Written Assignment 5

CS 538, Spring 2020

Vinay Patil

1 The Borrow Checker (15)

(a)
```
fn foo(s: String) { /* ... */ }

fn baz() {
    let str_1 = String::from("Hello");
    let str_2 = str_1;

    foo(str_1);
}
```

Solution:

1. Here inside the function baz, there is a function call to foo with str_1 as an argument, but str_1 doesn't own the string anymore (In other words the ownership has been moved to str_2) and hence using str_1 is not acceptable since it doesn't own anything and the function foo is expecting a owned string as an argument.

2. If Rust had accepted the program without any check, then at the end of foo function, the moved string would be de-allocated assuming that s had owned the data and this would raise a runtime error as double de-allocation occurs once str_2 goes out of scope as well.

(b)
```
fn foo(s: &mut String) { /* ... */ }

fn baz() {
    let my_str = String::from("Hello");
    let my_ref_one: &String = &my_str;
    let my_ref_two: &String = &my_str;

    foo(&my_ref_two);

    /* ... */
}
```

Solution:

1.Here my_ref_two is an immutable reference to my_str. The function foo expects a mutable reference but instead is given my_ref_two as an argument which will cause the compiler to complain.

2.If Rust had accepted this, then within function foo, the variable s would have been used to manipulate the underlying string without the ability to do so and hence would raise a runtime error.

(c)
```
fn baz() {
    let mut my_str = String::from("Hello");
    let my_ref = &my_str;

    my_str.push_str(" World!");
    println!("What's here? {}", my_ref);
}
```

Solution:

1. Here my_ref is created as an immutable reference and another mutable reference has been created (implicitly) when push_str function is called (Since push_str expects a mutable reference as an argument). This is in violation of Rust's golden rule - both a mutable and immutable reference cannot exist at the same time.

2. If Rust had accepted this, then the data what my_ref is pointing to would be changed without it's knowledge(Even invalidated in some cases) and end up printing the value which was not expected.

(d)
```
fn foo(s: &mut String) { /* ... */ }

fn baz() {
    let mut my_str = String::from("Hello");
    let my_ref = &my_str;
    let my_other_ref = &mut my_str;

    foo(my_other_ref);

    println!("What's here? {}", my_ref);
}
```

Solution:

1. Here my_ref is created as an immutable reference and another mutable reference called my_other_ref has been created. This is in violation of Rust's golden rule - both a mutable and immutable reference cannot exist at the same time.

2. If Rust had accepted this, then the data what my_ref is pointing to would be changed(within function foo) without it's knowledge(Even invalidated in some cases) and end up printing the value which was not expected.

(e)
```
struct MyStruct {
    my_str: String,
    my_int: i32,
}

fn foo(s: String) { /* ... */ }

fn baz() {
    let my_struct = MyStruct {
        my_str: String::from("Hello"),
        my_int: 42,
    };

    foo(my_struct.my_str);
    foo(my_struct.my_str);
}
```

Solution:

1. In the first foo function call, the ownership of my_struct.my_str has been moved to s in foo(Since foo expects an owned string) and hence when a second foo function call occurs, the program tries to access my_struct.my_str which doesn't exist anymore and hence the compiler complains.

2. If Rust had accepted this, then one of the problems here would be that, the same string my_struct.my_str would be de-allocated twice - once at the end of foo function and the other at the end of baz function which would be a run time error. The other minor problem would be that, the second call ideally expects the string to hold the value "Hello" but that would have been changed to something else in the first call to the foo function (Or may have been invalidated in some cases).

(f)
```
fn foo(s: String) { /* ... */ }

fn bar(s: &String) {
    let my_str: String = *s;

    foo(my_str);
}
```

Solution:

1. Inside bar function there is an attempt to move the string pointed to by s (immutable reference) into my_str variable, which is basically an ownership transfer (accompanied by de-referencing as an intermediate step) from the underlying string to my_str which is not possible using an immutable reference and hence the compiler complains.

2.If Rust had accepted this, one of the problems would be of double freeing, because s inside foo(since it expects an owned string) and the original owner of the string will both try to free the string at the end of their scope and hence will end up causing a run time error in the program execution.

(g)
```
fn foo(s: String) { /* ... */ }

fn bar(opt: Option<String>) {
    match opt {
        None => println!("Nothing!"),
        Some(my_str) => foo(my_str)
    }

    let opt_s = opt.unwrap_or(String::new());

    println!("What's here? {}", opt_s);
}
```

Solution:

1. Within the match block owned value of opt is used and hence the ownership of the string within opt is moved to my_str which is sent as an argument to function foo(Which accepts a owned string value). Hence the compiler complains when in the line after the match block, the program tries to use unwrap on opt which doesn't own the string.

