Exploring Stronger Aliasing Optimizations for Rust
Bachelor Thesis Project Description

Dominik Schwaiger

supervised by Johannes Hostert, Ralf Jung
ETH Zürich

April 2026

1 Introduction

The Rust Programming Language currently has two competing aliasing models: Stacked Borrows [1] and Tree Borrows [2]. Stacked Borrows is stricter, defining more code as Undefined Behavior. Tree Borrows is much more permissive, currently disallowing some optimizations (as the one in Listing 1). This Thesis tries to make Tree Borrows more restrictive, allowing for more optimizations.

2 Background

Consider the Rust function `func1`:

```
fn func1(x: &mut u32, n: u32) {
  let mut i = 0;
  while i < n {
    i = i + 1;
    *x = *x + 1;
  }
}
```

```
fn func2(x: &mut u32, n: u32) {
  *x = *x + n;
}
```

Listing 1: Desired Optimization

This function takes a mutable reference `x`. The Rust borrow checker checks that such mutable references have no aliases, that is, this mutable reference is the only reference to the memory it points at.

The Rust compiler would really like to exploit these strong type system guarantees to optimize the program. For example, it might want to replace the body of `func1` with the body of `func2`. This turns out to be surprisingly hard: it becomes necessary to impose a so-called "aliasing model," which precisely defines what the compiler is and is not allowed to assume about such mutable references.

While Stacked Borrows allows this optimizations, Tree Borrows doesn't: Under Tree Borrows, a mutable reference is not always considered writable, but only once it has actually been written through. But in the program above, if `n == 0`, then `*x` is never written to. So the optimization would introduce an extra write where there previously was none, which is not permissible in general under the Tree Borrows model.

3 Goals

As already mentioned, the goal of this thesis is to enable more optimizations under Tree Borrows by making it more restrictive.

This can be accomplished by always considering mutable references writeable. While this first implementation, the "strong mode", allows for the optimization, it also introduces some extra Undefined Behavior where there previously was none. Hence the next step will be to analyze those cases and selectively use the old behavior, the "weak mode".

Finally, the "strong mode" can be used to perform the optimization by modifying the Rust compiler backend, adding the `writable` flag [3] to the generated LLVM bitcode.

3.1 Core Goals

The core goals are based around the implementation in Miri:

- Understand UB, Tree Borrows, and the LLVM `noalias` and `writable` flag.
- Implement the "strong mode" in Tree Borrows in Miri.
- Implement a new attribute in rustc to disable the "strong mode" on a per-function basis.

3.2 Extension Goals

If the time allows, extensions are mostly based around evaluation of the implementation and updating MiniRust:

- Perform an evaluation of how much new UB is introduced, by testing a large amount of Rust code for compliance, and seeing how easy non-compliance is to fix.
- Extend the new attribute to also affect whether rustc emits a `writable` flag in LLVM.
- Benchmark the performance gained by emitting the `writable` flag.
- Implement the "strong mode" and the attribute in MiniRust.

4 References

[1] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020, doi: 10.1145/3371109.

[2] N. Villani, J. Hostert, D. Dreyer, and R. Jung, "Tree Borrows," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1019–1042, June 2025, doi: 10.1145/3735592.

[3] "LLVM Language Reference Manual — LLVM 23.0.0git documentation." Accessed: Feb. 26, 2026. [Online]. Available: https://llvm.org/docs/LangRef.html