The Rust Borrow Checker

Lukáš Hozda

2025-04-10

Braiins Systems s.r.o

Ownership and Lifetimes

Ownership: Rust's Secret Weapon

- Core feature of Rust's memory safety guarantees
- Allows memory efficiency without garbage collection
- Every value has exactly one owner
- Values are dropped when owner goes out of scope
- Ownership can be transferred (moved)

References: Borrowing Owned Values

- Shared reference: `&T` - multiple allowed, read-only
- Mutable reference: `&mut T` - only one allowed, can modify
- References are non-owning pointers with restrictions

Reference Rules

Two fundamental rules:

1. A reference cannot outlive its referent
2. A mutable reference cannot be aliased

These simple rules prevent memory safety issues like:

- Use-after-free
- Double-free
- Data races

Why These Rules Matter

```rust
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // This won't compile! We're returning a reference
    // to a value that will be dropped at function end
    &s
}
```

And Also This...

```rust
let mut data = vec![1, 2, 3];
// get a reference to first element
let x = &data[0];

// This won't compile! We can't modify while x is borrowed
data.push(4); // push might reallocate the vector's memory

println!("{}", x); // would be a dangling pointer
```

The Borrow Checker

What Is the Borrow Checker?

- Core component of the Rust compiler
- Analyzes how references are created and used
- Enforces the reference rules at compile time
- Tracks scopes, lifetimes, and borrows through control flow
- Prevents memory safety violations without runtime cost

Borrow Checker Analysis

- Tracks each variable's "state": owned, borrowed, mutably borrowed
- Follows all code paths and ensures rules are never violated
- Analyzes when references are created and last used
- Can understand non-overlapping borrows of different struct fields
- Much more advanced than simple scope-based checking

How Borrowck Works (Simplified)

1. Constructs a control-flow graph of the program
2. Tracks the state of each variable at every point
3. Ensures borrowing rules are never violated
4. Reports errors when unsafe conditions would occur
5. Optimizes away unnecessary restrictions when safe

Lifetimes

What Are Lifetimes?

- Regions of code where a reference is valid
- Usually implicit within function bodies
- Must be expressed at function boundaries
- Annotated with `'a`, `'b`, `'static`, etc.
- Not garbage collection - purely compile-time

Lifetime Annotations

```rust
// Lifetimes in function signatures
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Lifetimes in structs
struct Excerpt<'a> {
    part: &'a str,
}
```

Lifetime Desugaring

Function signatures use lifetimes to show relationships:

```rust
// What we write
fn first_word(s: &str) -> &str { /* ... */ }

// What Rust sees (after elision)
fn first_word<'a>(s: &'a str) -> &'a str { /* ... */ }
```

Example: References that Outlive Referents

```
,compile_fail
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s // Error: returns reference to data owned by local variable
}
```

Desugared:

```
,ignore
fn as_str<'a>(data: &'a u32) -> &'a str {
    let s = format!("{}", data);
    &s // Error: 's' lives for a smaller scope than 'a
}
```

Example: Aliasing a Mutable Reference

```
,compile_fail
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4); // Error: cannot borrow `data` as mutable
println!("{}", x);
```

The borrow checker prevents this because `push` might reallocate the vector's memory, invalidating the reference x.

Lifetime Elision

Rust has rules to infer lifetimes in common patterns:

1. Each input reference gets its own lifetime parameter
2. If there is exactly one input lifetime, it's assigned to all outputs
3. If there's a `&self`/`&mut self` input, its lifetime is assigned to outputs
4. Otherwise, output lifetimes must be specified

Lifetime Elision Examples

```rust
// Elided:
fn print(s: &str);
// Expanded:
fn print<'a>(s: &'a str);

// Elided:
fn substr(s: &str, until: usize) -> &str;
// Expanded:
fn substr<'a>(s: &'a str, until: usize) -> &'a str;

// Error - can't determine output lifetime:
fn get_str() -> &str;
```

Unbounded Lifetimes

In unsafe code, references can be created “out of thin air”:

```
,no_run
fn get_str<'a>(ptr: *const String) -> &'a str {
    unsafe { &*ptr } // Creates a reference with unbounded lifetime
}
```

- Dangerous! Creates references with arbitrary lifetimes
- Should be bounded as quickly as possible
- Common in `transmute`, raw pointers, FFI

Subtyping and Variance

What is Subtyping?

- Concept that one type can be used in place of another
- If `Sub <: Super`, then `Sub` satisfies all requirements of `Super`
- Allows more flexible type relationships

Lifetime Subtyping

For lifetimes: `'long <: 'short` if `'long` completely contains `'short`

```rust
let hello: &'static str = "hello";
{
    let world = String::from("world");
    let world = &world; // shorter lifetime than 'static

    // This works! 'static can be used where 'world is expected
    debug(hello, world);
}
```

Variance: How Subtyping Propagates

Given `Sub <: Super`:

- Covariant: `F<Sub> <: F<Super>` (subtyping preserved)
- Contravariant: `F<Super> <: F<Sub>` (subtyping inverted)
- Invariant: No relationship exists between `F<Sub>` and `F<Super>`