2. If Rust accepted this program, then unwrap instruction would end up with some garbage value in opt_s variable. Also, there exists the risk of double freeing as well.

(h)
```
fn foo<'a>(s: String) -> &'a String {
    &s
}
```

Solution:

1. Here the function is trying to return data (string s) that is owned by the function(as it accepts owned string as an argument). This is restricted by the compiler, hence it complains.

2. If Rust accepted this program, then we know that since s will be de-allocated at the end of the function definition, &s which is the reference to the string s will become a dangling pointer(pointing to nothing) and hence will raise a run-time error when used.

(i)
```
fn foo(s: &mut String) -> &mut String {
    s.push_str(" World!");
    s
}

fn baz() {
    let mut dummy = String::new();
    let mut str_ref = &mut dummy;

    {
        let mut my_str = String::from("Hello");
        str_ref = foo(&mut my_str);
    }

    println!("What's here? {}", str_ref);
}
```

Solution:

1. Here my_str is given a block scope (smaller life time) and that is the only place where the variable owns the data and after which it will de-allocate it. So when there is an attempt to assign this to str_ref which has a scope(lifetime) even outside the block the compiler complains.

2. If Rust accepted this program, then once the program exists the block, my_str will be de-allocated and str_ref will end up pointing to a junk value.

(j)
```
fn foo<'a>(s: &'a String, t: &'a String) -> &'a String {
    if s > t { s } else { t }
}

fn baz() {
    let mut foo1 = String::from("Hello");

    let mut str_ref: &String;

    {
        let mut foo2 = String::from("World");
        str_ref = foo(&foo1, &foo2);
    }

    println!("What's here? {}", str_ref);
}
```

Solution:

1. Here foo2 is given a block scope (smaller life time) and that is the only place where the variable owns the data and after which it will de-allocate it. So when there is an attempt to possibly assign this to str_ref which has a scope(lifetime) even outside the block the compiler complains.

2. If Rust accepted this program, then once the program exists the block, foo2 will be de-allocated and str_ref will end up pointing to a junk value if foo1 was less than foo2.

2 Lambda Calculus with References (15)

2.1 Stepping expressions

(a) $let\ x = new(42)\ in\ *x := *x + 1$

$$
\begin{aligned}
&(let\ x = new(42)\ in\ *x := *x + 1, [\,]) \\
\rightarrow &(let\ x = \#0\ in\ *x := *x + 1, [42]) \\
\rightarrow &(*(\#0) := *(\#0) + 1, [42]) \\
\rightarrow &(*(\#0) := 42 + 1, [42]) \\
\rightarrow &(*(\#0) := 43, [42]) \\
\rightarrow &((), [43])
\end{aligned}
$$

(b) $let\ x = new(7)\ in\ (let\ y = new(0)\ in\ *x := 0\ ;\ *y := *y + 1)$

$$
\begin{aligned}
&(let\ x = new(7)\ in\ (let\ y = new(0)\ in\ *x := 0\ ;\ *y := *y + 1), [\,]) \\
\rightarrow &(let\ x = \#0\ in\ (let\ y = new(0)\ in\ *x := 0\ ;\ *y := *y + 1), [7]) \\
\rightarrow &(let\ y = new(0)\ in\ *(\#0) := 0\ ;\ *y := *y + 1, [7]) \\
\rightarrow &(let\ y = \#1\ in\ *(\#0) := 0\ ;\ *y := *y + 1, [7, 0]) \\
\rightarrow &(*(\#0) := 0\ ;\ *(\#1) := *(\#1) + 1, [7, 0]) \\
\rightarrow &(*(\#1) := *(\#1) + 1, [0, 0]) \\
\rightarrow &(*(\#1) := 0 + 1, [0, 0]) \\
\rightarrow &(*(\#1) := 1, [0, 0]) \\
\rightarrow &((), [0, 1])
\end{aligned}
$$

(c) $let\ x = new(7)\ in\ (let\ y = x\ in\ *x := 0\ ;\ *y := *y + 1)$

$$
\begin{aligned}
&(let\ x = new(7)\ in\ (let\ y = x\ in\ *x := 0\ ;\ *y := *y + 1), [\,]) \\
\rightarrow &(let\ x = \#0\ in\ (let\ y = x\ in\ *x := 0\ ;\ *y := *y + 1), [7]) \\
\rightarrow &(let\ y = \#0\ in\ *(\#0) := 0\ ;\ *y := *y + 1, [7]) \\
\rightarrow &(*(\#0) := 0\ ;\ *(\#0) := *(\#0) + 1, [7]) \\
\rightarrow &(*(\#0) := *(\#0) + 1, [0]) \\
\rightarrow &(*(\#0) := 0 + 1, [0]) \\
\rightarrow &(*(\#0) := 1, [0]) \\
\rightarrow &((), [1])
\end{aligned}
$$

