Joint Rc and RefCell Analysis in Rust

ALEXANDER CARMICHAEL, University of Colorado Boulder, USA

I present a combined static analyzer for Rust. Rc and RefCell are frequently used together in order to implement recursive data structures in Rust. My analyzer combines analyzing both Rc and RefCell in a single pass, which finds errors with the usage of both primitives. Since common runtime errors stemming from improper usage of RefCell are similar to that of Rc, this process has the potential to find errors with the usage of both, with less code, and a faster runtime than checking them individualy.

ACM Reference Format:
Alexander Carmichael. 2025. Joint Rc and RefCell Analysis in Rust. 1, 1 (December 2025), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Rust is an incredibly popular statically typed programming language. Rust includes a built in borrow checker to guard against common memory errors, such as use-after-free. The borrow checker requires that every variable have a single "owner". Other data structures can "borrow" or "mutably borrow" a variable in order to read and write to it. A variable cannot be edited by its owner or mutably borrowed if at least one regular borrow is active, in order to prevent a data race. This rule makes it impossible to implement recursive data structures in Rust in the same manner as they would be done in a language such as C++. For that reason, Rust includes several primitives to get around the borrow checker's restrictions. They still have the same requirements as the borrow checker, but shift the verification from compile-time to runtime. In a data structure like a graph, multiple nodes can point to a child node. Rust's borrow checker makes that difficult. One node must be the "owner", and every other node that points to the child node must prove that it's lifetime is less than or equal to that of the owner. Rust provides the Rc primitive to address that requirement. Rc contains an immutable inner value, and can have many owners. Rc maintains an internal count of owners. When owners go out of scope, Rc decrements its internal counter. When the number of owners reaches zero, Rc deallocates its internal value and itself. This allows many structures to point to a single node without having to prove anything to the borrow checker. Rc has a restriction in that the internal data is immutable. Another primitive, RefCell, is commonly used to alleviate that. RefCell contains something mutable, and provides the .borrow() and .borrow_mut() methods to access it. These methods are dynamically checked, but obey the same rules as the borrow checker. A mutable borrow when a borrow is already active will crash the program.

1.1 Motivating Example

```
use std::rc::Rc;
use std::cell::RefCell;

```

Author's Contact Information: Alexander Carmichael, Alexander.Carmichael@colorado.edu, University of Colorado Boulder, Boulder, Colorado, USA.

ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
struct Node {
  id: i32,
  next: Option<Rc<RefCell<Node>>>,
}

impl Drop for Node {
  fn drop(&mut self) {
    println!("Node {} dropped.", self.id);
  }
}

fn main() {
  let a = Rc::new(RefCell::new(Node { id: 1, next: None }));

  let b = Rc::new(RefCell::new(Node { id: 2, next: None }));

  a.borrow_mut().next = Some(b.clone());
  b.borrow_mut().next = Some(a.clone());

  drop(a);
  drop(b);
}
```

This program allocates two "nodes". It then makes the next node of each, the other.it then drops both nodes. What is interesting is that the "Node _ dropped." message will never print. That's because the nodes are never actually deallocated. The fact that each node points to the other means that each node's Rc will have a strong reference count of one. An Rc only deallocates its inside when its strong reference count reaches zero. That means these nodes will remain as long as this program is running, a memory leak.

