {
      let h = Rc::new(String::from("Hello"));
      let w = Rc::new(String::from("World"));
      let mut rng = rand::thread_rng();
      for _ in 0..10 {
          if rng.gen() {
              let h_clone = h.clone();
              vec.push(h_clone);
          }
          else {
              let w_clone = w.clone();
              vec.push(w_clone);
          }
      }
      remove_string(&mut vec, String::from("Hell")); // スペルミス
    } // hもwもどちらも中身のオブジェクトは解放されない
    // その他の処理
}
```

図 3.3　ユーザのミスによるメモリリーク

```
use std::rc::Rc;
use rand::Rng;
fn remove_string(vec: &mut Vec<Rc<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
  let mut vec: Vec<Rc<String>> = Vec::new();
  {
    let h = String::from("Hello");
    let w = String::from("World");
    let mut rng = rand::thread_rng();
    let h_immut = Rc::new(h);
    let w_immut = Rc::new(w);
    for _ in 0..10 {
      if rng.gen() {
        let h_clone = h_immut.clone();
        vec.push(h_clone);
      }
      else {
        let w_clone = w_immut.clone();
        vec.push(w_clone);
      }
    }
    remove_string(&mut vec, String::from("Hello")); // 参照が消滅
    let mut h_mut = Rc::into_inner(h_immut).unwrap(); // into_inner
  }
  // その他の処理
}
```

図 3.4　into_inner 関数を用いた例

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、ユーザのミスによるメモリリークが検出できないという問題を解決するために、図 4.1 のように用いることができる参照オブジェクトを提案する。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 mutable 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 な参照オブジェクトを、10、11 行目で immutable な参照オブジェクトに変換し、14、19 行目で immutable な参照オブジェクトを複製している。remove_string が終了した後、26 行目、32 行目で mutable な参照オブジェクトに戻している。

この参照オブジェクトは、読み書き可能であり、かつ解放が可能な FRefMut オブジェクトと読み取りのみ可能である FRefImmut オブジェクトの 2 つがあり、静的に区別されている。FRefMut オブジェクトは Rust の通常の ownership に従い、同時に一つしか存在しない。FRefMut オブジェクトは、7、8 行目のように new 関数により生成できる。10、11 行目のように to_immut 関数を用いると、FRefMut オブジェクトを FRefImmut オブジェクトに変換でき、その際 Rust の静的な ownership の move により FRefMut オブジェクトは消滅する。14、19 行目のように、FRefImmut オブジェクトは、Rust の通常の ownership に違反して clone_immut 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。26、32 行目のように、参照カウントが 1 つであるときのみ、back_to_immut 関数を用いて FRefImmut オブジェクトから FRefMut オブジェクトへ変換でき、その際 move により FRefImmut オブジェクトは消滅する。FRefImmut オブジェクトから FRefMut オブジェクトに戻し忘れると、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

```
use fRef_implementation::fractional_ref::FRefImmut;
use fRef_implementation::fractional_ref::FRefMut;
use rand::Rng;
fn remove_string(vec: &mut Vec<FRefImmut<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
  let mut vec: Vec<FRefImmut<String>> = Vec::new();
  let mut h = FRefMut::new(String::from("Hello")); // Mutableな参照
  let mut w = FRefMut::new(String::from("World")); // Mutableな参照
  let mut rng = rand::thread_rng();
  let h_immut = h.to_immut(); // Mutableからimmutableへ変換
  let w_immut = w.to_immut(); // Mutableからimmutableへ変換
  for _ in 0..10 {
    if rng.gen() {
      let h_immut1 = h_immut.clone_immut(); // 参照の複製
      println!("{}", h_immut1);
      vec.push(h_immut1);
    }
    else {
      let w_immut1 = w_immut.clone_immut(); // 参照の複製
      println!("{}", w_immut1);
      vec.push(w_immut1);
    }
  }
  remove_string(&mut vec, String::from("Hello"));
  let mut h_mut = h_immut.back_to_mut(); // Immutableからmutableへ
  h_mut.push_str("hoge");
  println!("{}", h_mut);
  drop(h_mut);
  drop(vec);
  let mut w_mut = w_immut.back_to_mut(); // Immutableからmutableへ
}
```

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

きる。

また、図 4.4 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 FRefImmut オブジェクトを FRefMut オブジェクト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図 4.5 のような実行時エラーが発生するため、FRefMut オブジェクト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。

図 4.2 は、図 4.1 とほぼ同じプログラムであるが、FRefImmut オブジェクトを FRefMut オブ

ジェクトに戻していないプログラムである。この場合、FRefMut オブジェクトに戻さないまま参照が消滅してしまっているため図 4.3 のように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。

