Lecture 8: Lifetimes

Barinov Denis

March 23, 2023

barinov.diu@gmail.com

Lifetimes

Lifetimes

A lifetime is a construct the compiler (or more specifically, its borrow checker) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.

Take, for example, the case where we borrow a variable via &. The borrow has a lifetime that is determined by where it is declared. As a result, the borrow is valid as long as it ends before the lender is destroyed. However, the scope of the borrow is determined by where the reference is used.

Lifetimes: example

Lifetimes are annotated below with lines denoting the creation and destruction of each variable.**i** has the longest lifetime because its scope entirely encloses both **borrow1** and **borrow2**. The duration of **borrow1** compared to **borrow2** is irrelevant since they are disjoint.

```
fn main() {
    let i = 3; // Lifetime for `i` starts.
    {
        let borrow1 = &i; // `borrow1` lifetime starts.
        println!("borrow1: {}", borrow1); //
    } // `borrow1` ends.
    {
        let borrow2 = &i; // `borrow2` lifetime starts.
        println!("borrow2: {}", borrow2); //
    } // `borrow2` ends.
}   // Lifetime ends.
```

Lifetimes

The borrow checker uses explicit lifetime annotations to determine how long references should be valid. In cases where lifetimes are not elided, Rust requires explicit annotations to determine what the lifetime of a reference should be. The syntax for explicitly annotating a lifetime uses an apostrophe character as follows:

```
foo<'a, 'b>
// `foo` has lifetime parameters `'a` and `'b`
```

In this case, the lifetime of foo cannot exceed that of either **'a** or **'b**.

Lifetimes: Explicit annotation

```
// These two lifetimes must both be at
// least as long as the function `print_refs`.
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

// A function which takes no arguments, but has a lifetime parameter `'a`.
fn failed_borrow<'a>() {
    let _x = 12;

    // ERROR: `_x` does not live long enough
    let y: &'a i32 = &_x;
    // Lifetime of `&_x` is shorter than that of `y`.
    // A short lifetime cannot be coerced into a longer one.
}
```

Lifetimes: Explicit annotation

```
fn main() {
    // Create variables to be borrowed below.
    let (four, nine) = (4, 9);

    // Borrows (`&`) of both variables are passed into the function.
    print_refs(&four, &nine);
    // Any input which is borrowed must outlive the borrower.
    // In other words, the lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.

    failed_borrow();
    // `failed_borrow` contains no references to force `'a` to be
    // longer than the lifetime of the function, but `'a` is longer.
    // Because the lifetime is never constrained, it defaults to `'static`.
}
```

Lifetimes

Function signatures with lifetimes have a few constraints:

- any reference must have an annotated lifetime
- any reference being returned must have the same lifetime as an input or be static

Additionally, note that returning references without input is banned if it would result in returning references to invalid data

Lifetimes: Functions

```
// One input reference with lifetime `'a` which must live
// at least as long as the function.
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x is {}", x);
}

// Mutable references are possible with lifetimes as well.
fn add_one<'a>(x: &'a mut i32) {
    *x += 1;
}
```

Lifetimes: Functions

```
// Multiple elements with different lifetimes
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

// Returning references that have been passed in is acceptable.
// However, the correct lifetime must be returned.
fn pass_x<'a, 'b>(x: &'a i32, _: &'b i32) -> &'a i32 { x }

fn invalid_output<'a>() -> &'a String { &String::from("foo") }
// The above is invalid: `'a` must live longer than the function.
// Here, `&String::from("foo")` would create a `String`, followed by a
// reference. Then the data is dropped upon exiting the scope, leaving
// a reference to invalid data to be returned.
```

Lifetimes: Methods

```
struct Owner(i32);

impl Owner {
    fn add_one<'a>(&'a mut self) { self.0 += 1; }
    fn print<'a>(&'a self) {
        println!("`print`: {}", self.0);
    }
}

fn main() {
    let mut owner = Owner(18);

    owner.add_one();
    owner.print();
}
```

Lifetimes: Structs

```
// A type `Borrowed` which houses a reference to an `i32`.
// The reference to `i32` must outlive `Borrowed`.
struct Borrowed<'a>(&'a i32);

// Similarly, both references here must outlive this structure.
struct NamedBorrowed<'a> {
    x: &'a i32,
    y: &'a i32,
}

// An enum which is either an `i32` or a reference to one
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}
```

Lifetimes: Traits

Annotation of lifetimes in trait methods basically are similar to functions. Note that impl may have annotation of lifetimes too.

```
// A struct with annotation of lifetimes.
struct Borrowed<'a> {
    x: &'a i32,
}

// Annotate lifetimes to impl.
impl<'a> Default for Borrowed<'a> {
    fn default() -> Self {
        Self {
            x: &10,
        }
    }
}
```

Lifetimes: Bounds

Just like generic types can be bounded, lifetimes (themselves generic) use bounds as well. The : character has a slightly different meaning here, but + is the same

The example below shows the above syntax in action used after keyword where:

- T: 'a: All references in T must outlive lifetime 'a.
- T: Trait + 'a: Type T must implement trait Trait and all references in T must outlive 'a.

