Memory Safety in Rust

Nienke Wessel s4598350
Sander Hendrix s4231589

June 2017

Contents

1 Introduction

Rust is a full fledged systems programming language, seeking to provide memory safety without having to sacrifice speed or concurrency. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races.

One of Rust's most distinct and compelling features, is its ownership system. Ownership is how Rust achieves the aforementioned goal of memory safety. We will take a look at two key concepts that make up this ownership system: the idea of ownership, and borrowing.

In this report, we try to formalize the semantics of these two concepts. We do this by taking the standard language **While** and adjusting its semantics to fit our needs. We also analyze small snippets of Rust code to show that the semantic rules we developed are indeed suitable for the language.

1.1 Introduction to Rust

In order to talk about these concepts in terms of syntax and semantics, we have to establish some elementary understanding of the language. In order to keep the scope of this project manageable, we will ignore many of the advanced features. Instead we opt to keep the available features to a level akin to those of **While**.

1.1.1 Variables

Variable bindings are at the core of ownership: they are often the ones owning something or being owned. Bindings are done with the `let`-keyword. By default, bindings are immutable (they cannot be reassigned). This code will not compile:

```
1 let x = 37;
2 x = 42;
```

It will give us `error: re-assignment of immutable variable `x`` instead. If we want a binding to be mutable, we need to explicitly say so by using `mut`. The same example as above, but now x is mutable:

```
1 let mut x = 37;
2 x = 42;
```

1.1.2 Ownership

Variable bindings in Rust 'have ownership' of what they are bound to. Moreover, Rust ensures that there is exactly one binding to any given resource. An example:

```
1 let v = 42;
2 let v2 = v;
3 println!("v is: {}", v);
```

This would be valid code in many other languages. In Rust however, it will result in a ‘‘`error: use of moved value: ` v ``’’. `v2` took ownership of `v`, meaning we can no longer use `v`. When ownership has been transferred, we say that the thing we’re referring to has been ‘moved’.

The reason for this seemingly strict rule has its roots in problems such as data races. For more complex types, such as vectors or strings, memory is both allocated to the stack and to the heap. The stack houses a relatively small pointer to the potential large data on the heap. For assignment like in the example above, new memory on the stack will be allocated in the form of a new pointer that points to the same information on the heap. When multiple variables can access the same data on the heap, things can go awry.

Types like integers that have a known size are stored entirely on the stack, so copies of the actual values are as quick to make as a pointer. That is why integers in reality are optimized: they are actual copies, not pointers to another object. However, to keep the size of our project manageable, we will ignore this optimization and pretend that integers are heap allocated instead (for the Rust aficionados: think of it as Boxed integers). Otherwise, we would have to define syntax and semantics for more complex types, which is not the goal of this report.

1.1.3 Borrowing and references

Most of the time, we would like to access the values of variables without taking ownership over it. To accomplish this, Rust uses a borrowing mechanism. Instead of passing objects by value (`T`), objects can be passed by reference (`&T`). It is common practice to refer to generic types in Rust as `T`.

```
let x = 42;
{
    let y = &x;
}
println!("{}", x);
```

In this nonsensical example `y` borrows `x`. `y` goes out of scope at the closing } brace, returning ownership to `x`. We call the `&T` type a ‘reference’, and rather than *owning* the resource, it *borrows ownership*. Again, by default we cannot alter the resources we are borrowing. When we do want to be able to, we have to use `mut` again, creating a `&mut T` resource. This ‘mutable reference’ allows us to mutate the resource we are borrowing. Note that this allows us to modify the underlying resource that we are borrowing. This means that the underlying has to be mutable as well. Borrowing comes down to references, so there has to be some kind of dereferencing. Rust has both a coercion system and explicit dereferencing. Since this has no influence on the ownership model, we will not consider this. This has the added benefit of keeping the syntax clean and the semantics manageable.

There are some very strict rules around borrowing. The designers of Rust have written a digital book called ‘‘The Rust Programming Language’’[3], and list the rules as the following:

First, any borrow must last for a scope no greater than that of the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:

- one or more references (`&T`) to a resource,
- exactly one mutable reference (`&mut T`).

They then compare this definition to the definition of a data race, and conclude that it rules out a exactly that. This means that as long as we are not writing (`&mut T`), we can read all we want.

An edited example taken from ‘‘The Rust Programming Language’’[3] book, noting that the `println!()` function borrows ownership and returns it when done:

```
let mut x = 42;

let y = &mut x;     // -+ &mut borrow of `x` starts here.
                    //  |
y = y + 1;          //  |
                    //  |
println!("{}", &x);// -+ - Try to borrow `x` here.
                    // -+ &mut borrow of `x` ends here
```

This will not compile: `println!()` tries to borrow `x` whilst `y` still has ownership. Here, problems with, for example, multithreading could occur: we cannot guarantee that, while whatever `println!()` is doing what it is doing, `y` is not modifying `x`. We will not consider multithreading, but it is useful to see why these constructs could be problematic. We can fix all this by simply including a scope for `y`:

```
let mut x = 42;

{
    let y = &mut x; // -+ &mut borrow starts here.
    y = y + 1;      //  |
}                   // -+ ... and ends here.

println!("{}", &x); // <- Try to borrow `x` here.
```

Here, we can be sure that `println!()` can safely read `x` as there is no longer an active mutable borrow of `x`.

