ECE326

PROGRAMMING LANGUAGES

Lecture 17 : Lifetime

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Lifetime

- The scope for which a reference is valid
- Usually implicit and inferred

```
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. ──────────────────────────────────────────────────────┘
```

Borrow Checker

- Verifies all borrows are definitely valid
- Lifetime annotation
  - Single quote followed by a letter. E.g. **‘a**

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5; // -+-- 'b         |
        r = &x;    // |              |
    }              // -+              |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

r has a lifetime of a, x has a lifetime of b. Because b is shorter than a, this borrow is rejected

Return by Reference

- Functions can take references and return references
- Example: longest
  - Returns longer of the two string slices

```
fn longest(x: &str, y: &str) -> &str {
      if x.len() > y.len() {
            x
      } else {
            y
      }
}
let string1 = String::from("abcd");
let string2 = "xyz";
let result = longest(string1.as_str(), string2);
println!("The longest string is {}", result);     // abcd
```

Lifetime

- What if input has different lifetime?
  - Danger! Returned reference may become invalid
  - However, lifetime information is lost after passing to function

```
fn main() {
        let string1 = String::from("long string is long");
        let result;

        {
                let string2 = String::from("xyz");
                result = longest(string1.as_str(),
                                 string2.as_str());
        }
        // invalid reference if string2 were returned
        println!("The longest string is {}", result);
}
```

Annotation

- Functions with references require lifetime annotations
- Generic lifetime parameter
  - Adds lifetime information to function signature

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

longest takes two parameters with the same lifetime, *a*, and returns a reference, also with the same lifetime *a*.

Borrow Checker

```
let string1 = String::from("long string is long");
let result;
{
      let string2 = String::from("xyz");
      result = longest(string1.as_str(), string2.as_str());
}
println!("The longest string is {}", result);

error[E0597]: `string2` does not live long enough
|
| result = longest(string1.as_str(), string2.as_str());
|                                    ------- borrow occurs here
| }
| ^ `string2` dropped here while still borrowed
| println!("The longest string is {}", result);
| }
| - borrowed value needs to live until here
```

Lifetime Elision

- Automatically inferring lifetime of returned references
- Rule 1
  - If function takes one parameter by reference, it has the same lifetime as returned reference

```
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[..]
}
```

Rule 1 applies. Do not need to explicitly annotate lifetime.

Lifetime Elision

- Rule 2
  - A *method*'s returned reference has the same lifetime as *self*
    - Compiler error if the method returns other parameters by reference

```
struct ImportantExcerpt {
    part: String,
}

impl ImportantExcerpt {
    fn announce_and_return_part(&self, message: &str) -> &str {
        println!("Attention please: {}", message);
        self.part.as_str()
    }
}
```

Rule 2 applies. Do not need to explicitly annotate lifetime.

Structure

- References in structures must be annotated

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

impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 { 3 }
}

fn main() {
    let novel = String::from("Hello World. A long time ago...");
    let first = novel.split('.').next().expect("Split failed");
    let i = ImportantExcerpt { part: first };
}
```

The structure's impl blocks also needs to have lifetime annotation.

Static Lifetime

- References lives for entire duration of program
- Static global variables
  - E.g. all string literals

```
enum Either<'a> {
      Num(i32),
      Ref(&'a i32),
      StaticRef(&'static str),
}

let x = 18;
let reference = Either::Ref(&x);
let number = Either::Num(15);
let staticref = Either::StaticRef("bonjour");
```

If any variant has a reference, the entire enum must have a generic lifetime parameter

Lifetime Coercion

- Allows returning reference with a shorter lifetime

```
// `<'a: 'b, 'b>` reads as lifetime `'a` is at least as long as `'b`.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32
{
        first
}

fn main()
{
        let first = 2; // Longer lifetime
        {
                let second = 3; // Shorter lifetime
                println!("{} is the first",
                                        choose_first(&first, &second));
        };
}
```