Written Assignment 5

CS 538, Spring 2020

April 3, 2020

1 The Borrow Checker (15)

The Borrow Checker
(a benevolent dictator)
Hey, FYI I'm using this kthx bye
>cargo build
error: cannot borrow `x` as immutable because it is also borrowed as mutable
omg just cloned already!
that's a lie!
<inserts &'a everywhere>
NO! it totally lives long enough!
...10 mins later
...oh the error message told me how to fix it the entire time
Told you so
...1 hour later

— @avadacatavra

These exercises will give you practice with thinking about Rust's ownership rules and the borrow checker. For each of the following code snippets, answer the following questions in 1-2 sentences each:

1. Explain how the program breaks Rust's rules *in your own words*—do not just copy the compiler error.
2. *Why* is the program bad? Suppose that Rust accepted this program. What could go possibly wrong?

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

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

    foo(str_1);
}
```

(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);

    /* ... */
}
```

(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);
}
```

(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);
}
```

(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);
}
```

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

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

    foo(my_str);
}
```

(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);
}
```

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

(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);
}
```

(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);
}
```

2 Lambda Calculus with References (15)

In the most recent homework WR4, we saw how to model a core *imperative* language where programs can read/write to mutable variables in a store. One might think that an imperative language is required to support mutable variables, since the lambda calculus we saw in the first half of the course (WR1, WR2, and WR3) models functional languages without a store, but functional languages can also support mutable variables. In fact, most real-world functional languages (e.g., OCaml, Scheme, Lisp) allow reading or writing variables—Haskell is an exception.

In this part of the homework, we will outline one way to extend the lambda calculus with mutable references and a simple kind of heap. We will start with a basic lambda calculus with integers:

$$\begin{aligned}
n &::= 0 \mid 1 \mid 2 \mid \cdots \\
v &::= x \mid \lambda x.\ e \mid n \mid () \\
e &::= x \mid \lambda x.\ e \mid n \mid () \mid e_1\ e_2 \mid e_1 + e_2 \mid \mathit{if}\ \ e > 0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2
\end{aligned}$$

The special value *unit* () represents a value with no information; we can use this value to encode functions that take no arguments, and functions that don't return anything interesting. We include let-bindings for convenience, even though they can be encoded by function application. To allow stores, we first extend our language with memory locations:

$$\begin{aligned} l &::= \#0 \mid \#1 \mid \#2 \mid \cdots \\ v &::= \cdots \mid l \\ e &::= \cdots \mid l \end{aligned}$$

Locations are integers with a special tag # in front of them; this is to avoid confusing locations with regular integers. Locations are values. Then, we extend expressions with four more constructs:

$$e ::= \cdots \mid *e \mid \mathit{new}(e) \mid *e_1 := e_2 \mid e_1 \ ; e_2$$

The first one reads (dereferences) a location, the second one allocates a new location with initial value e, the third one writes e_2 to a location e_1, and the fourth one runs e_1, then continues with e_2.

To give an operational semantics to this language, we will use the same idea we used for the imperative language: step *pairs* (e, s) of expressions and stores. For this language, our stores s will be finite lists of numbers. For instance, $[\,]$ is the empty store where nothing has been allocated, while $[5, 3]$ is the store where location #0 holds 5 and location #1 holds 3. We will use some notation for working with stores:

- $s :: n$ is the list from appending n to the end of s;
- $\mathit{size}(s)$ is the length of the list s;
- $s[n]$ is the n-th element of the list s ($s[0]$ is the first element);
- $s[n \to v]$ is the same as s, except position n holds v.

We give the step rules for the new constructs first.

Dereference. To dereference, we first step the expression to a location, then look up the location in the store if the address is in bounds:

$$\frac{(e, s) \to (e', s')}{(*e, s) \to (*e', s')} \qquad \frac{l = \#n \qquad n < \mathit{size}(s) \qquad v = s[n]}{(*l, s) \to (v, s)}$$

Allocation. To allocate, we first step the initial expression to a value. Then, we extend the store by slot, and return the new index:

$$\frac{(e, s) \to (e', s')}{(\mathit{new}(e), s) \to (\mathit{new}(e'), s')} \qquad \frac{\mathit{size}(s) = n \qquad s' = s :: v}{(\mathit{new}(v), s) \to (\#n, s')}$$

Write. To write, we first step the left-hand-side to a location, then step the right-hand side to a value, then write the value to the location in the store if it is in bounds:

$$\frac{(e_1, s) \to (e_1', s')}{(*e_1 := e_2, s) \to (*e_1' := e_2, s')} \qquad \frac{(e_2, s) \to (e_2', s')}{(*l := e_2, s) \to (*l := e_2', s')}$$

$$\frac{l = \#n \qquad n < \mathit{size}(s) \qquad s' = s[n \mapsto v]}{(*l := v, s) \to ((), s')}$$

Sequence. The sequence step rules look like the rules for the imperative language. Note that the return value of the first expression is discarded (just like in Rust):

$$\frac{(e_1, s) \to (e_1', s')}{(e_1 \ ; e_2, s) \to (e_1' \ ; e_2, s')} \qquad \frac{}{(v \ ; e_2, s) \to (e_2, s)}$$

The step rules for the lambda calculus features are almost the same as before, except we need to thread the store through the evaluation since evaluating an expression may change the store.

Application. We first evaluate e_1, then e_2, then call the function:

$$\frac{(e_1, s) \to (e_1', s')}{(e_1\ e_2, s) \to (e_1'\ e_2, s')} \qquad \frac{(e_2, s) \to (e_2', s')}{((\lambda x.\ e_1)\ e_2, s) \to ((\lambda x.\ e_1)\ e_2', s')} \qquad \frac{}{((\lambda x.\ e_1)\ v_2, s) \to (e_1[x \mapsto v_2], s)}$$

Addition. We first evaluate e_1, then e_2, then add:

$$\frac{(e_1, s) \to (e_1', s')}{(e_1 + e_2, s) \to (e_1' + e_2, s')} \qquad \frac{(e_2, s) \to (e_2', s')}{(n_1 + e_2, s) \to (n_1 + e_2', s')} \qquad \frac{n = n_1 + n_2}{(n_1 + n_2, s) \to (n, s)}$$

If-then-else. We first evaluate the guard e, then decide which branch to step to:

$$\frac{(e, s) \to (e', s')}{(\textit{if}\ \ e > 0\ \textit{then}\ e_1\ \textit{else}\ e_2, s) \to (\textit{if}\ \ e' > 0\ \textit{then}\ e_1\ \textit{else}\ e_2, s')}$$

$$\frac{n > 0}{(\textit{if}\ \ n > 0\ \textit{then}\ e_1\ \textit{else}\ e_2, s) \to (e_1, s)} \qquad \frac{n = 0}{(\textit{if}\ \ n > 0\ \textit{then}\ e_1\ \textit{else}\ e_2, s) \to (e_2, s)}$$

Let-binding. We first evaluate e_1, then substitute it for x in e_2. This is very similar to function application.

2.1 Stepping expressions

Show how the following programs step, starting from the empty store [ ]. Your answer for each part should be a sequence of the form $(e, [\,]) \to (e_1, s_1) \to \cdots \to (e_n, s_n)$, where e_n cannot step further. We've done the first one for you.

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

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

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

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

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

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

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

2.2 Adding deallocation

The language we have described has a good property: as long as the programmer doesn't put any raw locations (things of the form $\#n$) into the program, there are no memory errors—it is not possible to read or write an unallocated location. Suppose we now try to add a deallocation instruction $\mathit{free}(e)$. We allow locations to contain a special symbol $\bot$, representing deallocated memory. Note that $\bot$ is *not* a value; for instance, the letter v can never refer to $\bot$. Then to deallocate memory, we check the address is allocated, and if so, we set the store to contain $\bot$ at that location:

$$\frac{(e, s) \to (e', s')}{(\mathit{free}(e), s) \to (\mathit{free}(e'), s')} \qquad \frac{l = \#n \qquad n < \mathit{size}(s) \qquad s[n] \neq \bot \qquad s' = s[n \mapsto \bot]}{(\mathit{free}(l), s) \to ((), s')}$$

In this section, we tweak the last rule for writes:

$$\frac{l = \#n \qquad n < \mathit{size}(s) \qquad s[n] \neq \bot \qquad s' = s[n \mapsto v]}{(*l := v, s) \to ((), s')}$$

Unfortunately, our language can now have memory errors.

(a) Write a small program that tries to read a freed location, also known as a *use-after-free*. Show how your program steps, and confirm that it gets stuck—it does not reach a value.

(b) Write a small program that tries to free a location twice, also known as a *double free*. Show how your program steps, and confirm that it gets stuck—it does not reach a value.