(d) $let\ x = new(4)\ in\ (if\ *x > 0\ then\ *x := 500\ else\ *x := 2)$

$$
\begin{aligned}
&(let\ x = new(4)\ in\ (if\ *x > 0\ then\ *x := 500\ else\ *x := 2), [\,]) \\
\rightarrow &(let\ x = \#0\ in\ (if\ *x > 0\ then\ *x := 500\ else\ *x := 2), [4]) \\
\rightarrow &(if\ *(\#0) > 0\ then\ *(\#0) := 500\ else\ *(\#0) := 2, [4]) \\
\rightarrow &(if\ 4 > 0\ then\ *(\#0) := 500\ else\ *(\#0) := 2, [4]) \\
\rightarrow &(*(\#0) := 500, [4]) \\
\rightarrow &((), [500])
\end{aligned}
$$

(e) $let\ x = new(39)\ in\ (\lambda y.\ *y := *y + 1)\ x$

$$
\begin{aligned}
&(let\ x = new(39)\ in\ (\lambda y.\ *y := *y + 1)\ x, [\,]) \\
\rightarrow &(let\ x = \#0\ in\ (\lambda y.\ *y := *y + 1)\ x, [39]) \\
\rightarrow &((\lambda y.\ *y := *y + 1)\ \#0, [39]) \\
\rightarrow &(*(\#0) := *(\#0) + 1, [39]) \\
\rightarrow &(*(\#0) := 39 + 1, [39]) \\
\rightarrow &(*(\#0) := 40, [39]) \\
\rightarrow &((), [40])
\end{aligned}
$$

(f) $(\lambda y.\ (\lambda x.\ x\ ()\ ;\ x\ ())\ (\lambda z.\ *y := *y + 1))\ new(5)$

$$
\begin{aligned}
&((\lambda y.\ (\lambda x.\ x\ ()\ ;\ x\ ())\ (\lambda z.\ *y := *y + 1))\ new(5), [\,]) \\
\rightarrow&((\lambda y.\ (\lambda x.\ x\ ()\ ;\ x\ ())\ (\lambda z.\ *y := *y + 1))\ \#0, [5]) \\
\rightarrow&((\lambda x.\ x\ ()\ ;\ x\ ())\ (\lambda z.\ *(\#0) := *(\#0) + 1), [5]) \\
\rightarrow&((\lambda z.\ *(\#0) := *(\#0) + 1)\ ()\ ;\ (\lambda z.\ *(\#0) := *(\#0) + 1)\ (), [5]) \\
\rightarrow&((*(\#0) := *(\#0) + 1)\ ;\ (\lambda z.\ *(\#0) := *(\#0) + 1)\ (), [5]) \\
\rightarrow&((*(\#0) := 5 + 1)\ ;\ (\lambda z.\ *(\#0) := *(\#0) + 1)\ (), [5]) \\
\rightarrow&((*(\#0) := 6)\ ;\ (\lambda z.\ *(\#0) := *(\#0) + 1)\ (), [5]) \\
\rightarrow&((\lambda z.\ *(\#0) := *(\#0) + 1)\ (), [6]) \\
\rightarrow&(*(\#0) := *(\#0) + 1, [6]) \\
\rightarrow&(*(\#0) := 6 + 1, [6]) \\
\rightarrow&(*(\#0) := 7, [6]) \\
\rightarrow&((), [7])
\end{aligned}
$$

2.2 Adding deallocation

(a)

$$
\begin{aligned}
&(let\ x = new(5)\ in\ free(x)\ ;\ *x := *x + 1, [\,]) \\
\rightarrow&(let\ x = \#0\ in\ free(x)\ ;\ *x := *x + 1, [5]) \\
\rightarrow&(free(\#0)\ ;\ *(\#0) := *(\#0) + 1, [5]) \\
\rightarrow&(*(\#0) := *(\#0) + 1, [\bot])
\end{aligned}
$$

Now it tries to read a freed location - #0 and hence gets stuck because of violating the tweak in the last rule for writes($s[n] \neq \bot$)

(b)

$$
\begin{aligned}
&(let\ x = new(5)\ in\ free(x)\ ;\ \ free(x), [\,]) \\
\rightarrow&(let\ x = \#0\ in\ free(x)\ ;\ \ free(x), [5]) \\
\rightarrow&(free(\#0)\ ;\ \ free(\#0), [5]) \\
\rightarrow&(free(\#0), [\bot])
\end{aligned}
$$

Now it tries to free the location - #0 twice (was already freed once in the previous step), and hence gets stuck because of violating the second rule for free($s[n] \neq \bot$)