```
use fRef_implementation::fractional_ref::FRefImmut;
use fRef_implementation::fractional_ref::FRefMut;
use rand::Rng;
fn remove_string(vec: &mut Vec<FRefImmut<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
  let mut vec: Vec<FRefImmut<String>> = Vec::new();
  let h = FRefMut::new(String::from("Hello")); // FRefMutの生成
  let w = FRefMut::new(String::from("World")); // FRefMutの生成
  let mut rng = rand::thread_rng();
  let h_immut = h.to_immut(); // FRefMutからFRefImmutへ変換
  let w_immut = w.to_immut(); // FRefMutからFRefImmutへ変換
  for _ in 0..10 {
    if rng.gen() {
      let h_immut1 = h_immut.clone_immut(); // FRefImmutの複製
      println!("{}", h_immut1);
      vec.push(h_immut1);
    }
    else {
      let w_immut1 = w_immut.clone_immut(); // FRefImmutの複製
      println!("{}", w_immut1);
      vec.push(w_immut1);
    }
  }
  remove_string(&mut vec, String::from("Hello"));
  // FRefMutに戻していない
} // error!
```

図 4.2　オブジェクトの解放し忘れを検出する例

図 4.6 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、FRefMut に戻すことができずに図 4.7 のよう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る。

```
$ cargo run
   Compiling dynamic_error_example2 v0.1.0
    (/home/baba/experiment-report/202401/0105/dynamic_error_example2)
    Finished dev [unoptimized + debuginfo] target(s) in 0.94s
     Running `target/debug/dynamic_error_example2`
Hello
World
Hello
World
Hello
Hello
Hello
World
World
World
thread 'main' panicked at /home/baba/experiment-report/202401/0105
    /dynamic_error_example2/src/lib.rs:154:17:
memory leak
note: run with `RUST_BACKTRACE=1`
    environment variable to display a backtrace
```

図 4.3　オブジェクトの解放し忘れのエラー

```
use fRef_implementation::fractional_ref::FRefImmut;
use fRef_implementation::fractional_ref::FRefMut;
use rand::Rng;
fn remove_string(vec: &mut Vec<FRefImmut<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<FRefImmut<String>> = Vec::new();
    let h = FRefMut::new(String::from("Hello"));
    let w = FRefMut::new(String::from("World"));
    let mut rng = rand::thread_rng();
    let h_immut = h.to_immut();
    let w_immut = w.to_immut();
    for _ in 0..10 {
        if rng.gen() {
            let h_immut1 = h_immut.clone_immut();
            println!("{}", h_immut1);
            vec.push(h_immut1);
        }
        else {
            let w_immut1 = w_immut.clone_immut();
            println!("{}", w_immut1);
            vec.push(w_immut1);
        }
    }

    remove_string(&mut vec, String::from("Hello"));
    let mut w_mut = w_immut.back_to_mut(); // error!
    w_mut.push_str("hhhhh");
    println!("{}", w_mut);
}
```

図 4.4　Mutable な参照へ戻せないことを検出する例

```
$ cargo run
   Compiling dynamic_error_example v0.1.0
    (/home/baba/experiment-report/202401/0105/dynamic_error_example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.98s
     Running `target/debug/dynamic_error_example`
Hello
Hello
Hello
World
Hello
Hello
World
World
World
Hello
thread 'main' panicked at /home/baba/experiment-report/202401/0105
    /dynamic_error_example/src/lib.rs:72:17:
cannot back to mut
note: run with `RUST_BACKTRACE=1`
    environment variable to display a backtrace
```

図 4.5　FRefMut に戻し忘れた際のエラー

