Lifetimes

CAS CS 392: Rust, in Theory and in Practice

Lecture 9

Outline

Ownership

1. Every value has one owner at a time
2. When the owner of a value goes out of scope, any memory associated with the value is freed

Working with just ownership is too restrictive:

```
fn is_empty(x : String) -> (bool, String) {
    (x.len() == 0, x)
}


fn main() {
    let x = String::from("foo");
    let (out, x) = is_empty(x);
    println!("is_empty({}) == {}", x, out)
}
```

References

1. A reference can't outlive it's referent
2. A mutable reference can't be *aliased*

This second rule says: **we can't have *any other* reference if we already have a mutable reference**

```
// THIS DOES NOT COMPILE
let mut x = String::from("foo");
let y = &mut x;         // mutable borrow
let z = &x;             // immutable borrow
y.push_str("bar");      // use of mutable borrow
println!("{z}")         // use of immutable borrow
```

Question

```
let mut x = String::from("foo");
let y = &mut x;         // mutable borrow
let z = &x;             // immutable borrow
y.push_str("bar");      // use of mutable borrow
println!("{z}")         // use of immutable borrow
```

How do we guarantee that these rules are followed?

The Borrow Checker

The **borrow checker** of Rust verifies that references satisfy the two rules of references:

- It checks that references *live* longer than their referents
- It checks that no value is borrowed if it already mutably borrowed
- It checks that no variable is moved if it is borrowed

Dangling References

A **dangling reference** is a reference to memory which is no longer valid

It's easy to create dangling pointer in C:

```c
int main(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 2;
  free(x);
  printf("%d\n", *x);
  return 0;
}
```

It's *impossible* to create a dangling reference in Rust

Data Races

```
// THIS DOES NOT COMPILE
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

A **data race** occurs when multiple threads access the same data, and one writes (and there's no synchronization)

Rust's borrow system ensures that there are no data races

Compiler Optimizations

No-aliasing also allows for better compiler optimizations:

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

Optimized (morally speaking):

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

Aside: Linearity vs. Affinity vs. References

Linearity in the context of typing refers to **resource consumption**. It has to do with *moving* values

Strictly speaking rust is *affine* (not linear) with respect to its resource consumption, we are allowed to drop values, but we can't copy them for free:

```
fn drop(x : String) {
    // x is consumed and discarded
}
```

Liftimes and references are not related to resource consumption, but what we can do with a value before it is consumed. In particular, **borrows do not break affinity**

Outline

References and Borrowing

Lifetimes

Lifetimes

A **lifetime** is a (labeled) region of code. In loose terms, lifetimes are *scopes*, and include

- scoping blocks
- function scopes
- variable scopes
- if-expression branches

Lifetimes are often associated with their labels since there is a one-to-one correspondence

A lifetime is notated as `'a`, `'b`, etc., a tick mark followed by a (short) lowercase name

Lifetimes and References

The **lifetime** of a reference is the scope (i.e., a lifetime in the "region" sense) in which the reference is valid

All references are implicitly annotated with a lifetime (label):

- `&'a T`
- `&'b mut T`

In nearly all cases, we *never* have to annotate references with lifetimes (in most cases, it's not possible)

Example

```
// THIS DOES NOT COMPILE
fn main() {
    let r;                      // ----------+-- 'a
                                //           |
    {                           //           |
        let x = 5;              // -+-- 'b   |
        r = &x;                 //  |        |
    }                           // -+        |
                                //           |
    println!("r: {r}");         //           |
}                               // ----------+
```

RPL visualizes lifetimes according to these diagrams

Example (Continued)

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

Removing the scoping block ensures the lifetime of `r` is contained in that of `x`, so that `x` must outlive `r`

Another Example

```
// THIS DOES NOT COMPILE
fn main() {
    let mut x : Option<&String> = None;
    if x.is_none() {
        let y : String = String::from("foo");
        x = Some(&y);
    }
    println!("{}", x.unwrap());
}
```

Things like if-expressions also define scopes, so this is the same kind of example

Lifetime Annotations in Functions

The situation gets complicated when we use references in functions:

```
// THIS DOES NOT COMPILE
fn if_then_else<T>(b : bool, x : &T, y : &T) -> &T {
    if b { x } else { y }
}

fn main() {
    let x : String = String::from("foo");
    let y : String = String::from("foo");
    println!("{}", if_then_else(true, &x, &y));
}
```

The lifetimes of references are no longer *contained* in the scope of the function. In fact, the lifetime of a parameter which is a reference **must contain the lifetime of the function.**

Lifetime Annotations in Functions

Turns out we can annotate this function with lifetime labels:

```
fn if_then_else<'a, 'b, T>(
    b : bool, x : &'a T, y : &'b T
) -> &??? T {
    if b { x } else { y }
}
```

And we can see the potential problem: **How long does the output live?** (It depends on `b`)

Just like we read type generic function as: *for any type `T`, this function takes a reference to a `T`...*

We read *lifetime* generic functions as: *for any lifetimes `'a` and `'b`, this function take a reference with which lives at least as long as `'a`...*

Lifetime Annotations and Functions

