Stacked Borrows: An Aliasing Model for Rust

Ralf Jung\textsuperscript{1,2}, Hoang-Hai Dang\textsuperscript{1}, Jeehoon Kang\textsuperscript{3}, Derek Dreyer\textsuperscript{1}

PRiML 2020 in Beijing Saarbrücken The Internet

\textsuperscript{1}MPI-SWS, Germany
\textsuperscript{2}Mozilla, USA
\textsuperscript{3}KAIST, Korea
Rust – Mozilla’s replacement for C/C++

Rust is the only language to provide...

- Low-level control à la C/C++
- Strong safety guarantees
- Modern, functional paradigms
- Industrial development and backing
Rust is the only language to provide…

- Low-level control à la C/C++
- Strong safety guarantees
- Modern, functional paradigms
- Industrial development and backing

"mainstream"
Rust – Mozilla’s replacement for C/C++

Rust is the only language to provide...

• Low-level control à la C/C++
• Strong safety guarantees
• Modern, functional paradigms
• Industrial development and backing

Core ingredient: sophisticated type system
Rust – Mozilla’s replacement for C/C++

- Low-level control à la C/C++
- Strong safety guarantees
- Modern, functional paradigms
- Industrial development and backing

Core ingredient: sophisticated type system

Goal: exploit the unique type information available in Rust for optimizations
Rust’s reference type comes in two flavors:

1. **Mutable reference**: `&mut T` (no aliasing)
2. **Shared reference**: `&T` (no mutation by anyone)

Mutation + Aliasing
Rust’s reference type comes in two flavors:

1. **Mutable reference**: `&mut T` (no aliasing)

2. **Shared reference**: `&T` (no mutation by anyone)

Rust’s reference types provide strong **aliasing information**.

The optimizer should exploit that!
fn test_noalias(x: &mut i32, y: &mut i32) -> i32 {
    // x, y cannot alias: they are unique pointers
    *x = 42;
    *y = 37;
    return *x; // must return 42
}
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    // unknown_function cannot have an alias to x
    unknown_function();
    return *x; // must return 42
}
Aliasing guarantees: \&mut T Examples

```rust
def fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    // unknown_function cannot have an alias to x
    unknown_function();
    return *x; // must return 42
}
```

escaped pointer

unknown code
fn test_noalias_shared(x: &i32, y: &mut i32) -> i32 {
    let val = *x;
    // cannot mutate x: x points to immutable data
    *y = 37;
    return *x == val; // must return true
}
fn test_shared(x: &i32) -> bool {
    let val = *x;
    // unknown_function_shared cannot mutate x
    unknown_function_shared(x);
    return *x == val; // must return true
}
fn test_shared(x: &i32) -> bool {
    let val = *x;
    // unknown_function_shared cannot mutate x
    unknown_function_shared(x);
    return *x == val; // must return true
}

escaped pointer

unknown code with access to x
These optimizations are the wildest dreams of C compiler developers!
These optimizations are the **wildest dreams** of C compiler developers!

But there is a problem:
These optimizations are the **wildest dreams** of C compiler developers!

But there is a problem:

**UNSAFE CODE!**
Unsafe code can access hazardous operations that are banned in safe code.

```rust
unsafe fn hazardous(x: usize) -> i32 {
    // *mut T is the type of raw (unsafe) pointers
    let x_ptr = x as *mut i32;
    return *x_ptr; // dereferencing an arbitrary integer
}
```
Unsafe code can access hazardous operations that are banned in safe code.

```rust
unsafe fn hazardous(x: usize) -> i32 {
    // *mut T is the type of raw (unsafe) pointers
    let x_ptr = x as *mut i32;
    return *x_ptr; // dereferencing an arbitrary integer
}
```

- Used for better performance, FFI, implementing many standard library types
- Generally encapsulated by safe APIs
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // must return 42
}
fn main() {
    let mut l = 13;

    let answer = test_unique(&mut l);
    println!("The answer is {}", answer);
}

fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // must return 42
}
1: static mut ALIAS: *mut i32 = ptr::null_mut();
2: fn main() {
3:   let mut l = 13;
4:   unsafe { ALIAS = &mut l as *mut i32; }
5:   let answer = test_unique(&mut l);
6:   println!("The answer is {}", answer);
7: }

11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function();
14:   return *x; // should return 42
15: }

