Lifetimes

Performant Software Systems with Rust — Lecture 9

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Lifetimes — a First Cut

- Every **reference** has a lifetime
  - Typically (and in early versions of Rust), it’s the scope for which the reference is valid
  - We will see variations soon
- Just like type inference, lifetimes are **inferred** by the compiler in most cases
- But also like type annotation, we must annotate lifetimes when inference is not possible

Consider This Example

```rust
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }

    println!("r: {r}");
}
```

What will happen at compile-time?

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
5 |           let x = 5;
  |               - binding `x` declared here
6 |           r = &x;
  |               ^^ borrowed value does not live long enough
7 |       }
  |       - `x` dropped here while still borrowed
8 |
9 |       println!("r: {r}");
  |                  --- borrow later used here
```

The Borrow Checker

```rust
fn main() {
    let r;                   // ---------+-- 'a
    {                        //          |
        let x = 5;           // -+-- 'b  |
        r = &x;              //  |       |
    } // x goes out of scope // -+       |
                             //          |
    println!("r: {r}");      //          |
}                            // ---------+
```

- This code is rejected at compile-time because `x`'s lifetime, `'b`, is not as long as `r`'s lifetime, `'a`
- Or, as the compiler says, *x does not live long enough!*

Lifetime Annotations in Functions

```rust
fn main() {
    let s1 = String::from("abcd");
    let s2 = "xyz";

    // `longest()` takes string slices as we don't want it
    // to take ownership
    let result = longest(s1.as_str(), s2);
    println!("The longest string is {result}");
}
```

Implementing `longest()`

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
  signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

Lifetime Annotations

- We need to define the **relationship between the references** using **lifetime annotations**, so the borrow checker can perform its analysis
- **Lifetime annotations** don’t change how long any of the references live — they are just hints to the **borrow checker**

```
&i32         // a reference
&mut i32     // a mutable reference
&'a i32      // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

Revisiting `longest()`

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
// the returned reference will live as long as 'a
// or, the returned reference will be valid as long as both the
// parameters are valid
// or, the returned reference cannot outlive either x or y
// or, the lifetime of the returned reference is the same as the
// smaller of the lifetimes of the two references passed in
    ...
}
```

The Borrow Checker: Working with Annotated Lifetimes

```rust
fn main() {
    let s1 = String::from("long string is long");

    {
        let s2 = String::from("xyz");
        let result = longest(s1.as_str(), s2.as_str());
        println!("The longest string is {result}");
    }
}
```

```
The longest string is long string is long
()
```

The Borrow Checker: Working with Annotated Lifetimes

```rust
fn main() {
    let s1 = String::from("long string is long");
    let result;
    {
        let s2 = String::from("xyz");
        result = longest(s1.as_str(), s2.as_str());
    }

    println!("The longest string is {result}");
}
```

```
error[E0597]: `s2` does not live long enough
  --> src/main.rs:14:39
   |
13 |         let s2 = String::from("xyz");
   |             -- binding `s2` declared here
14 |         result = longest(s1.as_str(), s2.as_str());
   |                                       ^^ borrowed value does not live long
15 |     }
   |     - `s2` dropped here while still borrowed
16 |     println!("The longest string is {result}");
   |                                     --------- borrow later used here
```

Lifetime Annotations in Structs

If a **struct** holds references, we need a lifetime annotation for each reference

```rust
struct ImportantExcerpt<'a> {
    // an instance of `ImportantExcerpt` can't outlive the reference
    // it holds in `part`
    part: &'a str,
}

fn main() {
    let novel = String::from("Rust vs. C++");
    let first_sentence = novel.split('.').next().unwrap();
    let i = ImportantExcerpt {
        part: first_sentence,
    };
    println!("{}", i.part);
}
```

Lifetime Elision

- Each input parameter gets its own lifetime
- If there is exactly one input lifetime parameter, its lifetime is assigned to all output lifetime parameters
- If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime parameters

Lifetime Elision: Example

```rust
fn first_word(s: &str) -> &str {...}
```

↓

```rust
fn first_word<'a>(s: &'a str) -> &'a str {...}
```

Lifetime Annotations in Method Definitions

```rust
struct ImportantExcerpt<'a> {
    part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {}
}
```

↓

```rust
impl<'a, 'b> ImportantExcerpt<'a, 'b> {
    fn announce_and_return_part(&'a self, announcement: &'b str)
        -> &'a str {}
}
```

The `'static` Lifetime

- A special lifetime that tells the compiler that the reference can live for the entire duration of the program
- All string literals have the `'static` lifetime, as they are in **static memory** (or the **data** segment)
- Watch out on following the compiler's suggestions — do not use `'static` lifetimes if you don't know what you are doing

```rust
let s: &'static str = "I have a static lifetime.";
```

Anonymous Lifetimes

- The compiler should try to infer the lifetime annotation by itself
- It is typically for simplifying the grammar when
  - writing `impl` blocks
  - returning structs and enums with annotated lifetimes

```rust
struct ImportantExcerpt<'a> {
    // an instance of `ImportantExcerpt` can't outlive the reference
    // it holds in `part`
    part: &'a str,
}

// impl<'a> ImportantExcerpt<'a> {
impl ImportantExcerpt<'_> {
    fn print(&self) {
        println!("{}", self.part);
    }
}
```

```rust
// impl<'a> ImportantExcerpt<'a> {
impl ImportantExcerpt<'_> {
    fn print(&self) {
        println!("{}", self.part);
    }

    // fn get_part<'a>(&self) -> &'a str {
    fn get_part(&self) -> &'_ str {
        self.part
    }
}
```

Lifetimes **from scratch** again: an **in-depth** coverage

Let's revisit our first example:

