Ensuring memory safety in rust: ownership, borrowing, and lifetime

petercommand

Why?

- Rust does not use garbage collection or reference count by default
- So…manual memory management?
- Even without these systems, you still have to deal with these issues, only that the compiler is not helping you

Introduction

- Examples on cases that you might got yourself into when using this system
- Ownership, Borrowing, and lifetime
- Local region inference
- A lot of examples examining the use of the different cases

Ownership

- Statically decide where an object is dropped
- What if I can’t statically decide when I should free the object? — retreat to reference count/GC
- Move semantic by default

Ownership

```
fn test() {
   let v = vec![1,2,3];
   let v2 = v; // you can no longer use v
   println!("{}", v); //use of moved value
}
```

```
fn test2() {
  let v = vec![1,2,3];
  f(v); //you can no longer use v
  println!("{}", v); //use of moved value
}
fn f(vec: Vec<i32>) { }
```

Ownership — Copy trait

- Copy semantic
- types like i32, u32, f32, f64, and bool implements the Copy trait

Ownership — Copy trait

```
fn test() {
   let v = 1i32;
   let v2 = v; //v is copied
   println!("{}", v); //v is still accessible
}
//v2 is dropped at the end of the scope
```

Ownership

- Even though we don’t want to copy a big object every time we use it, we also don’t want to hand out the ownership of an object every time we use it..
- This is where the borrowing system kicks in

Borrowing

- Borrow the ownership of the object to other function/object
- Pass by reference?

Borrowing

```
fn sum(vec: &Vec<i32>) -> i32 {
   vec.iter().fold(0, |acc, elem|  a + b)
}
fn sum2(vec: &Vec<i32>) -> i32 {
   let mut acc = 0;
   for i in vec {
     acc = acc + i;
   }
  acc
}
fn main() {
   let vec = vec![1,2,3];
   let r1 = sum(&vec);
   let r2 = sum2(&vec);//we can still use vec here
}
```

Borrowing —
Mutable References

- Data Race
- Prevent immutable reference from being modified
- Prevent borrowed object from being dropped

Borrowing —
Mutable References

- Immutable objects cannot be borrowed as mutable
- Borrowing rules:
-     Any number of immutable references.. OR
-     Exactly one mutable reference
- Mutable borrows in data structure — multiple mutable borrows from the same data structure?

Borrowing — Mutable References

```
fn main() {
    let mut x = vec![1,2,3];
    let y = &mut x;
    y.push(1);
    println!("{:?}", x);
}
```

- cannot borrow `x` as immutable because it is also borrowed as mutable

Borrowing — Mutable References

```
fn main() {
    let mut x = vec![1,2,3];
    {
        let y = &mut x;
        y.push(1);
    }
    println!("{:?}", x);
}
```

Borrowing — Mutable References

```
fn main() {
    let mut x = vec![1,2,3];
    let y = &mut x;
    x = vec![9,0]; //this drops the original object x points
                   //to and invalidates y
}
```

- cannot assign to `x` because it is borrowed
- `x` is freezed until any borrow is out of scope

Borrowing — Structs

- Only disjoints fields can be borrowed mutably simultaneously

```
struct Foo {
    a: Bar
}
struct Bar {
    b: i32
}
```

```
fn main() {
    let mut foo = Foo {
        a: Bar {
            b: 1
        }
    };
    let mut b_ref = &mut foo.a.b;
    let mut c_ref = &mut foo.a;
}
```

Borrowing — Move?

- You don’t have ownership!
- Cannot move out of borrowed context

```
fn f1(v: &Vec<i32>) {
    f2(*v);
}
fn f2(v: Vec<i32>) {}
```

Borrowing

- What if…the borrowed reference outlives the object?

```
fn main() {
    let v_ref : &i32;
    let v = 1;
    v_ref = &v;
    println!("v is: {}", v);
}
```

(Values in a scope are dropped
in the opposite order they are created)

Borrowing

- What if…the borrowed reference outlives the object?

```
fn main() {
  let vec_ref : &Vec<i32>;
  {
    let vec = vec![1,2,3];
    vec_ref = &vec;
  }
  println!("{}", vec_ref[0]);
  //error: `vec` does not live long enough
}
```

Lifetime

- Used to ensure the validity of references
- Prevent creating references that can potentially access dropped objects
- Use after free/dangling pointers

Lifetime

- Only input/output parameters needs lifetime annotation
- Local lifetimes are deduced using regional inference

Lifetime Annotation

```
struct Foo<‘a> {
   bar: &’a i32
   bar2: &’a i32
}
```

```
impl<‘a> Foo<‘a> {
   fn x(&self) -> &’a i32 {
      self.bar
   }
}
```

```
fn my_func<‘a>(t : &’a  Vec<i32>) {}
```

Lifetime Annotation

- Every reference in Foo should outlive Foo itself: ‘Foo must outlive ‘a and ‘b (‘Foo should live as least as long as ‘a and ‘b)

```
struct Foo<‘a, ‘b> {
   bar: &’a i32
   bar2: &’b i32
}
```

Lifetime

- We can see that in foo, x and y have a longer lifetime than foo, which is why the struct initialisation is valid.

```
struct Foo<‘a> {
  bar: &’a i32
  bar2: &’a i32
}
```

```
fn f() {
  let x = 32i32;
  let y = 64i32;
  let foo = Foo {
     bar : &x,
     bar2 : &y
  }
}
```

Lifetime

- Is there a lifetime ‘a such that both bar and bar2 outlives the lifetime ‘a and that ‘a outlives ‘foo?
- ‘x > ‘y > ‘foo
- choose the smallest ‘a such that ‘y >= ‘a >= ‘foo (therefore, ‘a is satisfiable)

```
fn f() {
    let x = 32i32;
    let y = 64i32;
    let foo = Foo {
        bar : &x,
        bar2 : &y
    }
}
```

```
struct Foo<‘a> {
    bar: &’a i32
    bar2: &’a i32
}
```

Lifetime — Covariance

- ‘a : ‘b means that ‘a outlives ‘b (‘b is a subtype of ‘a)
- *implies that* Foo<‘a> is a subtype of Foo<‘b> (covariance)

```
struct Foo<'a> {
    foo: &'a i32
}
fn test<'a, 'b>(foo: Foo<'a>)
   where 'a: 'b {
    let u: Foo<'b> = foo;
}
```

Lifetime — Invariance

- With internal mutability, only invariance is allowed
- Using shorter lifetime to approximate longer lifetime is unsound!