Lifetimes: Traits

```
struct Ref<'a, T: 'a>(&'a T);
// `Ref` contains a reference to a generic type `T` that has
// an unknown lifetime `'a`. `T` is bounded such that any
// *references* in `T` must outlive `'a`. Additionally, the lifetime
// of `Ref` may not exceed `'a`.

// Here a reference to `T` is taken where `T` implements
// `Debug` and all *references* in `T` outlive `'a`. In
// addition, `'a` must outlive the function.
fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}
```

Lifetimes: Coercion

A longer lifetime can be coerced into a shorter one so that it works inside a scope it normally wouldn't work in. This comes in the form of inferred coercion by the Rust compiler, and also in the form of declaring a lifetime difference:

```
// Here, Rust infers a lifetime that is as short as possible.
// The two references are then coerced to that lifetime.
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b, 'b>` reads as lifetime `'a` is at least as long as `'b`.
// We take in an `&'a i32` and return a `&'b i32` as a result of coercion.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}
```

Lifetimes: Static

Rust has a few reserved lifetime names. One of those is 'static. You might encounter it in two situations:

```
// A reference with 'static lifetime:
let s: &'static str = "hello world";

// 'static as part of a trait bound:
fn generic<T>(x: T) where T: 'static {}
```

Both are related but subtly different and this is a common source for confusion.

Lifetimes: Static reference

```
// Make a constant with `'static` lifetime.
static NUM: i32 = 18;

// Returns a reference to `NUM` where its `'static`
// lifetime is coerced to that of the input argument.
fn coerce_static<'a>(_: &'a i32) -> &'a i32 {
    &NUM
}
```

Lifetimes: Static reference

```
fn main() {
    {
        let static_string = "I'm in read-only memory";
        println!("static_string: {}", static_string);
        // When `static_string` goes out of scope, the reference
        // can no longer be used, but the data remains in the binary.
    }
    {
        let lifetime_num = 9;
        // Coerce `NUM` to lifetime of `lifetime_num`:
        let coerced_static = coerce_static(&lifetime_num);
        println!("coerced_static: {}", coerced_static);
    }
    println!("NUM: {} stays accessible!", NUM);
}
```

Lifetimes: Static Bound

As a trait bound, it means the type does not contain any non-static references. Eg. the receiver can hold on to the type for as long as they want and it will never become invalid until they drop it.

It's important to understand this means that any owned data always passes a 'static lifetime bound, but a reference to that owned data generally does not:

Lifetimes: Static Bound

```
fn print_it( input: impl Debug + 'static ) {
    println!( "'static value passed in is: {:?}", input );
}

fn main() {
    // i is owned and contains no references, thus it's 'static:
    let i = 5;
    print_it(i);

    // oops, &i only has the lifetime defined by the scope of
    // main(), so it's not 'static:
    print_it(&i);
}
```

Lifetimes: Static Bound

The compiler will tell you:

```
error[E0597]: `i` does not live long enough
  --> src/lib.rs:15:15
   |
15 |     print_it(&i);
   |     ---------^^--
   |     |         |
   |     |         borrowed value does not live long enough
   |     argument requires that `i` is borrowed for `'static`
16 | }
   | - `i` dropped here while still borrowed
```

Lifetimes: Elision

Some lifetime patterns are overwhelmingly common and so the borrow checker will allow you to omit them to save typing and to improve readability. This is known as elision. Elision exists in Rust solely because these patterns are common.

```
// `elided_input` and `annotated_input` essentially have identical signature
// because the lifetime of `elided_input` is inferred by the compiler:
fn elided_input(x: &i32) {
    println!("`elided_input`: {}", x);
}

fn annotated_input<'a>(x: &'a i32) {
    println!("`annotated_input`: {}", x);
}
```

Lifetimes: Elision

```
// Similarly, `elided_pass` and `annotated_pass` have identical signatures
// because the lifetime is added implicitly to `elided_pass`:
fn elided_pass(x: &i32) -> &i32 { x }

fn annotated_pass<'a>(x: &'a i32) -> &'a i32 { x }
```

Questions?