



TECHNISCHE
UNIVERSITÄT
WIEN

Brushing off the Rust: Towards Compositional Memory Safety Verification for unsafe Rust

Rust Verification Workshop 2026

Florian Sextl
2026-04-13

Motivation 1

```
/// ...
/// # Safety
/// TODO
unsafe fn neighbors<T: Copy>(s: *mut [T], i: usize) -> (T,T)
{
    unsafe {
        let (before, after) = s.split_at_mut(i);
        (ptr::read((before as *const T).offset(
            i as isize - 1)),
         ptr::read((after as *const T).offset(1)))
    }
}
```

Motivation 2

```
#[requires(TODO)]
#[ensures(TODO)]
unsafe fn neighbors<T: Copy>(s: *mut [T], i: usize) -> (T,T)
{
    unsafe {
        let (before, after) = s.split_at_mut(i);
        (ptr::read((before as *const T).offset(
            i as isize - 1)),
         ptr::read((after as *const T).offset(1)))
    }
}
```

Solution: Biabduction

Solution: Biabduction

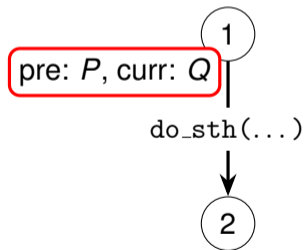
Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)

Solution: Biabduction

Framework

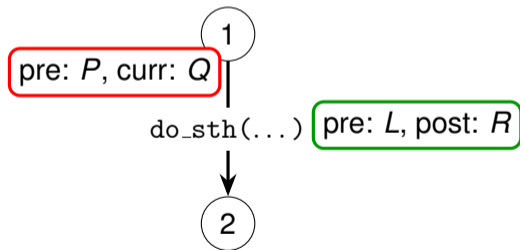
Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)



Solution: Biabduction

Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)



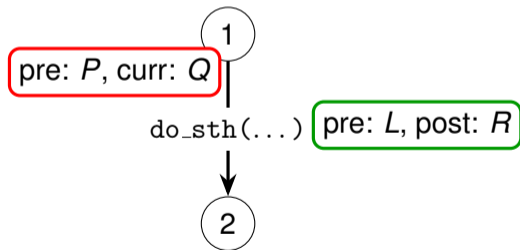
Solution: Biabduction

Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)

Biabduction

$Q * M \triangleright L * F$



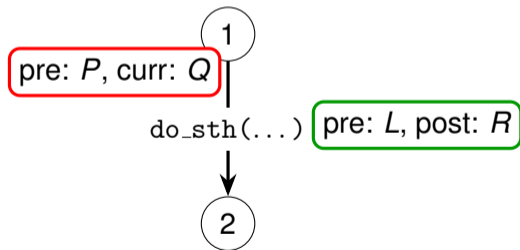
Solution: Biabduction

Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)

Biabduction

$Q * \boxed{M} \triangleright L * \boxed{F}$ s.t. $Q * M \vdash L * F$



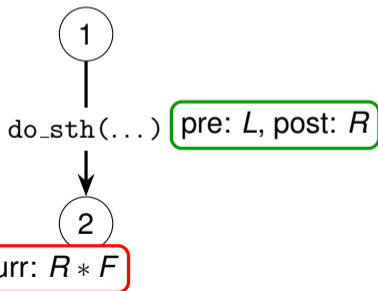
Solution: Biabduction

Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q)

Biabduction

$Q * M \triangleright L * F$ s.t. $Q * M \vdash L * F$



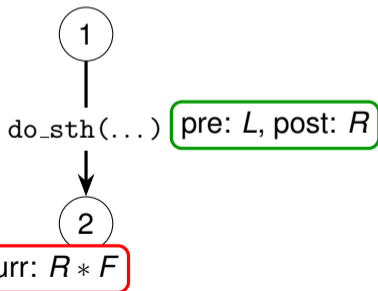
Solution: Biabduction

Framework

Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q) , resulting contracts (pre-/post-condition)

Biabduction

$Q * M \triangleright L * F$ s.t. $Q * M \vdash L * F$



Solution: Biabduction

Framework

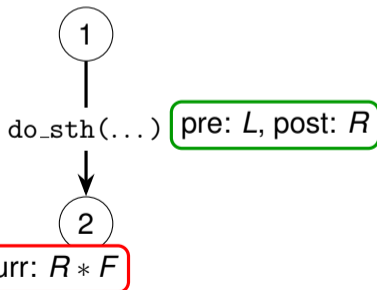
Static analysis via symbolic execution, symbolic state with pre-condition and current state (P, Q), resulting contracts (pre-/post-condition)