2 Syntax

For a description of the syntax, we will use the syntax of **While** as a foundation.

We will use the **While** meta-variable way to talk about constructs of each category. Most of this is the same as in **While**. This results in the following definitions:

n will range over numerals, **Num**,
x will range over variables, **Var**,
a will range over arithmetic expressions, **Aexp**,
b will range over boolean expressions, **Bexp**, and
S will range over statements, **Stm**.

Like in **While**, these meta-variables can be primed or subscripted. We will use the same structure for the numerals and variables as **While**. The structure of the other constructs are as follows:

$$
\begin{array}{lll}
\text{S} & ::= & \texttt{let}\ x = a \mid \texttt{let mut}\ x = a \mid \texttt{let}\ x_1 = \&x_2 \mid \texttt{let mut}\ x_1 = \&x_2 \mid \\
& & \texttt{let}\ x_1 = \texttt{\&mut}\ x_2 \mid \texttt{let mut}\ x_1 = \texttt{\&mut}\ x_2 \mid \\
& & x = a \mid x_1 = \&x_2 \mid x_1 = \texttt{\&mut}\ x_2 \mid S_1; S_2 \mid \{S\} \mid \\
& & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \\
\text{a} & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
\text{b} & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2
\end{array}
$$

Note that the official syntax of Rust is different for while-loops, if-statements and compositions. We chose to use the syntax of **While** to describe these constructs for clarity. Besides, the while-loops and if-statements in Rust use parentheses, which only make things messier in derivation trees.

3 Semantics

For a description of the semantics, we will use the natural semantics. We can use most of the natural semantics defined for the standard language **While** [1] as the basics are similar. Also the natural semantics of **Block** form a starting point from which we can derive our own natural semantics for Rust. We will use this to define scope for Rust.

However, before we can start with defining the natural semantics of Rust, we first need to define some semantic functions. In the next section, we will define and explain these functions. After that, we will describe the natural semantics, starting with the assignments, as they play a major role in memory safety in Rust.

3.1 Semantic functions

In order to keep track of whether a variable is mutable and whether it is being borrowed, we differ from some of the semantics of **While**. In **While**, a state is a function from a variable to a value (in $\mathbb{Z}$). In our description of the semantics, a state is a function from a variable to a four-tuple.

$$\mathbf{State} = \mathbf{Var} \to \mathbf{V} \times \mathbf{bool} \times \mathbf{bool} \times \mathbf{ref}$$

where bool = { true, false } and ref = { none, ref, mut}.

The first element of the tuple, **V**, is the value or an indication that no value was assigned so far or another variable. We first define another set **Z** as

$$\mathbf{Z} = \mathbb{Z} \cup \{\nabla\}$$

where ∇ denotes the absence of a value. Then **V** is defined as

$$\mathbf{V} = \mathbf{Z} \cup \mathbf{Var}$$

The set **Z** will be used to distinguish between actual values and variables later on.

The second element of the tuple is a boolean value to indicate whether the variable is mutable or not, the third is a boolean that indicates whether the variable has been moved (recall: lost ownership) or not, and the last one keeps track whether or not the variable has been borrowed and if so, how (not at all, as reference (`&T`), or as mutable reference `&mut T`). The initial state of every program is the state that assigns (∇, false, false, none) to every variable.

Since we are now working with tuples, we will define some functions to easily access the elements of the tuples:

$$\begin{aligned} \text{value} &: (a, b, c, d) \to a \\ \text{mut} &: (a, b, c, d) \to b \\ \text{moved} &: (a, b, c, d) \to c \\ \text{borrow} &: (a, b, c, d) \to d \end{aligned}$$

These functions are defined in the following way:

$$\begin{aligned}\text{value}((x_1, x_2, x_3, x_4)) &= x_1\\ \text{mut}((x_1, x_2, x_3, x_4)) &= x_2\\ \text{moved}((x_1, x_2, x_3, x_4)) &= x_3\\ \text{borrow}((x_1, x_2, x_3, x_4)) &= x_4\end{aligned}$$

Since we have changed the states, we also need to change the definition of $\mathcal{A}$, as that function assumes the state is a function to a value. The functionality of $\mathcal{A}$ stays

$$\mathcal{A} : \mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$$

but the semantics of the arithmetic expressions change.