ALIAS is a raw pointer (*mut T)
```rust
1: static mut ALIAS: *mut i32 = ptr::null_mut();
2: fn main() {
3:   let mut l = 13;
4:   unsafe { ALIAS = &mut l as *mut i32; }
5:   let answer = test_unique(&mut l);
6:   println!("The answer is {}", answer);
7: }

11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function();
14:   return *x; // should return 42
15: }
```
static mut ALIAS: *mut i32 = ptr::null_mut();

fn main() {
    let mut l = 13;
    unsafe { ALIAS = &mut l as *mut i32; }
    let answer = test_unique(&mut l);
    println!("The answer is {}", answer); // prints 7
}

fn unknown_function() {
    unsafe { *ALIAS = 7; }
}

fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // should return 42, but returns 7
}
```rust
1: static mut ALIAS: *mut i32 = ptr::null_mut();
2: fn main() {
3: let mut l = 13;
4: unsafe { ALIAS = &mut l as *mut i32; }
5: let answer = test_unique(&mut l);
6: println!("The answer is {}", answer); // prints 7
7: }
8: fn unknown_function() {
9: unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12: *x = 42;
13: unknown_function();
14: return *x; // should return 42, but returns 7
15: }
```
static mut ALIAS: *mut i32 = ptr::null_mut();

fn main() {
    let mut l = 13;
    unsafe { ALIAS = &mut l as *mut i32; }
    let answer = test_unique(&mut l);
    println!("The answer is {}", answer); // prints 7
}

fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // should return 42, but returns 7
}

Goal: rule out misbehaving programs
Use of unsafe code imposes proof obligations on the programmer:
No use of dangling/NULL pointers, no data races, …
Review: Undefined Behavior

Use of unsafe code imposes **proof obligations** on the programmer:

No use of dangling/NULL pointers, no data races, …

Violation of proof obligation leads to **Undefined Behavior**.
Review: Undefined Behavior

Use of unsafe code imposes proof obligations on the programmer: No use of dangling/NULL pointers, no data races, . . .

Violation of proof obligation leads to Undefined Behavior.

Compilers can rely on these proof obligations when justifying optimizations.
Review: Undefined Behavior

Use of unsafe code imposes proof obligations on the programmer: no use of dangling/NULL pointers, no data races, ... Violation of proof obligation leads to Undefined Behavior.

Compilers can rely on these proof obligations when justifying optimizations.
static mut ALIAS: *mut i32 = ptr::null_mut();

fn main() {
    let mut l = 13;
    unsafe { ALIAS = &mut l as *mut i32; }
    let answer = test_unique(&mut l);
    println!("The answer is {}", answer); // prints 7
}

fn unknown_function() {
    unsafe { *ALIAS = 7; }
}

fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // should return 42, but returns 7
}
Stacked Borrows

Aliasing model defining which pointers may be used to access memory, ensuring
• uniqueness of mutable references, and
• immutability of shared references.

Stacked Borrows is restrictive enough to enable useful optimizations
formal proof
Stacked Borrows is permissive enough to enable programming
checked standard library test suite by instrumenting the Rust interpreter Miri/one.osf/zero.osf
Stacked Borrows

**Aliasing model** defining which pointers may be used to access memory, ensuring

- **uniqueness** of mutable references, and
- **immutability** of shared references.
Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - formal proof

Stacked Borrows is permissive enough to enable programming checked standard library test suite by instrumenting the Rust interpreter Miri
Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - formal proof
- Stacked Borrows is permissive enough to enable programming
  - checked standard library test suite by instrumenting the Rust interpreter Miri
**Stacked Borrows: Key Idea**

Model proof obligations after existing static “borrow” check

<table>
<thead>
<tr>
<th>Borrow Checker</th>
<th>Stacked Borrows</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>only safe code</td>
<td>safe &amp; unsafe code</td>
</tr>
</tbody>
</table>
let mut l = 13;
let a = &mut l; // a *borrows* from l
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
5: *a = 4;
1: let mut l = 13;
2: let a = &mut l;  // a *borrows* from l
3: let b = &mut *a;  // b *reborrows* from a
4: *b = 3;
5: *a = 4;
6: *b = 4;  // ERROR: lifetime of ‘b‘ has ended
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended

1. The lender a does not get used until the lifetime of the loan has expired.
```rust
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended
```

1. The lender `a` does not get used until the lifetime of the loan has expired.
2. The recipient of the borrow `b` may only be used while its `lifetime` is ongoing.
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of `b` has ended

- **Chain** of borrows:
  - l borrowed to a reborrowed to b
- **Well-bracketed**: no ABAB
1: let mut l = 13;
2: let a = &mut l; // a *borrows* from l
3: let b = &mut *a; // b *reborrows* from a
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of `b` has ended

(Re)borrows are organized in a stack.