```
01  fn main() {
02    let a = Rc::new(RefCell::new(Node { id: 1, next: None }));
03
04    let b = a.borrow();
05    let c = a.borrow_mut();
06  }
```

This second motivating example shows a potential error when using RefCell. The RefCell must abide by the normal borrowing rules. A mutable borrow is not allowed when there is at least one other borrow, as that could lead to a memory error such as use after free. Since RefCell is checked at runtime, this code passes Rust's borrow checker and compiles. When run, line 5 crashes the program as the requirement of no other borrows is not met.

1.2 Contribution

A dual-purpose static analyzer for Rust.

- An analyzer for Rc errors. It keeps an internal graph of all active variables and the variables that they point to. Upon the deallocation of a variable, it checks to see if the variable is still somehow accessible from the code. If it is not, it then checks the strong reference count to the variable. If it's greater than 0, then there is a memory leak, and the analyzer detects it.

- An analyzer for RefCell errors. Building on top of the first analyzer, it also keeps a record of the number of borrows made for each struct in the graph. If the combination of borrows made violates Rust's borrow checking rules, the analyzer flags it as a location where the program will crash.

The idea is to check both of these similar but different properties together. The benefit is that the shared work between the analyses can be factored out, and therefore the overall analysis is easier to write.

2 Semantics

$$\begin{array}{rcll}
\text{struct} & s & ::= & \{f_1 : s_1, f_2 : s_2, f_3 : s_3, \ldots, f_n : s_n\} \\
\text{borrow} & b & ::= & b\{s\} \\
\text{mutable borrow} & m & ::= & m\{s\} \\
\text{variable} & v & ::= & s \mid b \mid m \\
\text{all vars} & a & ::= & \{v, v_2, v_3, \ldots, v_n\}
\end{array}$$

$$\boxed{a \vdash e : a} \qquad \textsc{FieldAssign}\ \frac{a \vdash x \Downarrow s \qquad a[v] \Downarrow s}{a \vdash \text{s.f = x} \Downarrow a\{v \longrightarrow s\{f \longrightarrow x\}\}} \qquad \textsc{DropStruct}\ \frac{a[v] \Downarrow s}{a \vdash \text{drop(v)} \Downarrow a\text{.drop(v)}}$$

$$\textsc{VarAssignStruct}\ \frac{a \vdash x \Downarrow s \mid b \mid m}{a \vdash \text{let v = x} \Downarrow a\{v \longrightarrow x\}} \qquad \textsc{VarAssignBorrow}\ \frac{\forall y \in a, a[y] \neq m\{x\}}{a \vdash \text{let v = b\{x\}} \Downarrow a\{v \longrightarrow b\{x\}\}}$$

$$\textsc{VarAssignBorrowMutable}\ \frac{\forall y \in a, a[y] \neq b\{x\}, a[y] \neq m\{x\}}{a \vdash \text{let v = b\{x\}} \Downarrow a\{v \longrightarrow b\{x\}\}}$$

3 Conclusion

The analyzer works. When given the first motivating example, it outputs "dropping b leaked memory as b still has remaining references". When given the second, "borrowing a failed because of an invalid borrow configuration". The actual Rust implementation is somewhat brittle, in that it doesn't comprehend enough of the Rust AST to traverse programs with irrelevant statements. It does not support Rust's more complicated methods/patterns for variable initialization, and it will therefore get confused/fail on more complicated syntax. A lot is left to be implemented. Recursive data structures are frequently initialized through loops, so support of loops would be a significant improvement. As for the actual goals of the project, this analyzer was somewhat of a success and somewhat of a failure. It's a success in that it does successfully find both types of issues, and there was a lot of similarity between the issues that was successfully factored out. The problem is that RefCell errors are rather simplistic by nature. It's difficult to come up with examples where misuse of RefCell results in a program crash that aren't just the programmer borrowing, forgetting they already have the value, and borrowing again to look at it without freeing or reusing the first. It's hard to make a more complicated error as that essentially requires "fighting" Rust's built in borrow checker, which already does much to make dangling references and other memory errors impossible. The result is that the Rc side of the analyzer is somewhat complex, and there are many ways to improve it and make it more useful. The RefCell side of the analyzer is simplistic and already does everything it can. Any further attempt to find issues with RefCell would be uninteresting, as those occur extremely rarely and require willful ignorance of Rust design patterns and the borrow checker's advice. So yes, while this analyzer does succeed in sharing some logic between the RefCell and Rc analyses, it's rather limited, and there isn't any room for further sharing of logic. I still think the idea has merit. RefCell in part has simple errors because it itself is a simple primitive. Perhaps Rust's more complex primitives, meant for multi-threaded use, have the answer. Maybe a combination analyzer for a multi-threaded ownership primitive, and a multi-threaded borrowing primitive has the needed complexity to make a multi-purpose analyzer that works and has enough overlap that something is gained by doing the analysis together.