There are two major differences between our semantics of $\mathcal{A}[\![x]\!]$ and those in the language **While**. For starters, we need to redefine the semantics for $\mathcal{A}[\![x]\!]s$. Moreover, we need to take the ∇ into account.

$$\begin{aligned}\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!]\\ \mathcal{A}[\![x]\!]s &= \begin{cases}\text{value}(s\ x) & \text{if value}(s\ x) \in \mathbf{Z}\\ \mathcal{A}[\![\text{value}(s\ x)]\!]s & \text{otherwise}\end{cases}\\ \mathcal{A}[\![a_1 + a_2]\!]s &= \begin{cases}\nabla & \text{if } \mathcal{A}[\![a_1]\!]s = \nabla \text{ or } \mathcal{A}[\![a_2]\!]s = \nabla\\ \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s & \text{otherwise}\end{cases}\\ \mathcal{A}[\![a_1 * a_2]\!]s &= \begin{cases}\nabla & \text{if } \mathcal{A}[\![a_1]\!]s = \nabla \text{ or } \mathcal{A}[\![a_2]\!]s = \nabla\\ \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s & \text{otherwise}\end{cases}\\ \mathcal{A}[\![a_1 - a_2]\!]s &= \begin{cases}\nabla & \text{if } \mathcal{A}[\![a_1]\!]s = \nabla \text{ or } \mathcal{A}[\![a_2]\!]s = \nabla\\ \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s & \text{otherwise}\end{cases}\end{aligned}$$

The definition for $\mathcal{A}[\![x]\!]s$ has two cases, depending on whether the result is in $\mathbf{Z}$ or **Var**. When it is in **Var**, we need to recursively use the function.

We illustrate this with an example borrowed from Nielson & Nielson, where

$$s\ x = (3, \text{true}, \text{false}, \text{none})$$

$$\begin{aligned}\mathcal{A}[\![x + 1]\!]s &= \mathcal{A}[\![x]\!]s + \mathcal{A}[\![1]\!]s\\ &= \text{value}(s\ x) + \mathcal{N}[\![1]\!]\\ &= \text{value}(3, \text{true}, \text{false}, \text{none}) + 1\\ &= 3 + 1\\ &= 4\end{aligned}$$

Similarly, in the case $s\ x = (\nabla, \text{true}, \text{false}, \text{none})$:

$$\mathcal{A}[\![x + 1]\!]s = \nabla$$

since $\mathcal{A}[\![x]\!]s = \nabla$.

We will change the boolean expressions accordingly.

$$
\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]s &= \texttt{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s &= \texttt{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \nabla & \text{if } \mathcal{A}[\![a_1]\!]s = \nabla \text{ or } \mathcal{A}[\![a_2]\!]s = \nabla \\ \texttt{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \texttt{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \nabla & \text{if } \mathcal{A}[\![a_1]\!]s = \nabla \text{ or } \mathcal{A}[\![a_2]\!]s = \nabla \\ \texttt{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \texttt{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg b]\!]s &= \begin{cases} \nabla & \text{if } \mathcal{B}[\![b]\!]s = \nabla \\ \texttt{tt} & \text{if } \mathcal{B}[\![b]\!]s = \texttt{ff} \\ \texttt{ff} & \text{if } \mathcal{B}[\![b]\!]s = \texttt{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \nabla & \text{if } \mathcal{B}[\![b_1]\!]s = \nabla \text{ or } \mathcal{B}[\![b_2]\!]s = \nabla \\ \texttt{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \texttt{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \texttt{tt} \\ \texttt{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \texttt{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \texttt{ff} \end{cases}
\end{aligned}
$$

3.1.1 Free variables

There is another function that we will use quite often, the function **FV**. This function is explained in Nielson & Nielson, but since it was not part of the material of the Semantics and Correctness course, we will briefly explain it here.

The function is a mapping from an expression to a subset of **Var**. The function is defined in the following way:

$$
\begin{aligned}
\mathbf{FV}(n) &= \emptyset \\
\mathbf{FV}(x) &= \{x\} \\
\mathbf{FV}(a_1 + a_2) &= \mathbf{FV}(a_1) \cup \mathbf{FV}(a_2) \\
\mathbf{FV}(a_1 * a_2) &= \mathbf{FV}(a_1) \cup \mathbf{FV}(a_2) \\
\mathbf{FV}(a_1 - a_2) &= \mathbf{FV}(a_1) \cup \mathbf{FV}(a_2)
\end{aligned}
$$

In very much the same way, this is defined for boolean expressions:

$$
\begin{aligned}
\mathbf{FV}(\texttt{true}) &= \emptyset \\
\mathbf{FV}(\texttt{false}) &= \emptyset \\
\mathbf{FV}(a_1 = a_2) &= \mathbf{FV}(a_1) \cup \mathbf{FV}(a_2) \\
\mathbf{FV}(a_1 \leq a_2) &= \mathbf{FV}(a_1) \cup \mathbf{FV}(a_2) \\
\mathbf{FV}(\neg b) &= \mathbf{FV}(b) \\
\mathbf{FV}(b_1 \wedge b_2) &= \mathbf{FV}(b_1) \cup \mathbf{FV}(b_2)
\end{aligned}
$$

For more information on the function and the proofs of several properties of it, we refer to chapter 1 of the book of Nielson & Nielson.

3.2 Error handling

Since the main focus of our project is on memory safety and the things you can and cannot do in Rust, we also need a way to handle programs that do no adhere to the rules. That is why we introduce the error status, which works in a similar way as the break status that was introduced in the first lecture of the computation models course [2]. We use the symbol $\bigodot$ to indicate that no error has occurred and the symbol $\bigotimes$ to indicate that an error did occur. That means our transitions will be of the form

$$\langle S, s\rangle \to (s', e)$$

where $e \in E = \{\bigodot, \bigotimes\}$. As will become clear when defining the semantics, we have chosen to let a program end when an error has occured. That means subsequent statements will no longer be performed.