- Chain of borrows:
  1 borrowed to a reborrowed to b

- Well-bracketed: no ABAB
Stacked Borrows ingredients

Pointer values carry a tag \( \text{tag} \quad (\text{PtrVal} \triangleq \text{Loc} \times \mathbb{N}) \)

Example: \((0x40, 1)\)

references \((&\text{mut } T)\) are identified by a tag
Stacked Borrows ingredients

Pointer values carry a tag \((\text{PtrVal} \triangleq \text{Loc} \times \mathbb{N})\)
Example: \((0x40, 1)\)

Every location in memory comes with an associated stack \((\text{Mem} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Byte} \times \text{Stack})\)

: 0x40: 0xFE, [0: Unique, 1: Unique]
Reference tagged 1 borrows from reference tagged 0

Every location in memory comes with an associated stack \( (\text{Mem} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Byte} \times \text{Stack}) \)

: 0x40: 0xFFE, \([0: \text{Unique}, 1: \text{Unique}]\)

:
For every use of a reference or raw pointer:

- Extra **proof obligation**:  
  \[ \Rightarrow \text{the tag must be in the stack} \]
- Extra operational effect:  
  \[ \Rightarrow \text{pop elements further up off the stack} \]
1: let mut l = 13;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended
1: let mut l = 13; // Tag: 0

Stack:
In safe code, such Stacked Borrow violations are prevented by the borrow checker.
1: let mut l = 13; // Tag: 0

Stack:
[0: Unique]
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1

Stack:
[0: Unique, 1: Unique]

Find old tag 0 on stack;
pop items above (none);
add new tag 1: Unique above it
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1
3: let b = &mut *a; // Tag: 2

Stack:
[0: Unique, 1: Unique, 2: Unique]

Find old tag 1 on stack;
pop items above (none);
add new tag 2: Unique above it
```rust
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1
3: let b = &mut *a; // Tag: 2
4: *b = 3;
```

Stack:

[0: Unique, 1: Unique, 2: Unique]

Find tag 2 on stack;
pop items above (none)
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1
3: let b = &mut *a; // Tag: 2
4: *b = 3;
5: *a = 4;

Stack:
[0: Unique, 1: Unique, 2: Unique]

Find tag 1 on stack;
pop items above (2: Unique)
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1
3: let b = &mut *a; // Tag: 2
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b’ has ended

Stack:
[0: Unique, 1: Unique]

Find tag 2 on stack – there is no such item! 🌟
```rust
1: let mut l = 13; // Tag: 0
2: let a = &mut l; // Tag: 1
3: let b = &mut *a; // Tag: 2
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of `b` has ended
```

In **safe** code, such Stacked Borrows violations 🌟 are prevented by the borrow checker.
static mut ALIAS: *mut i32 = ptr::null_mut();

fn main() {
    let mut l = 13;
    unsafe { ALIAS = &mut l as *mut i32; }
    let answer = test_unique(&mut l);
    println!("The answer is {}", answer); // prints 7
}

fn unknown_function() {
    unsafe { *ALIAS = 7; }
}

fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // should return 42, but returns 7
}
1: let mut l = 13;
2: let ALIAS = &mut l as *mut i32;
3: let x = &mut l; // was argument to test_unique
4: *x = 42;
5: unsafe { *ALIAS = 7; } // was unknown_function
6: println!("The answer is {}", *x);
1: let mut l = 13; // Tag: 0
1: let mut l = 13; // Tag: 0

Stack:
[0: Unique]

It is undefined behavior to use a pointer whose tag is not on the stack.
1: let mut l = 13; // Tag: 0
2: let ALIAS = &mut l as *mut i32; // Tag: ⊥

Stack:
[0: Unique, ⊥: SharedRW]

Find old tag 0 on stack;
pop items above (none);
add new tag ⊥: SharedRW above it
1: let mut l = 13;  // Tag: 0
2: let ALIAS = &mut l as *mut i32;  // Tag: ⊥
3: let x = &mut l;  // Tag: 1

