

# Possible paper to study during MFoCS seminar

Robbert Krebbers

Radboud University Nijmegen, The Netherlands

5 September 2025

A paper about semantics of Rust

# Tree Borrows

Neven Villani, Johannes Hostert, Derek Dreyer, Ralf Jung

In PLDI 2025: ACM SIGPLAN International Conference on Programming Language Design and Implementation


Recipient of PLDI 2025 Distinguished Paper Award

# Tree Borrows

Neven Villani, Johannes Hostert, Derek Dreyer, Ralf Jung

In PLDI 2025: ACM SIGPLAN International Conference on Programming Language Design and Implementation

Recipient of PLDI 2025 Distinguished Paper Award



Advisor to the official Rust language team

# Tree Borrows

Neven Villani, Johannes Hostert, Derek Dreyer, Ralf Jung

In PLDI 2025: ACM SIGPLAN International Conference on Programming Language Design and Implementation

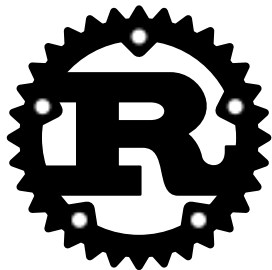
Recipient of PLDI 2025 Distinguished Paper Award

Advisor to the official Rust language team

Given to ca. 10% of accepted papers at PLDI

You should pick this paper if you like

- ▶ Rust
- ▶ Compilers and program transformations
- ▶ Low-level semantics
- ▶ Program verification
- ▶ Formalization in Rocq



# What is the paper about?

**A semantics for Rust using which one can prove compiler optimizations:**

- ▶ Whose correctness relies on Rust's strong type system
- ▶ In the presence of `unsafe` (raw pointers, as used internally in many libraries)



# Rust 101

Pointer types in Rust:

- ▶ `&mut` A – **unique and mutable** pointer to value of type A
- ▶ `&` A – **shared and immutable** pointer to value of type A

# Rust 101

Pointer types in Rust:

- ▶ `&mut A` – **unique and mutable** pointer to value of type A
- ▶ `& A` – **shared and immutable** pointer to value of type A

**Insight:** If you have `x : &mut A` and `y : &mut A` (as function arguments), then `x` and `y` are unequal (they do not alias)



In which way would a compiler optimize the following program?

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

In which way would a compiler optimize the following program?

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    *x  
}
```

It can just return 13:

```
fn write_both(x: &mut i32, y: &mut i32) -> i32 {  
    *x = 13;  
    *y = 20;  
    13 // avoid load from 'x'  
}
```

**Why is that correct?**  $x$  and  $y$  are both unique pointers, so they cannot alias, i.e.,  $x \neq y$ , hence writes via  $y$  do not affect the value of  $x$

**Note:** This optimization would not be correct in C/C++

## The challenge

**Developing a semantics for Rust that allows one to prove the correctness of such optimizations is really tricky**

A tree structure turns out to be a good fit, hence the title “Tree Borrows”

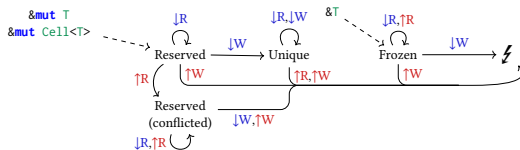
# The challenge

Developing a semantics for Rust that allows one to prove the correctness of such optimizations is really tricky

A tree structure turns out to be a good fit, hence the title “Tree Borrows”

## Challenges:

- ▶ The semantics matches up with the behaviors expected in real Rust code
- ▶ Optimizations can be proved correct without getting down in details
  - ▶ Use of a relational separation logic
- ▶ Results can be trusted
  - ▶ Implementation of the semantics applied to many real Rust libraries
  - ▶ Theory formalized in Rocq



## Possible topics for the second paper

- ▶ Semantics of other languages (C or LLVM)
- ▶ Verification of the Rust type system (RustBelt)
- ▶ Compiler verification methods (CompCert or Simuliris)
- ▶ Comparison to prior Rust semantics (Stacked Borrows)