3.3 Natural semantics

Now we have sufficient semantic functions and a way to keep track of errors, we can start to define the natural semantics for the part of Rust we are investigating. Since most of the syntax we cover is about assignments, this naturally becomes the most interesting part of the semantics and also the part that differs most from **While**. We will need to make use of the functions defined above to make the description.

3.3.1 Assignments

We will first consider the assignments, as these are the most interesting. There are a lot of different rules, as we need two rules for each syntactic element: one for the case the assignment is correct and one where it is not. A big difference from **While** is that variables must first be declared via a `let-` or `let mut` statement. In the latter case, the value can be altered with a statement that closer resembles the original **While** assignment rule.

$$[\text{let}^{\bigodot}_{\text{ns}}] \quad \langle \texttt{let } x = a, s\rangle \to (s \left[\begin{array}{ll} x & \mapsto (\mathcal{A}[\![a]\!]s, \text{false}, \text{false}, \text{none}) \\ \forall v \in \text{FV}(a) & \mapsto (\text{value}(s\ v), \text{mut}(s\ v), \text{true}, \text{borrow}(s\ v)) \end{array}\right], \bigodot)$$
$$\text{if value}(s\ x) = \nabla$$
$$\text{and } \neg\exists v \in \text{FV}(a) : \text{moved}(s\ v) = \text{true}$$

$$[\text{let}^{\bigotimes}_{\text{ns}}] \quad \langle \texttt{let } x = a, s\rangle \to (s, \bigotimes)$$
$$\text{if value}(s\ x) \neq \nabla$$
$$\text{or } \exists v \in \text{FV}(a) : \text{moved}(s\ v) = \text{true}$$

$$[\text{letm}^{\bigodot}_{\text{ns}}] \quad \langle \texttt{let mut } x = a, s\rangle \to s \left[\begin{array}{ll} x & \mapsto (\mathcal{A}[\![a]\!]s, \text{true}, \text{false}, \text{none}) \\ \forall v \in \text{FV}(a) & \mapsto (\text{value}(s\ v), \text{mut}(s\ v), \text{true}, \text{borrow}(s\ v)) \end{array}\right], \bigodot)$$
$$\text{if value}(s\ x) = \nabla$$
$$\text{and } \exists v \in \text{FV}(a) : \text{moved}(s\ v) = \text{true}$$

$$[\text{letm}^{\bigotimes}_{\text{ns}}] \quad \langle \texttt{let mut } x = a, s\rangle \to (s, \bigotimes)$$
$$\text{if value}(s\ x) \neq \nabla$$
$$\text{or } \neg\exists v \in \text{FV}(a) : \text{moved}(s\ v) = \text{true}$$

The $[\text{let}_{\text{ns}}]$ rules are the immutable variable declarations, and the $[\text{letm}_{\text{ns}}]$ are the mutable ones. Their only difference is the second value in the four-tuple:

false for immutable and true for mutable. Mutable variables can be reassigned using the [ass_{ns}] rules.

We cannot use moved variables to declare a new variable, that is what the check handles. In return, declaring a variable using other variables leads to the latter ones being moved.

$$
\begin{aligned}
[\text{let\_ref}^{\odot}_{\text{ns}}] \quad & \langle \texttt{let } x_1 = \&x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{false}, \text{false}, \text{ref})], \bigodot) \\
& \qquad \text{if value}(s\ x_1) = \nabla \\
& \qquad \text{and moved}(s\ x_2) = \text{false} \\
& \qquad \text{and } (\neg\exists y \in \mathbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut})) \\
[\text{let\_ref}^{\bigotimes}_{\text{ns}}] \quad & \langle \texttt{let } x_1 = \&x_2, s\rangle \to (s, \bigotimes) \\
& \qquad \text{if value}(s\ x_1) \neq \nabla \\
& \qquad \text{or moved}(s\ x_2) = \text{true} \\
& \qquad \text{or } (\exists y \in \mathbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut})) \\
[\text{letm\_ref}^{\odot}_{\text{ns}}] \quad & \langle \texttt{let mut } x_1 = \&x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{true}, \text{false}, \text{none})], \bigodot) \\
& \qquad \text{if value}(s\ x_1) = \nabla \\
& \qquad \text{and moved}(s\ x_2) = \text{false} \\
& \qquad \text{and } (\neg\exists y \in \mathbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut})) \\
[\text{letm\_ref}^{\bigotimes}_{\text{ns}}] \quad & \langle \texttt{let mut } x_1 = \&x_2, s\rangle \to (s, \bigotimes) \\
& \qquad \text{if value}(s\ x_1) \neq \nabla \\
& \qquad \text{or moved}(s\ x_2) = \text{true} \\
& \qquad \text{or } (\exists y \in \mathbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut}))
\end{aligned}
$$

Borrowing a reference differs from normal `let` assignment. We still have to check whether or not we are using moved variables: we cannot do anything with those, so we cannot borrow them either. Where it differs has to do with possible previous borrows: we cannot take a borrow as immutable reference (`&T`) if there already is another borrow as mutable reference (&mut T).