```rust
fn main() {
    let r;                    // ---------+-- 'a
    {                         //          |
        let x = 5;            // -+-- 'b  |
        r = &x;               //  |       |
    } // x goes out of scope // -+        |
                              //          |
    println!("r: {r}");       //          |
}
```

```
error[E0597]: `x` does not live long enough
```

There's just one problem, though.

Let's consider the following revised code:

```rust
fn main() {
    let r;                        // ---------+-- 'a
    {                             //          |
        let x = 5;                // -+-- 'b  |
        r = &x;                   //  |       |
    } // x goes out of scope // -+       |
                                  //          |
}
```

Now the code compiles successfully. But why?

Lifetime vs. Scope: Revised Code

```rust
fn main() {
    let r;
    {
        let x = 5;             // x is not a reference, no lifetime!
        r = &x;                // ---------+-- 'b
    } // x goes out of scope
}
```

Lifetime vs. Scope: Original Example

```rust
fn main() {
    let r;
    {
        let x = 5;            // x is not a reference, no lifetime!
        r = &x;               // ---------+-- 'b
    } // x goes out of scope // -+        |
    println!("r: {r}");      //           |
}
```

Let's Consider Another Example

```rust
fn main() {
    let foo = 1;
    let mut r;
    {
        let x = 5;
        r = &x;

        println!("r: {r}");
    }

    r = &foo;
    println!("r: {r}");
}
```

Now the code compiles successfully. But why?

Let’s Look At Lifetimes Again

```rust
fn main() {
    let foo = 1;
    let mut r;
    {
        let x = 5;
        r = &x;                // ---------+-- 'b
                               //          |
        println!("r: {r}");    // ---------+
    }                          // 'b is not alive
                               // 'b is not alive
    r = &foo;                  // ---------+-- 'b
    println!("r: {r}");        // ---------+
}
```

Let's Make Another Revision

```rust
fn main() {
    let foo = 1;
    let r;
    {
        let x = 5;
        r = &foo;            // ---------+-- 'b
                             //          |
        println!("r: {r}");  // ---------+
    }                        //          |
                             //          |
    println!("r: {r}");      // ---------+
}
```

The revised code also compiles successfully. But why?

Let's Look At a Third Example

```rust
use rand::Rng;

fn main() {
    let mut rng = rand::rng();
    let mut x = String::from("Hello");
    let random_float: f64 = rng.random();
    let r = &x; // -----+- 'b, immutable borrow on x
    //      |
    if random_float > 0.5 {
        // 'b is not alive
        x.push_str(" World!"); // 'b is not alive, mutable borrow on x
    } else {
        //      |
        println!("{r}"); // -----+- 'b
    }
}
```

This example compiles and runs successfully!

But What If We Add One Line of Code?

```rust
use rand::Rng;

fn main() {
    let mut rng = rand::rng();
    let mut x = String::from("Hello");
    let random_float: f64 = rng.random();
    let r = &x;                         // -----+- 'b, immutable borrow on x
                                        //      |
    if random_float > 0.5 {             // 'b is alive
        x.push_str(" World!");          // 'b is alive, mutable borrow on x
        println!("{r}");                // immutable borrow!
    } else {                            //      |
        println!("{r}");                // -----+- 'b
    }
}
```

```
error[E0502]: cannot borrow `x` as mutable because it is also
borrowed as immutable
  --> src/main.rs:11:9
   |
7  |     let r = &x;
   |             -- immutable borrow occurs here
...
11 |         x.push_str(" World!");
   |         ^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
12 |         println!("{r}");
   |                   --- immutable borrow later used here
```

The data flow model

— Chapter 1, Foundations, in Rust for Rustaceans, Jon Gjengset

References

- There are two kinds of references
  - **Shared reference**: `&`
  - **Mutable reference**: `&mut`
- All references obey the following rules
  - A reference cannot outlive its referent
  - A mutable reference cannot be **aliased**
- But what do we mean by **aliasing**?

Aliasing

Variables and pointers **alias** if they refer to overlapping regions of memory

Consider the following code:

```rust
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
    // `*output` will be `2` if `input > 10`
}
```

Can the Rust compiler optimize the previous code to the following?

```rust
fn compute(input: &u32, output: &mut u32) {
    let cached_input = *input; // keep `*input` in a register
    if cached_input > 10 {
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

Alias Analysis Helps

- We used the fact that `&mut u32` can’t be aliased to prove that writes to `*output` can’t possibly affect `*input`
- This lets us cache `*input` in a register, eliminating a read
- **Alias analysis** lets the compiler perform useful optimizations!

But should we be concerned with aliasing in the following modified code?

```rust
fn compute(input: &u32, output: &mut u32) {
    let mut temp = *output;
    if *input > 10 {
        temp = 1;
    }
    if *input > 5 {
        temp *= 2;
    }
    *output = temp;
}
```

No. `input` doesn’t alias `temp`, because the value of a local variable can’t be aliased by things that existed before it was declared.

The definition of alias in Rust needs to involve some notion of liveness and mutation — we don’t actually care if aliasing occurs if there aren’t any actual writes to memory happening

Now Let’s Revisit Lifetimes

- A **Lifetime** involves **named** **regions of code** that a reference must be valid — **alive** — for
- A lifetime corresponds to a **path of execution**, with potential holes in them
- In most cases, a reference’s lifetime coincides to its scope
- Most lifetimes in Rust are inferred by the compiler

```rust
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

And the compiler sees

```rust
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

```
error[E0502]: cannot borrow `data` as mutable because it is also
borrowed as immutable
```

Recommended two-hour video: Crust of Rust: Lifetime Annotations, **Jon Gjengset**

Required Additional Reading

The Rust Programming Language, Chapter 10.3

The Rustonomicon, Chapter 3.1 - 3.3