Lifetime — Invariance

```
use std::cell::Cell;
struct Foo<'a> {
    foo: Cell<&'a i32>
}
fn test<'a, 'b>(foo: &Foo<‘a>)
   where 'a: 'b {
    let u: &Foo<‘b> = foo; // ‘b does not outlive ‘a
    let short_i32 = 1;
    u.foo.set(&short_i32);
}
```

- Approximating ‘a using ‘b is unsound
- What if..we change ‘a: ‘b to ‘b: ‘a?

Lifetime — Invariance

```
use std::cell::Cell;
struct Foo<'a> {
    foo: Cell<&'a i32>
}
fn test<'a, 'b>(foo: &Foo<‘a>)
   where 'b: 'a {
    let u: &Foo<‘b> = foo; // ‘a does not outlive ‘b
    let short_i32 = 1;
    u.foo.set(&short_i32);
}
```

- Of course this doesn’t work :)
- Both ‘a: ‘b and ‘b: ‘a are necessary!

Lifetime — Invariance

```
use std::cell::Cell;
struct Foo<'a> {
    foo: Cell<&'a i32>
}
fn test<'a, 'b>(foo: &Foo<‘a>)
   where 'b: ‘a, ‘a: ‘b {
    let u: &Foo<‘b> = foo;
    let short_i32 = 1;
    u.foo.set(&short_i32);
}
```

- Both ‘a: ‘b and ‘b: ‘a are necessary!
- Invariance!

Lifetime — Invariance

```
use std::cell::Cell;
struct Foo<'a> {
    foo: Cell<&'a i32>
}
fn test<'a, 'b>(foo: Foo<‘a>)
   where ‘a: ‘b {
    let u: Foo<‘b> = foo;
    let short_i32 = 1;
    u.foo.set(&short_i32);
}
```

- The compiler still rejects the program, even though this use case is sound…

Lifetime — Invariance

- Traits are invariant in there types…so…this example is rejected by the compiler..Even though this is perfectly sound

```
fn test<'a>(foo: Box<Fn(Foo<'a>) -> i32>) {
    let t : Box<Fn(Foo<'static>) -> i32> = foo;
}
```

Lifetime — Invariance

- Is invariance too restrictive?
- Tradeoff between design simplicity and expressiveness

Lifetime — Contravariance

- fn(T) is contravariant on T
- Contravariant on input parameters
- Covariant on output parameters

```
static N: i32 = 10;
struct Foo<'a> {
    t: &'a i32
}
impl<'a> Foo<'a> {
    fn test(p: &'a i32) {}
    fn test2() {
        Foo::<'a>::test(&N);
    }
}
```

Determining The Variance of Datatype

- Recursively traverse a struct to determine the variance of a lifetime
- PhantomData<T> — its sole use is to guide lifetime inference on datatypes

Examples with Lifetimes

- Temporary object lifetime
- Trait object lifetime

Example — Temporary Object Lifetime

```
struct Foo {}
impl Foo {
    fn test(&self) -> &Self { self }
}
fn main() {
    let t = (Foo {}).test();
}//borrowed value does not live long enough
```

- Temporary objects only live until the end of the innermost enclosing statement
- Kind of Annoying!

Example — Temporary Object Lifetime

```
struct Foo {}
impl Foo {
    fn test(&self) -> &Self { self }
}
fn main() {
    let foo = Foo {};
    let t = foo.test();
}
```

- Fix the issue by splitting the original statement into two statements

Example — Trait Object Lifetime

- The scope of the borrow is affected by lifetime!

```
struct TestImpl {}
trait Test<'a> {
    fn test(&'a self);
}
impl<'a> Test<'a> for TestImpl {
    fn test(&'a self) {}
}
fn main() {
    let imp: Box<Test> = Box::new(TestImpl {});
    imp.test();
}//`*imp` does not live long enough
```

Example — Trait Object Lifetime

```
struct TestImpl {}
trait Test<'a> {
    fn test<‘b>(&'b self) where ‘a: ‘b;
}
impl<'a> Test<'a> for TestImpl {
    fn test<‘b>(&'b self) where ‘a: ‘b {}
}
fn main() {
    let imp: Box<Test> = Box::new(TestImpl {});
    imp.test();
}
```

Example — Trait Object Lifetime

```
struct TestImpl {}
trait Test {
    fn test<‘a>(&'a self);
}
impl Test for TestImpl {
    fn test<‘a>(&'a self) {}
}
fn main() {
    let imp: Box<Test> = Box::new(TestImpl {});
    imp.test();
}
```

References

- https://doc.rust-lang.org/nomicon/subtyping.html
- https://github.com/rust-lang/rfcs/blob/master/text/0738-variance.md
- https://github.com/rust-lang/rfcs/pull/738
- https://internals.rust-lang.org/t/refining-variance-and-traits/1787
- https://github.com/rust-lang/rust/blob/master/src/librustc_typeck/check/regionck.rs