The difference between the [$\text{let\_ref}_{\text{ns}}$] and [$\text{letm\_ref}_{\text{ns}}$] rules is much the same however: mutable references can be reassigned so they reference to another variable using one of the [$\text{ass\_ref}_{\text{ns}}$] rules outlined below.

$$[\text{let\_refm}^{\odot}_{\text{ns}}] \quad \langle \texttt{let } x_1 = \&\texttt{mut } x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{false}, \text{false}, \text{none})], \bigodot)$$
$$\text{if value}(s\ x_1) = \nabla$$
$$\text{and mut}(s\ x_2) = \text{true}$$
$$\text{and moved}(s\ x_2) = \text{false}$$
$$\text{and } (\neg\exists y \in \mathbf{Var} : \text{value}(s\ y) = x_2)$$

$$[\text{let\_refm}^{\otimes}_{\text{ns}}] \quad \langle \texttt{let } x_1 = \&\texttt{mut } x_2, s\rangle \to (s, \bigotimes)$$
$$\text{if value}(s\ x_1) \neq \nabla$$
$$\text{or mut}(s\ x_2) = \text{false}$$
$$\text{or moved}(s\ x_2) = \text{true}$$
$$\text{or } (\exists y \in \mathbf{Var} : \text{value}(s\ y) = x_2)$$

$$[\text{letm\_refm}^{\odot}_{\text{ns}}] \quad \langle \texttt{let mut } x_1 = \&\texttt{mut } x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{true}, \text{false}, \text{none})], \bigodot)$$
$$\text{if value}(s\ x_1) = \nabla$$
$$\text{and mut}(s\ x_2) = \text{true}$$
$$\text{and moved}(s\ x_2) = \text{false}$$
$$\text{and } (\neg\exists y \in \mathbf{Var} : \text{value}(s\ y) = x_2)$$

$$[\text{letm\_refm}^{\otimes}_{\text{ns}}] \quad \langle \texttt{let mut } x_1 = \&\texttt{mut } x_2, s\rangle \to (s, \bigotimes)$$
$$\text{if value}(s\ x_1) \neq \nabla$$
$$\text{or mut}(s\ x_2) = \text{false}$$
$$\text{or moved}(s\ x_2) = \text{true}$$
$$\text{or } (\exists y \in \mathbf{Var} : \text{value}(s\ y) = x_2)$$

The [let_refm$_{\text{ns}}$] and [letm_refm$_{\text{ns}}$] rules are much like the [let_ref$_{\text{ns}}$] and [letm_ref$_{\text{ns}}$] rules, but now we make sure that there are no other references (not `&T` nor `&mut T`).

In addition, to take a `&mut T` the variable that we are borrowing also has to be mutable. Again, we can still not make use of moved variables.

Again, the only difference between the 'let' and the 'letm' rules are their own mutability. The mutable rules are able to be reassigned using one of the [ass_refm$_{\text{ns}}$] rules below.

$$[\text{ass}^{\odot}_{\text{ns}}] \quad \langle x = a, s\rangle \to (s \left[\begin{array}{ll} x & \mapsto (\mathcal{A}[\![a]\!]s, \text{true}, \text{false}, \text{none}) \\ \forall y \in \text{FV}(a) & \mapsto (\text{value}(s\ y), \text{mut}(s\ y), \text{true}, \text{borrow}(s\ y)) \end{array}\right], \bigodot)$$
$$\text{if mut}(s\ x) = \text{true and}$$
$$\neg\exists v \in \text{FV}(a)\text{: moved}(v) = \text{true}$$
$$\text{and } (\neg\exists y \in \mathbf{Var} : \text{value}(s\ y) = x)$$

$$[\text{ass}^{\otimes}_{\text{ns}}] \quad \langle x = a, s\rangle \to (s, \bigotimes)$$
$$\text{if mut}(s\ x) = \text{false or}$$
$$\exists v \in \text{FV}(a)\text{: moved}(v) = \text{true}$$
$$\text{or } (\exists y \in \mathbf{Var} : \text{value}(s\ y) = x)$$

Reassignment of variables is only possible if the variable is mutable, if we are using variables they cannot be moved, and there is no borrow. If we would not check if x was borrowed, we would either be able to write whilst also reading, or write via multiple ways. Both are strictly forbidden.

$$[\text{ass\_ref}^{\odot}_{\text{ns}}] \qquad \langle x_1 = \&x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{false}, \text{false}, \text{ref})], \bigodot)$$
$$\text{if mut}(s\ x_1) = \text{true}$$
$$\text{and moved}(s\ x_2) = \text{false}$$
$$\text{and } (\neg\exists y \in \textbf{Var} : \text{value}(s\ y) = x_1)$$
$$\text{and } (\neg\exists y \in \textbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut}))$$

$$[\text{ass\_ref}^{\bigotimes}_{\text{ns}}] \qquad \langle x_1 = \&x_2, s\rangle \to (s, \bigotimes)$$
$$\text{if mut}(s\ x_1) = \text{false or}$$
$$\text{or moved}(s\ x_2) = \text{true}$$
$$\text{or } (\exists y \in \textbf{Var} : \text{value}(s\ y) = x_1)$$
$$\text{or } (\exists y \in \textbf{Var} : (\text{value}(s\ y) = x_2 \wedge \text{borrow}(s\ y) = \text{mut}))$$

Reassignment of references is relatively easy, but comes with some checks. The variable being changed, x_1, has to be mutable and may not be borrowed: like above, we cannot change a variable that is being borrowed.

