Advanced Rust - Lab 2: The Borrow Checker

Lukáš Hozda

2025

Introduction

Don Quixote against the borrow checker wind mills

Exercise 1: Understanding Lifetime Annotations (10 minutes)

Objective

Implement functions with complex lifetime relationships and understand their implications.

Instructions

Create the following functions with proper lifetime annotations:

1. A function that returns the longest slice among three input slices
2. A function that combines data from multiple sources with different lifetimes

Requirements

Implement the following signatures with appropriate lifetime annotations:

```
// return the longest of three string slices
fn longest_slice(x: &str, y: &str, z: &str) -> &str {
    // TODO: Implement
}

// storing references in a struct
struct MultiRef {
    name: /* reference to a string */,
    values: /* reference to a vector of integers */
}

impl MultiRef {
    // constructor that takes references with different lifetimes
    fn new(n: &str, v: &Vec<i32>) -> MultiRef {
        // TODO: Implement
    }

    // method that returns the first value if available
    fn first_value(&self) -> Option<&i32> {
        // TODO: Implement
    }
}
```

Questions to Consider

1. Why are explicit lifetime parameters necessary in these examples?

Exercise 2: Higher-Rank Trait Bounds (20 minutes)

Objective

Implement functions that work with higher-rank trait bounds for lifetime-agnostic callbacks.

Instructions

Create functions that:

1. Apply a callback to each element in a slice, where the callback works with any lifetime
2. Define a struct that holds a function accepting references of any lifetime

Requirements

```rust
// 1. A function that applies a transformation to each element in a slice
// The callback should work with any possible lifetime
fn transform_elements<T, F, O>(slice: &[T], callback: F) -> Vec<O>
where
    // TODO: Add appropriate HRTB bounds
{
    // TODO: Implement
}

// 2. A struct that holds a callback working with references of any lifetime
struct CallbackHolder<F> {
    callback: F,
}

impl<F> CallbackHolder<F> {
    fn new(callback: F) -> Self {
        CallbackHolder { callback }
    }

    // Call method that works with any reference
    fn call_with<T>(&self, value: &T) -> /* return type */
    where
        // TODO: Add appropriate bounds
    {
        // TODO: Implement
    }
}

// Example usage
fn example_usage() {
    // Example using transform_elements
    let numbers = vec![1, 2, 3, 4, 5];
    let squares = transform_elements(&numbers, |x| x * x);

    // Example using CallbackHolder
    let string_length = CallbackHolder::new(|s: &str| s.len());
```

```
    let len = string_length.call_with("hello");
    assert_eq!(len, 5);
}
```

Questions to Consider

1. Why do we need higher-rank trait bounds in these examples?
2. How does the syntax ‘for<'a>‘ differ from simply adding a lifetime parameter?

Exercise 3: Disjoint Borrowing Patterns (20 minutes)

Objective

Learn techniques for working with multiple mutable references safely.

Instructions

Implement the following structures and functions that demonstrate how to:

1. Simultaneously borrow different parts of a data structure
2. Split mutable collections to obtain multiple mutable references
3. Use interior mutability when appropriate

Requirements

```
// 1. A struct with methods that mutate different fields at the same time
struct Person {
    name: String,
    age: u32,
    address: String,
}

impl Person {
    // TODO: Implement a method that mutates both name and age simultaneously

    // TODO: Implement a method that mutates both name and address simultaneously
}

// 2. Function that processes different parts of a vector in parallel
fn process_halves(data: &mut Vec<i32>) {
    // TODO: Split the vector into two parts and modify them independently
}

// 3. A safe API for a matrix that allows mutating different rows simultaneously
struct Matrix<T> {
    data: Vec<Vec<T>>,
    rows: usize,
    cols: usize,
}

impl<T> Matrix<T> {
    fn new(rows: usize, cols: usize, default_value: T) -> Self
    where
        T: Clone
    {
```

```
        // TODO: Implement
    }

    // Get mutable references to two different rows
    fn get_two_rows_mut(&mut self, row1: usize, row2: usize) -> Option<(&mut Vec<T>, &mut Vec<T>)> {
        // TODO: Implement - return None if row1 == row2 or either is out of bounds
    }
}
```

Questions to Consider

1. How does the borrow checker understand when references are disjoint?

Exercise 4: Diagnosing and Fixing Borrow Checker Errors (20 minutes)

Objective

Identify and fix common borrow checker errors in code examples.

Instructions

For each code snippet below:

1. Identify why the code doesn't compile
2. Fix the code to satisfy the borrow checker
3. Explain your solution

Code Snippet 1: Dangling References

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}

fn main() {
    let word;
    {
        let s = String::from("hello world");
        word = first_word(&s);
    }
    println!("the first word is: {}", word);
}
```

Code Snippet 2: Multiple Mutable Borrows

```rust
fn main() {
    let mut v = vec![1, 2, 3, 4];
    let first = &mut v[0];
    let last = &mut v[v.len() - 1];

    *first += 10;
    *last += 20;

    println!("First: {}, Last: {}", first, last);
    println!("Vector: {:?}", v);
}
```

Code Snippet 3: Moving a Value While Borrowed

```rust
struct Counter {
    count: usize,
}

impl Counter {
    fn new() -> Self {
        Counter { count: 0 }
    }

    fn increment(&mut self) {
        self.count += 1;
    }

    fn count(&self) -> usize {
        self.count
    }
}

fn main() {
    let mut counter = Counter::new();
    let count_ref = &counter.count;

    counter.increment();
    println!("Count via reference: {}", count_ref);
    println!("Count via method: {}", counter.count());
}
```

Code Snippet 4: Self-Referential Struct

```rust
struct Parser {
    data: String,
    current_position: usize,
    // This field tries to point into the data field
    current_token: Option<&str>,
}

impl Parser {
    fn new(data: String) -> Self {
        let mut parser = Parser {
```

```
            data,
            current_position: 0,
            current_token: None,
        };

        if !parser.data.is_empty() {
            // Try to set current_token to the first character of data
            parser.current_token = Some(&parser.data[0..1]);
        }

        parser
    }
}

fn main() {
    let parser = Parser::new(String::from("hello"));
    println!("Token: {:?}", parser.current_token);
}
```