Biabduction

$Q * \boxed{M} \triangleright L * \boxed{F}$ s.t. $Q * M \vdash L * F$

Biabduction-based Shape Analysis

Most widely known tool: Infer



Biabduction for Rust

```
unsafe fn neighbors<T: Copy>(s: *mut [T], i: usize) -> (T,T)
{ ... }
```

Safety of neighbors

Biabduction for Rust

```
unsafe fn neighbors<T: Copy>(s: *mut [T], i: usize) -> (T,T)  
{ ... }
```

Safety of neighbors

- s must span a single allocation
- s must be correctly aligned

Biabduction for Rust

```
unsafe fn neighbors<T: Copy>(s: *mut [T], i: usize) -> (T,T)  
{ ... }
```

Safety of `neighbors`

- `s` must span a single allocation
- `s` must be correctly aligned
- both `i - 1` and `i + 1` must be in range for `s`
- the elements at index `i - 1` and `i + 1` must be valid for reading

New WIP Prototype

- Implements biabduction-based shape analysis for (unsafe) Rust

New WIP Prototype

- Implements biabduction-based shape analysis for (unsafe) Rust
- Open source (working name: *brush_rs*)



New WIP Prototype

- Implements biabduction-based shape analysis for (unsafe) Rust
- Open source (working name: *brush_rs*)
- Currently supported: slices, arrays, raw pointer arithmetic, casts/transmutes, enums, etc.



New WIP Prototype

- Implements biabduction-based shape analysis for (unsafe) Rust
- Open source (working name: *brush_rs*)
- Currently supported: slices, arrays, raw pointer arithmetic, casts/transmutes, enums, etc.
- Uses Charon and Z3



New WIP Prototype

- Implements biabduction-based shape analysis for (unsafe) Rust
- Open source (working name: *brush_rs*)
- Currently supported: slices, arrays, raw pointer arithmetic, casts/transmutes, enums, etc.
- Uses Charon and Z3
- Now a demo...



Scope and Goals

Intended Scope

Scope and Goals

Intended Scope

- Focus on memory-related UB

Scope and Goals

Intended Scope

- Focus on memory-related UB
- Other UB mostly ignored

Scope and Goals

Intended Scope

- Focus on memory-related UB
- Other UB mostly ignored
- Primary use case: small to medium sized unsafe code with complex safety requirements

Scope and Goals

Intended Scope

- Focus on memory-related UB
- Other UB mostly ignored
- Primary use case: small to medium sized unsafe code with complex safety requirements
- Single-thread semantics

Scope and Goals

Intended Scope

- Focus on memory-related UB
- Other UB mostly ignored
- Primary use case: small to medium sized unsafe code with complex safety requirements
- Single-thread semantics
- Layout-agnostic (unless `std`)

Scope and Goals

Intended Scope

- Focus on memory-related UB
- Other UB mostly ignored
- Primary use case: small to medium sized unsafe code with complex safety requirements
- Single-thread semantics
- Layout-agnostic (unless `std`)
- Automatically verify safety of caller of unsafe functions

Scope and Goals

```
fn caller() {  
    let mut s = [1u32; 5];  
    let p = &raw mut s as *mut [u32];  
    let (x,y) = unsafe {neighbors(p, 3)};  
}
```

Scope and Goals

Goals and Future

Scope and Goals

Goals and Future

- compute contracts for interesting unsafe functions (e.g. `std`, `zerocopy`, linux kernel code)

Scope and Goals

Goals and Future

- compute contracts for interesting unsafe functions (e.g. `std`, `zerocopy`, linux kernel code)
- support developers and proof engineers, please check it out

Scope and Goals

Goals and Future

- compute contracts for interesting unsafe functions (e.g. `std`, `zerocopy`, linux kernel code)
- support developers and proof engineers, please check it out
- be part of a vast range of formal method tools


Scope and Goals

Goals and Future

- compute contracts for interesting unsafe functions (e.g. `std`, `zerocopy`, linux kernel code)
- support developers and proof engineers, please check it out
- be part of a vast range of formal method tools
- enable shape analysis across **Rust-C-FFI** with our C analyzer Broom

Acknowledgements



The project leading to this application has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440. 

The work executed under the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" is funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022. 