The variable being borrowed, x_2, may not be moved as always, and may not have an active borrow as mutable reference: remember that we can have multiple immutable reference or one mutable reference.

$$[\text{ass\_refm}^{\odot}_{\text{ns}}] \quad \langle x_1 = \&\texttt{mut}\ \ x_2, s\rangle \to (s[x_1 \mapsto (x_2, \text{true}, \text{false}, \text{none})], \bigodot)$$
$$\text{if mut}(s\ x_1) = \text{true}$$
$$\text{and moved}(s\ x_2) = \text{false}$$
$$\text{and } (\neg\exists y : \text{value}(s\ y) = x_1)$$
$$\text{and } (\neg\exists y \in \textbf{Var} : \text{value}(s\ y) = x_2)$$

$$[\text{ass\_refm}^{\bigotimes}_{\text{ns}}] \qquad \langle x_1 = \&\texttt{mut}\ \ x_2, s\rangle \to (s, \bigotimes)$$
$$\text{if mut}(s\ x_1) = \text{false}$$
$$\text{or moved}(s\ x_2) = \text{true}$$
$$\text{or } (\exists y \in \textbf{Var} : \text{value}(s\ y) = x_1)$$
$$\text{or } (\exists y \in \textbf{Var} : \text{value}(s\ y) = x_2)$$

Reassignment of mutable references, much like immutable references, requires x_1 to be mutable and not borrowed. Again x_2 may not be moved, but in this case there may be no borrow of x_2 whatsoever.

3.3.2 Compositions

As per the lecture notes [2], although after an error occurs we stop executing, as opposed to skipping, rendering the rest of the composition void. As in the notes, we use a generic metavariable e instead of formulating two separate rules for when S_1 does not result in an error.

$$[\text{comp}^{\odot}_{\text{ns}}] \quad \frac{\langle S_1, s\rangle \to (s', \bigodot) \qquad \langle S_2, s'\rangle \to (s'', e)}{\langle S_1;\ S_2, s\rangle \to (s'', e)}$$

$$[\text{comp}^{\bigotimes}_{\text{ns}}] \quad \frac{\langle S_1, s\rangle \to (s', \bigotimes)}{\langle S_1;\ S_2, s\rangle \to (s', \bigotimes)}$$

3.3.3 Conditionals

The point of the third value in the tuple is to mark whether the value has been moved (ownership has been transferred to another variable). Moved variables can no longer be used. In our case, this has consequences for both the `If` and `While`: they can not use moved variables in their booleans.

The main difference compared to **While** is that both gain the $\bigotimes$-rule, and the existing rules are expanded to make sure that none of the variables in the boolean clause has been moved.

$$[\text{if}^{\texttt{tt}}_{\text{ns}}\bigodot] \quad \frac{\langle S_1, s\rangle \to (s', e)}{\langle\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', e)} \quad \text{if } (\neg\exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}) \text{ and } \mathcal{B}[\![b]\!]s = \texttt{tt}$$

$$[\text{if}^{\texttt{ff}}_{\text{ns}}\bigodot] \quad \frac{\langle S_2, s\rangle \to (s', e)}{\langle\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', e)} \quad \text{if } (\neg\exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}) \text{ and } \mathcal{B}[\![b]\!]s = \texttt{ff}$$

$$[\text{if}_{\text{ns}}\bigotimes] \quad \langle\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s, \bigotimes) \quad \text{if } \exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}$$

$$[\text{while}^{\texttt{tt}}_{\text{ns}}\bigodot] \quad \frac{\langle S, s\rangle \to (s', \bigodot) \qquad \langle\ \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to (s'', e)}{\langle\ \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s'', e)} \quad \text{if } (\neg\exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}) \text{ and } \mathcal{B}[\![b]\!]s = \texttt{tt}$$

$$[\text{while}^{\texttt{ff}}_{\text{ns}}\bigodot] \quad \langle\ \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s, \bigodot) \quad \text{if } (\neg\exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}) \text{ and } \mathcal{B}[\![b]\!]s = \texttt{ff}$$

$$[\text{while}_{\text{ns}}\bigotimes] \quad \langle\ \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s, \bigotimes) \quad \text{if } \exists v \in \text{FV}(b) : \text{moved}(v) = \texttt{true}$$

3.3.4 Blocks

Blocks are denoted by curly braces. Variables declared within a block are local to that block. Blocks and local variables are useful because these local variables cease to exist after the block. This is especially useful for returning a reference: recall that we can borrow either many immutable references, or exactly one mutable reference. We would be limited to either one if there was no way to drop a reference. With blocks we can, enabling the use of one after the other.