```
use fRef_implementation::fractional_ref::FRefImmut;
use fRef_implementation::fractional_ref::FRefMut;
use rand::Rng;
fn remove_string(vec: &mut Vec<FRefImmut<String>>, s: String) {
    vec.retain(|x| **x != s);
}

fn main() {
    let mut vec: Vec<FRefImmut<String>> = Vec::new();
    let h = FRefMut::new(String::from("Hello"));
    let w = FRefMut::new(String::from("World"));
    let mut rng = rand::thread_rng();
    let h_immut = h.to_immut();
    let w_immut = w.to_immut();
    for _ in 0..10 {
        if rng.gen() {
            let h_immut1 = h_immut.clone_immut();
            println!("{}", h_immut1);
            vec.push(h_immut1);
        }
        else {
            let w_immut1 = w_immut.clone_immut();
            println!("{}", w_immut1);
            vec.push(w_immut1);
        }
    }

    remove_string(&mut vec, String::from("Hell")); // スペルミス
    let mut h_mut = h_immut.back_to_mut(); // error!
    h_mut.push_str("hoge");
    println!("{}", h_mut);
    drop(h_mut);

    // その他の処理

    drop(vec);
    let _w_mut = w_immut.back_to_mut();
}
```

図 4.6　ユーザのミスを検出する例

```
$ cargo run
   Compiling random v0.1.0
    (/home/baba/experiment-report/202401/0105/random)
warning: unused variable: `w_new`
  --> src/main.rs:39:9
   |
39 |     let w_new = w_immut.back_to_mut();
   |         ^^^^^ help: if this is intentional,
    prefix it with an underscore: `_w_new`
   |
   = note: `#[warn(unused_variables)]` on by default

warning: `random` (bin "random") generated 1 warning
    (run `cargo fix --bin "random"` to apply 1 suggestion)
    Finished dev [unoptimized + debuginfo] target(s) in 0.67s
     Running `target/debug/random`
Hello
Hello
Hello
World
World
World
Hello
Hello
Hello
Hello
thread 'main' panicked at /home/baba/experiment-report/202401/0105
    /random/src/lib.rs:72:17:
cannot back to mut
note: run with `RUST_BACKTRACE=1`
    environment variable to display a backtrace
```

図 4.7　ユーザのミスによるエラー

4.2 実装

FRefMut オブジェクト、FRefImmut オブジェクトは図 4.8 のように実装した。FRefMut 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。FRefImmut 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 オブジェクトと、PhantomData 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。また、PhantomData により、FRefImmut が T 型の値を保持することをコンパイラに示している。FRefImmut 内のポインタは FRefInner 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 FRefMut<T: ?Sized> {
2      data: T,
3  }
4  pub struct FRefImmut<T: ?Sized> {
5      ptr: NonNull<FRefInner<T>>,
6      phantom: PhantomData<FRefInner<T>>
7  }
8  struct FRefInner<T: ?Sized> {
9      ref_count: atomic::AtomicUsize,
10     data: T
11 }
```

図 4.8　新たな参照オブジェクトの実装

オブジェクトを引数として受け取り、そのオブジェクトを中身として持つ FRefMut オブジェクトを生成する new 関数と、FRefMut オブジェクトを引数として受け取り、FRefImmut オブジェクトに変換して返す to_immut 関数は図 4.9 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 FRefMut が消滅するように実装した。

FRefImmut 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_immut 関数は図 4.10 ように実装した。clone_immut 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 FRefImmut の内部の参照カウントが 1 増加するように実装した。

FRefImmut オブジェクトを引数として受け取り、FRefMut オブジェクトに変換する back_to_mut 関数は図 4.11 のように実装した。back_to_mut 関数の引数を borowwing でなく move するようにし、変換した後引数の FRefImmut が消滅するように実装した。また、2 行目で FRefImmut 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、FRefImmut のままでは消滅できないように実装した。

