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

2023 年 11 月 30 日 17:55–18:35
オンライン

要旨

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。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。ownership は、変数への代入や関数に引数を渡す際に移動する。

しかし、静的な ownership のみで行うメモリ管理は柔軟性に欠ける。そのため Rust には、Rc (Reference Counted) オブジェクトが存在する。Rc オブジェクトはコンテナであり、中に別のオブジェクトが存在し、そのオブジェクトの参照の数を動的にカウントする。Rc オブジェクトは、clone 関数により、同じオブジェクトを指す参照を複製し参照のカウントを 1 増やす。Rc オブジェクトは、参照が消滅する際にカウントを 1 減らし、カウントが 0 になった時に中身のオブジェクトのメモリ領域が解放される。しかし Rc オブジェクトを用いた場合、メモリ領域が自動で解放されるため、ユーザが意図していないタイミングでメモリ領域が解放されることがある。その場合、メモリ領域が解放されているべきタイミングで解放されていないことがあり、メモリリークが起こることがある。

以上のような問題に対処するため、本研究では fractional ownership [Boyland 2003] を導入する。Fractional ownership とは、0 以上かつ 1 以下の有理数であり、1 はオブジェクトが読み書き可能、0 より大きく 1 未満の値は読み取りのみ可能であることを表す。オブジェクト生成時には ownership として 1 を与え、参照生成時には ownership を分割し、参照が消滅する際には ownership を集約する。fractional ownership は本来静的に検査される。

本研究では Rust の静的な ownership と、fractional ownership の動的検査を組み合わせることで、より柔軟なメモリ管理の手法を提案・実装する。この手法は、メモリ領域の解放が起きるタイミングがユーザの意図に反しないため Rust の Rc オブジェクトとは異なり、メモリリークを検出することができる。また、fractional ownership の検査を、本来のように静的でなく動的に行うことで、ベクターなどを用いた動的に検査ができない場合も柔軟に検査を行うことができる。具体的には、メモリ領域の解放のタイミングを指定し、解放が可能かを動的に検査するオブジェクトの実装を行った。

目次

第 1 章

序論

1.1 背景

Rust についての簡単な説明 Rust の Arc の問題点

1.2 目的

Arc より強いメモリ管理の実現によりメモリリークを防ぎたい

1.3 本論文の構成

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いる。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開放される。コード例は以下に示す。3 行目で、変数 s は String オブジェクトの owner となる。4 行目で、owner である s のスコープが終了するため 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"); // がsowner
4   } // が指すオブジェクトが解放sString
5 }
```

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コード例のように move が起きる。3 行目で s から new_s に ownership が move しているため、4 行目では s を利用することができない。

```
1 fn main() {
2   let s = String::from("Hello"); // がsowner
3   let new_s = s; // がに移動ownershipnew_s
4   s.push_str("World"); // はでないためコンパイルエラーsowner
5 } // が指すオブジェクトが解放new_sString
```

また、関数に引数を渡す際には以下のコード例のように move が起きる。7 行目で s から関数 f の引数に ownership が move している。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 でなくなっておりコンパイルエラーとなる。

```
1  fn f(mut x: String) {
2    x.push_str("World");
3  } // の指すオブジェクトが解放xString
4
5  fn main() {
6    let mut s = String::from("Hello"); // がsowner
7    f(s); // がの引数に移動ownershipf
8    s.push_str("Oops!"); // はでないためコンパイルエラーsowner
9    // オブジェクトは解放されており危険String
10 }
```

2.1.3 borrowing

Ownership は Rust に特有の「参照」を作成することで借りることができる。これを borrowing と呼ぶ。

2.1.4 lifetime

2.2 Arc について

2.3 Rust の並列処理

2.3.1 通常のスレッド

(move, Arc, Mutex)

2.3.2 scoped thread

2.4 問題点

scoped thread では不自由な例 Arc でメモリリークが起きる場合

第 3 章

分数権限を動的に検査する参照オブジェクトの提案

3.1 インターフェース

新たな参照オブジェクトでは、

3.2 実装

3.3 例

第 4 章

考察

第 5 章

結論と今後の課題