For example, say we borrow a mutable reference in a block. We are free to borrow multiple immutable references after the block, as the mutable reference does not exist anymore; the block ended.

The scoped variables of a block $\{S\}$ is defined to be the set of variables declared in it. Formally we give a compositional definition of the subset $\text{SV}(\{S\})$ of **Var**:

$$\begin{array}{lcl}
\text{SV}(\texttt{let}\ x = a) & = & \{\ x\ \} \\
\text{SV}(\texttt{let mut}\ x = a) & = & \{\ x\ \} \\
\text{SV}(\texttt{let}\ x_1 = \&x_2) & = & \{\ x_1\ \} \\
\text{SV}(\texttt{let mut}\ x_1 = \&x_2) & = & \{\ x_1\ \} \\
\text{SV}(\texttt{let}\ x_1 = \texttt{\&mut}\ x_2) & = & \{\ x_1\ \} \\
\text{SV}(\texttt{let mut}\ x_1 = \texttt{\&mut}\ x_2) & = & \{\ x_1\ \} \\
\text{SV}(x = a) & = & \emptyset \\
\text{SV}(x_1 = \&x_2) & = & \emptyset \\
\text{SV}(x_1 = \texttt{\&mut}\ x_2) & = & \emptyset \\
\text{SV}(S_1; S_2) & = & \text{SV}(S_1) \cup \text{SV}(S_2) \\
\text{SV}(\{\ S\ \}) & = & \text{SV}(S) \\
\text{SV}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) & = & \text{SV}(S_1) \cup \text{SV}(S_2) \\
\text{SV}(\texttt{while}\ b\ \texttt{do}\ S) & = & \text{SV}(S)
\end{array}$$

As an example $\text{SV}(\{\ \texttt{let mut}\ x1 = 42; \texttt{let mut}\ x2 = \&x1\ \}) = \{\ x1, x2\ \}$

$$[\text{block}_{\text{ns}}] \quad \frac{\langle S, s\rangle \to (s', e)}{\langle\{S\}, s\rangle \to (s'[s' \longmapsto (s\ \{S\})], e)}$$

We write $s'[s' \longmapsto (s\ \{S\})]$ for the state that is as s', except:

- all variables local to the block $\{\ S\ \}$ are returned to their initial $(\nabla, \text{false}, \text{false}, \text{none})$ state;
- variables not local to the block that are borrowing a local variable are invalidated

Formally,

$$(s'[s' \longmapsto (s\ \{S\})])x = \begin{cases} (\nabla, \text{false}, \text{false}, \text{none}) & \text{if } x \in \text{SV}(\{S\}) \\ (\nabla, \text{mut}(s'x), \text{moved}(s'x), \text{none}) & \text{if } \exists y \in \text{SV}(\{S\}) : \text{value}(s\ x) = y \end{cases}$$

4 Analysis

We will take a look at some interesting examples.

4.1 Mutable and immutable

We will first analyze a program that results in an error, as x is not mutable. The following program P is the same as in the introduction, but with some extra statements to show how the error handling works.

```
1 let x = 37;
2 let y = 420;
3 x = 42;
4 let z = 10
```

This gives the following derivation tree

$$\dfrac{[\text{let}^{\odot}_{\text{ns}}]\ [\text{comp}^{\odot}_{\text{ns}}]\ \dfrac{\langle \text{ let } x = 37, s_0\rangle \to (s_1, \odot) \qquad [\text{let}^{\odot}_{\text{ns}}]\ [\text{comp}^{\odot}_{\text{ns}}]\ \dfrac{\langle \text{ let } y = 420;, s_1\rangle \to (s_e, \odot) \qquad [\text{ass}^{\otimes}_{\text{ns}}]\ [\text{comp}^{\otimes}_{\text{ns}}]\ \dfrac{\langle x = 42, s_e\rangle \to (s_e, \otimes)}{\langle R_2, s_e\rangle \to (s_e, \otimes)}}{\langle R_1, s_1\rangle \to (s_e, \otimes)}}{\langle P, s_0\rangle \to (s_e, \otimes)}$$

where

R_1 = `let` y = 420; x = 42; let z = 10
R_2 = x = 42; let z = 10
s_e: $x \mapsto$ (37, false, false, none), $y \mapsto$ (420, false, false, none), all others $\mapsto$ (∇, false, false, none)
s_0: all variables $\mapsto$ (∇, false, false, none)
s_1: $x \mapsto$ (37, false, false, none), all others $\mapsto$ (∇, false, false, none)

As one can see, the statement `let z = 10` is never reached. The program stops executing statements after an error has occured.