Stack:
[0: Unique, ⊥: SharedRW, 1: Unique]

Find old tag 0 on stack;
pop items above (⊥: SharedRW);
push new tag 1: Unique
1: let mut l = 13; // Tag: 0
2: let ALIAS = &mut l as *mut i32; // Tag: ⊥
3: let x = &mut l; // Tag: 1
4: *x = 42;

Stack:
[0: Unique, 1: Unique]

Find tag 1 on stack;
pop items above (none)
1: let mut l = 13; // Tag: 0
2: let ALIAS = &mut l as *mut i32; // Tag: ⊥
3: let x = &mut l; // Tag: 1
4: *x = 42;
5: unsafe { *ALIAS = 7; }

Stack:
[0: Unique, 1: Unique]

Find tag ⊥ on stack – there is no such item! ✯
1: let mut l = 13; // Tag: 0
2: let ALIAS = &mut l as *mut i32; // Tag: ⊥
3: let x = &mut l; // Tag: 1
4: *x = 42;
5: unsafe { *ALIAS = 7; }

It is **undefined behavior** to use a pointer whose tag is not on the stack.

Find tag ⊥ on stack – there is no such item!
Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - formal proof 🐧
- Stacked Borrows is permissive enough to enable programming
  - checked standard library test suite by instrumenting the Rust interpreter Miri
Stacked Borrows

• Stacked Borrows is restrictive enough to enable useful optimizations
  ✔ formal proof 🐧
fn test_unique(x: &mut i32) -> i32 {
    \*x = 42;
    unknown_function();
    return \*x; // must return 42
}
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // must return 42
}
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // must return 42
}

if unknown_function accesses this memory, it will pop x’s tag off the stack
Incomplete proof sketch

```rust
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    unknown_function();
    return *x; // must return 42
}
```

UB unless x’s permission is still in the stack

if unknown_function accesses this memory, it will pop x’s tag off the stack

x’s tag is at the top of the stack
Stacked Borrows

- Stacked Borrows is restrictive enough to enable useful optimizations
  - ✔️ formal proof
Stacked Borrows

- Stacked Borrows is permissive enough to enable programming
  ✔ checked standard library test suite by instrumenting the Rust interpreter Miri
1: static mut ALIAS: *mut i32 = ptr::null_mut();
2: fn main() {
3:    let mut l = 13;
4:    unsafe { ALIAS = &mut l as *mut i32; }
5:    let answer = test_unique(&mut l);
6:    println!("The answer is {}", answer);  // prints 7
7: }
8: fn unknown_function() {
9:    unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:    *x = 42;
13:    unknown_function();
14:    return *x;  // should return 42, but returns 7
15: }
1: static mut ALIAS: *mut i32 = ptr::null_mut();
2: fn main() {
3:   let mut l = 13;
4:   unsafe { ALIAS = &mut l as *mut i32; }
5:   let answer = test_unique(&mut l);
6:   println!("The answer is {}", answer); // prints 7
7: }
8: fn unknown_function() {
9:   unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:     *x = 42;
13:     unknown_function();
14:     return *x;
15: }

error: Miri evaluation error: no item granting write access to tag <untagged> found in borrow stack.
    example.rs:9:12
    | unsafe { *ALIAS = 7; }
    | ^^^^^^^^^^ no item granting write access to tag <untagged> found in borrow stack.
    note: inside call to `unknown_function` at example.rs:13:3

The Rust **standard library** and an increasing number of **user crates** regularly have their test suites checked by **Miri**.

So far, this uncovered 11 aliasing violations. 🎇
What else?

What I didn’t talk about:

• Shared references & interior mutability
• Protectors (enable *writes* to be moved across unknown code)

Future work:

• Concurrency
• Integrating Stacked Borrows into RustBelt
A **dynamic model** of Rust’s reference checker ensures soundness of type-based optimizations, even in the presence of **unsafe** code.
Try Miri out yourself!

- **Web version:** [https://play.rust-lang.org/](https://play.rust-lang.org/) ("Tools")
- **Installation:** `rustup component add miri`
- **Miri website:** [https://github.com/rust-lang/miri/](https://github.com/rust-lang/miri/)

Also check out our project website: [https://plv.mpi-sws.org/rustbelt](https://plv.mpi-sws.org/rustbelt)