FRefImmut オブジェクトを引数として受け取り、その参照を消滅させる drop 関数は図 4.12 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、FRefImmut を FRefMut 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。

```
pub fn new(data: T) -> FRefMut<T> {
    Self { data: data }
}
pub fn to_immut(self: FRefMut<T>) -> FRefImmut<T>{ // Moveにより消滅
    let mut this = ManuallyDrop::new(self);
    let inner =
        unsafe{ptr::read(&mut this.data)};
    let x = Box::new(FRefInner {
        ref_count: AtomicUsize::new(1),
        data: inner,
    });
    FRefImmut {ptr: Box::leak(x).into(), phantom:PhantomData}
}
```

図 4.9　関数 new と関数 to_immut の実装

```
pub fn clone_immut(&self: &FRefImmut<T>) -> FRefImmut<T>
{
    let old_size = self.inner().ref_count.fetch_add(1, Release);
    if old_size > MAX_REFCOUNT {
        abort();
    }
    FRefImmut { ptr: self.ptr, phantom: PhantomData }
}
```

図 4.10　関数 clone_immut の実装

なお、実装からわかるように提案する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ことから、Rc オブジェクトと Rc オブジェクトの中身を取り出す関数 into_inner を用いて実装することが可能のように思える。その際、図 4.13 のように immutable な参照オブジェクトのフィールドが Rc オブジェクトになる。Immutable な参照から mutable な参照に戻す際には、フィールドに対して into_inner を呼び出す必要があり、into_inner の引数は、borrowing でなく move が起きることから、オブジェクトのフィールドは move する。そのため、本来オブジェクトが解放された後にそのフィールドが解放されるはずだが、オブジェクトの解放より前にフィールドが解放されることになる。提案する参照オブジェクトでは、immutable な参照を消滅させる drop 関数内で、オブジェクトの解放し忘れを検出するために mutable な参照に戻し忘れていないか確認する必要があり、drop 関数によってオブジェクトの解放の際にフィールドを用いた処理が行われる可能性がある。この場合、すでに解放されているフィールドを用いる処理

```
pub fn back_to_mut(self: FRefImmut<T>) -> FRefMut<T> { // Move
    if self.inner().ref_count.load(Acquire) != 1{
        panic!("cannot back to mut");
    }
    let mut this = ManuallyDrop::new(self);
    let inner =
        unsafe { ptr::read(&mut (*this.ptr.as_ptr()).data)};
    unsafe {
        dealloc(this.ptr.cast().as_mut(),
                Layout::for_value_raw(this.ptr.as_ptr()))
    }
    FRefMut { data: inner }
}
```

図 4.11　関数 back_to_mut の実装

```
impl<T: ?Sized> Drop for FRefImmut<T> {
    fn drop(&mut self) {
        let count = self.inner().ref_count.fetch_sub(1, Relaxed);
        atomic::fence(Acquire);
        if count > 1{
            return
        }
        else if count == 1{
            unsafe { ptr::drop_in_place(self.ptr.as_ptr()) };
            if !panicking() {
                panic!("memory leak");
            }
        }
        else {
            abort();
        }
    }
}
```

図 4.12　関数 drop の実装

が行われる可能性があるため、Rust では静的なエラーが発生する。なお、オブジェクトとフィールドの解放を同時に行うように書くことはできない。従って、Rc オブジェクトを用いて提案する参照オブジェクトを実装する場合、Rc オブジェクト以外に unsafe な機能を用いる必要がある。

```
1 pub struct FRefMut<T: ?Sized> {
2     data: T,
3 }
4 pub struct FRefImmut<T: ?Sized> {
5     rc: Rc<T>
6 }
```

図 4.13　Rc オブジェクトと関数 into_inner による実装の試み

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。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 use std::thread;
2 fn main() {
3     let s = String::from("Hello");
4     thread::spawn(move || { // sがスレッドにmove
5         println!("{}", s);
6     });
7 }
```

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。図 5.2 は、Rust で通常のスレッドで共有オブジェクトに対して borrowing をしようとしている例である。4、7 行目で borrowing を行っているが、9、10 行目で ownership が返却されず、コンパイルエラーとなる。