```
fn if_then_else<'a, 'b, T>(
    b : bool, x : &'a T, y : &'b T
) -> &??? T {
    if b { x } else { y }
}

fn main() { // 'a
    let x : String = String::from("foo");
    { // 'b
        let y : String = String::from("bar");
        let z : &String = if_then_else(true, &x, &y);
        // does z live for 'a or 'b?
        println!("{}", z)
    }
}
```

Given references with different lifetimes, we can't know at compile time how long the resulting reference will live.

Lifetime "Subtyping" Relation

We've been careful to say: lives "at least as long as"

The "at least as" implies a subtyping relation: If `x` is of type `&'a T` and `'b` is contained in `'a`, then `x` is also of type `&'b T`.

Important: This does not affect the *semantics* of the program. If we type a value of having a shorter lifetime, that doesn't change how long it's valid.

We can use values of longer lifetimes, but not shorter:

```
fn if_then_else<'a, T>(
    b : bool, x : &'a T, y : &'a T
) -> &'a T {
    if b { x } else { y }
}
```

Lifetime Annotations in Structures

We are required to annotate references in structures with lifetimes.

```
struct Foo<'a> {
    foo: &'a mut String,
}

fn main() {
    let mut x = String::from("foo");
    let foo = Foo { foo: &mut x };
    *foo.foo = String::from("bar");
    println!("{}", foo.foo)
}
```

The annotation <'a> implies that we cannot build a structure that outlives the references it holds, because the lifetime is determined by the scope in which the structure is created

Lifetime Annotations in Structures

We cannot assign the reference to something with a shorter lifetime:

```
// THIS DOES NOT COMPILE
struct Foo<'a> {
    foo: &'a mut String,
}

fn main() {
    let mut x = String::from("foo");
    let mut foo = Foo { foo: &mut x };
    {
        let mut y = String::from("bar");
        foo.foo = &mut y;
        println!("{}", foo.foo)
    }
    *foo.foo = String::from("bar");
    println!("{}", foo.foo)
}
```

Lifetime Annotations in Structures

Important (Again): This does not affect the semantics of our program:

```
struct Foo<'a> {
    foo: &'a mut String,
}

fn main() { // 'a
    let mut x = String::from("foo");
    { // 'b
        let mut foo = Foo { foo: &mut x };
        foo.foo = &mut String::from("bar");
    }
}
```

This is possible because when the `Foo` is created, the lifetime is determined by the scope block (labeled with `'b`) and `x` has a longer lifetime, so `&mut x` can be typed as `&'b mut String`

Lifetime Elision

We don't need to annotate lifetimes in the case of non-generic functions because lifetimes are entirely determined by the structure of the program, they can be *inferred*

For generic functions we can avoid annotating with lifetimes because the compiler implements common patterns

Note: This is a *heuristic*. The compiler is not doing any smart lifetime inference

Lifetime Elision Rules

1. Each elided lifetime annotation is replaced with a *distinct* lifetime parameter
2. If a function's input has a single lifetime parameter then *all* elided lifetime parameters of the output should match this
3. If a function has a `self` reference, then the lifetime of `self` is used for all elided lifetime parameters in the output

Back to our example

```
fn if_then_else<'a, 'b, T>(
    b : bool, x : &'a T, y : &'b T
) -> &??? T {
    if b { x } else { y }
}
```

The first rule states that each input should get it's own fresh lifetime parameter

The compiler complains because the other two rules don't tell it what to assign to the final elided parameter

Again, Rust does not *guess* the parameter. It will only allow elision if the rules entirely specify the lifetime parameters

A Word of Warning

The notion of lifetimes in Rust is a *moving target* (My advisor used to say that working with new PLs is like the wild west)

It's entirely possible that these rules will be different in the near future

Lifetime Annotations in Methods

The third rule implies we almost never need to annotate lifetimes in methods:

```
struct Foo<'a> {
    foo : &'a str
}

impl<'a> Foo<'a> {
    fn bar(&self, s: &str) -> &str {
        println!("{s}");
        self.foo
    }
}
```

The Static Lifetime

Values which live the entirety of the program have `'static` lifetime:

```
struct Foo {
    foo : &'static str
}

fn main() {
    let x = Foo { foo: "foo" };
    println!("{}", x.foo)
}
```

This is true of constants, like string slices

Aliveness vs. Lifetime

There is an important distinction between the *lifetime* of a reference and how long it is *alive*:

```
let mut x : i32 = 1;
let y : &mut i32 = &mut x;
let z : &mut i32 = &mut x;
*z += 1;
println!("{}", z)
```

This code is fine even though it *looks* like we have two mutable references because `y` is not longer alive when `z` is defined

The borrow checker is smarter enough to know that `y` does not get *used* after `z` is created, even though technically the liftime of `y` extends to the end of its scope