Variance of Common Types

	'a	T	U
`&'a T `	covariant	covariant	
`&'a mut T`	covariant	invariant	
`Box<T>`		covariant	
`Vec<T>`		covariant	
`Cell<T>`		invariant	
`fn(T) -> U`		**contra**variant	covariant
`*const T`		covariant	
`*mut T`		invariant	

Why is this Important?

```rust
fn assign<T>(input: &mut T, val: T) {
    *input = val;
}

fn main() {
    let mut hello: &'static str = "hello";
    {
        let world = String::from("world");
        assign(&mut hello, &world); // Error!
    }
    println!("{hello}"); // Would use freed memory
}
```

Understanding the Error

- We're assigning `&world` to `hello` (via `input`)
- `&world` has a shorter lifetime than `&'static str`
- If allowed, it would create a dangling reference
- `&mut T` is invariant over `T` to prevent this exact problem

Why `&mut T` Must Be Invariant

The invariance of `&mut T` is crucial for memory safety:

```rust
let mut v: Vec<&'static str> = Vec::new();
{
    let s = String::from("hello");
    let rs = &s;
    // If &mut Vec<&'static str> was a subtype of &mut Vec<&'short str>
    // this would allow us to put a short-lived reference into a collection
    // that promises all its contents live for 'static
    v.push(rs); // Error!
}
println!("{:?}", v); // Would use freed memory!
```

Function Pointers and Contravariance

Function arguments are contravariant:

```rust
fn store(input: &'static str) {
    // Stores in a collection requiring 'static values
}

fn demo<'a>(input: &'a str, f: fn(&'a str)) {
    f(input);
}

fn main() {
    let local = String::from("local");
    // Error! Can't pass store (requires 'static) where
    // a function accepting any lifetime is expected
    demo(&local, store);
}
```

Advanced Lifetime Patterns

Higher-Rank Trait Bounds (HRTBs)

Used when a function or closure needs to work with **any** lifetime:

```rust
// Accept any function that works with any lifetime
fn call_with_ref<F>(f: F)
where
    F: for<'a> Fn(&'a i32) -> &'a i32,
{
    let x = 10;
    let result = f(&x);
    println!("{}", result);
}
```

`'static` Lifetimes

Two meanings:

1. Lives for the entire program duration
2. Has no lifetime dependencies (for trait bounds)

```rust
// Lives for the program duration
let s: &'static str = "hello";

// No lifetime dependencies
fn process<T: 'static>(t: T) { /* ... */ }
```

Splitting Borrows

Borrowck understands field-level borrowing:

```
struct Point { x: i32, y: i32 }

let mut p = Point { x: 0, y: 0 };
let x = &mut p.x;
let y = &mut p.y; // OK! Different fields

*x += 10;
*y += 20;
```

Splitting Complex Structures

But borrowck doesn't understand all containers:

```
let mut arr = [1, 2, 3];
let a = &mut arr[0];
let b = &mut arr[1]; // Error! Can't borrow arr mutably twice
```

Working with Disjoint Slices

```rust
let mut arr = [1, 2, 3, 4, 5];

// split_at_mut creates two distinct mutable slices
let (left, right) = arr.split_at_mut(2);

// Now we can modify both parts independently
left[0] += 10;
right[0] += 20;
```

How Mutable Iterators Work

Iterator API produces multiple values from a single `&mut self`:

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

For `&mut` iterators, this seems to create multiple `&mut`s to the same data, but:

- Iterators are one-shot (each element returned at most once)
- Carefully implemented to never alias mutable references

Stacked Borrows Model

What is Stacked Borrows?

- Formal memory model defining Rust’s aliasing rules
- Used by compiler developers and unsafe code authors
- Defines when two pointers can access the same memory
- Models memory access permissions using a stack structure
- MIRI (MIR Interpreter) implements and checks this model

The Permission Stack

For each memory location:

- Permissions are tracked in a stack
- Each new reference operation pushes to the stack
- Creating a `&mut` invalidates all other access
- When references go out of scope, they're popped
- Accessing memory requires appropriate permission

Stacked Borrows Example

```rust
let mut x = 10;
let r1 = &mut x;  // Stack: [r1]
*r1 = 20;         // OK, r1 has permission
let r2 = &*r1;    // Stack: [r1, r2]
println!("{}", *r2); // OK, r2 has permission
*r1 = 30;         // Stack: [r1], r2 invalidated
// Using r2 here would be UB under stacked borrows
```

Two-Phase Borrows

Special case for patterns like:

```rust
let mut v = vec![1, 2, 3];
v.push(v.len()); // Calls v.len() then mutates v
```

- First phase: "Reserved" mutable borrow
- Second phase: Activated after shared borrows are done
- Allows interleaved shared and mutable borrows in specific cases

Best Practices

Avoiding Common Borrow Checker Fights

- Use cloning for complex ownership scenarios
- Structure code to make lifetime relationships clear
- Let ownership flow naturally through function calls
- Break problems into smaller, well-defined components
- Prefer consuming ownership to borrowing when reasonable

Fighting with the Borrow Checker?

Ask yourself:

- Is this actually a memory safety issue?
- Can I restructure my code to avoid the problem?
- Would using `Clone` simplify things?
- Is interior mutability (`RefCell`, `Mutex`) appropriate?
- Do I actually need references here?

Questions?