```
1  use std::thread;
2  fn main() {
3      let s = String::from("Hello");
4      let t1 = thread::spawn(|| {
5          println!("{}", &s); // コンパイルエラー
6      });
7      let t2 = thread::spawn(|| {
8          println!("{}", &s); // コンパイルエラー
9      });
10     t1.join(); // Ownershipが奪取されない
11     t2.join(); // Ownershipが奪取されない
12 }
```

図 5.2　通常のスレッドで borrowing ができない例

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。図 5.3 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  use std::thread;
2  fn main () {
3      let mut s = String::from("Hello");
4      thread::scope(|sc| { // scがスコープ
5          sc.spawn(|| {
6              println!("{}", &s);
7          });
8          sc.spawn(|| {
9              println!("{}", &s);
10         });
11     }); // スコープが終了し、スレッドが合流
12     s.push_str("World");
13 }
```

図 5.3　Scoped thread の例

5.2 並列処理の際の問題点

図 5.4 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  use std::thread;
2  use std::thread::JoinHandle;
3  fn my_spawn (a: &String) -> JoinHandle<()>{
4      let t = thread::spawn(|| {
5          println!("{}", &a);
6      });
7      return t
8  }
9  fn main() {
10     let mut a = String::from("Hello");
11     let t1 = my_spawn(&a);
12     let t2 = my_spawn(&a);
13     t1.join();
14     t2.join();
15     a.push_str("World");
16 }
```

図 5.4　scoped thread で表せない例

5.3 我々の提案の応用

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図 5.5 のように表すことができる。関数 my_spawn は、引数として String オブジェクトを保持する FRefImmut 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。16 行目で FRefMut を生成し、17、18 行目で FRefImmut 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、21、22 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 FRefMut に戻している。

```
1  #![feature(thread_spawn_unchecked)]
2  use fRef_implementation::fractional_ref::FRefImmut;
3  use fRef_implementation::fractional_ref::FRefMut;
4  use std::thread;
5  use std::thread::JoinHandle;
6  fn my_spawn (a: &FRefImmut<String>) -> JoinHandle<()>{
7    let builder = thread::Builder::new();
8    unsafe {
9      let t = builder.spawn_unchecked(||{
10       println!("{}", a);
11     }).unwrap();
12     return t
13   }
14 }
15 fn main() {
16   let mut s = FRefMut::new(String::from("Hello")); // FRefMutの生成
17   let s1 = s.to_immut(); // FRefMutからFRefImmutへ変換
18   let s2 = s1.clone_immut(); // FRefImmutの複製
19   let t1 = my_spawn(&s1);
20   let t2 = my_spawn(&s2);
21   t1.join();
22   t2.join();
23   drop(s2);
24   let mut s3 = s1.back_to_mut(); // FRefImmutからFRefMutへ変換
25   s3.push_str("World");
26 }
```

図 5.5　新たな参照オブジェクトを用いた並列処理の例

5.3.1　Readers-Writer Lock

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。図 5.6 は、図 5.5 を Readers-Writer Lock を用いて表したプログラムである。しかし、Readers-Writer Lock を用いた際には、16 行目のように本来不要なロックが発生するおそれがある。本研究の提案を応用すると、ロックではなく `FRefMut` と `FRefImmut` の区別により競合状態を防止できる。

```
1  #![feature(thread_spawn_unchecked)]
2  use std::thread;
3  use std::thread::JoinHandle;
4  use std::sync::RwLock;
5  fn my_spawn (a: &String) -> JoinHandle<()>{
6      let builder = thread::Builder::new();
7      unsafe {
8          let t = builder.spawn_unchecked(||{
9              println!("{}", a);
10         }).unwrap();
11         return t
12     }
13 }
14 fn main() {
15     let mut s = RwLock::new(String::from("Hello"));
16     let s1 = s.read().unwrap();
17     let s2 = s.read().unwrap();
18     let t1 = my_spawn(&s1);
19     let t2 = my_spawn(&s2);
20     t1.join();
21     t2.join();
22     drop(s1);
23     drop(s2);
24     let mut s3 = s.write().unwrap(); // 不要なロック
25     s3.push_str("World");
26 }
```

図 5.6　Readers-Writer Lock 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 [1] 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 [7]。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 [4] 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実験することも挙げられる。

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

参考文献

[1] 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.

[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] 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.

[5] 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.

[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] 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.