We now show that the adjusted program where `mut` is used results in no error. This time without the extra statements, as there is no error handling.

```
1 let mut x = 37;
2 x = 42
```

$$\dfrac{[\text{letm}^{\odot}_{\text{ns}}]\ [\text{comp}^{\odot}_{\text{ns}}]\ \langle \text{let mut x} = 37,\ s_0\rangle, \to (s_1, \odot) \qquad [\text{ass}^{\odot}_{\text{ns}}]\ \langle \text{x} = 42,\ s_1\rangle, \to (s_e, \odot)}{\langle \text{let mut x} = 37; \text{ x} = 42,\ s_0\rangle, \to (s_e, \odot)}$$

where

s_e: $x \mapsto$ (42, true, false, none), all others $\mapsto$ (∇, false, false, none)
s_0: all variables $\mapsto$ (∇, false, false, none)
s_1: $x \mapsto$ (37, true, false, none), all others $\mapsto$ (∇, false, false, none)

4.2 Move

Recall: we call a variable moved if that variable has been assigned to another variable. Moved variables can no longer be used.

```
let b = true;
while b is true do (
    let a = b
)
```

Which gives the following tree:

$$\dfrac{[\text{let}^{\odot}_{\text{ns}}]\ \dfrac{}{\langle \text{let b = true}, s_0\rangle, \to (s_1, \odot)} \quad [\text{comp}^{\odot}_{\text{ns}}]\ \dfrac{[\text{let}^{\odot}_{\text{ns}}]\ \dfrac{}{\langle \text{let a = b}, s_1\rangle, \to (s_2, \odot)} \quad [\text{while}^{\text{tt}}_{\text{ns}}\ \odot]\ \dfrac{[\text{while}^{\text{tt}}_{\text{ns}}\ \otimes]\ \dfrac{}{\langle \text{while b is true do (let a = b)}, s_2\rangle, \to (s_2, \otimes)}}{\langle \text{while b is true do (let a = b)}, s_1\rangle, \to (s_2, \otimes)}}{\langle \text{ let b = true; while b is true do (let a = b)}, s_0\rangle, \to (s_2, \otimes)}$$

where

s_0: $\{a, b\} \mapsto$ (∇, false, false, none)
s_1: $b \mapsto$ (true, false, false, none), $a \mapsto$ (∇, false, false, none)
s_2: $b \mapsto$ (true, false, true, none), $a \mapsto$ (true, false, false, none)

This results in an error via $[\text{while}_{\text{ns}}\ \otimes]$, because moved($b$) = `true`.

4.3 References and mutable references

Recall the rules around borrowing: there can either be one or more immutable references, or one mutable reference. A simple way to break that rule is to take both. Lets try to take a mutable reference while we already have multiple immutable references, and see if it does indeed result in an error:

```
let mut x = 42;
let y = &x;
let z = &x;
let e = &mut x
```

Which gives the following tree:

$$\dfrac{[\text{letm}^{\odot}_{\text{ns}}]\ \dfrac{}{\langle \text{let mut x = 42}, s_0\rangle, \to (s_1, \odot)} \quad [\text{comp}^{\odot}_{\text{ns}}]\ \dfrac{[\text{let\_ref}^{\odot}_{\text{ns}}]\ \dfrac{}{\langle \text{let y = \&x}, s_1\rangle, \to (s_2, \odot)} \quad [\text{comp}^{\odot}_{\text{ns}}]\ \dfrac{[\text{let\_ref}^{\odot}_{\text{ns}}]\ \dfrac{}{\langle \text{let z = \&x}, s_2\rangle, \to (s_3, \odot)} \quad [\text{let\_refm}^{\otimes}_{\text{ns}}]\ \dfrac{}{\langle \text{let e = \&mut x}, s_e\rangle, \to (s_3, \otimes)}}{\langle \text{let z = \&x; let e = \&mut x}, s_2\rangle, \to (s_3, \otimes)}}{\langle \text{let y = \&x; let z = \&x; let e = \&mut x}, s_1\rangle, \to (s_3, \otimes)}}{\langle \text{ let mut x = 42; let y = \&x; let z = \&x; let e = \&mut x}, s_0\rangle, \to (s_3, \otimes)}$$

where

s_0: all variables $\mapsto$ (∇, false, false, none)
s_1: $x \mapsto$ (42, true, false, none), $\{y, z, e\} \mapsto$ (∇, false, false, none)
s_2: $x \mapsto$ (42, true, false, ref), $y \mapsto$ (x, false, false, none), $\{z, e\} \mapsto$ (∇, false, false, none)
s_3: $x \mapsto$ (42, true, false, ref), $y \mapsto$ (x, false, false, none), $z \mapsto$ (x, false, false, none), e $\mapsto$ (∇, false, false, none)

We see here that `let` e = `&mut` x results in a $\bigotimes$ via [let_refm$^{\bigotimes}_{\text{ns}}$], since there exists a variable v where value(s v) = x. Namely: both y and z.

Meaning we have already borrowed x as a reference (`&T`), so we can no longer take a mutable reference (`&mut T`).

4.4 Block

We can fix the previous piece of code by including a block for y and z:

```
1  let mut x = 42;
2  {
3      let y = &x;
4      let z = &x
5  };
6  let e = &mut x
```

Which gives the following tree:

$$\dfrac{\dfrac{}{\langle \text{let mut x = 42; } s_0\rangle, \to (s_1, \odot)}\,[\text{letm}^{\odot}_{\text{ns}}]\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\langle \text{let y = \&x, } s_1\rangle, \to (s_2, \odot)}\,[\text{let\_ref}^{\odot}_{\text{ns}}]\quad \dfrac{}{\langle \text{let z = \&x, } s_2\rangle, \to (s_3, \odot)}\,[\text{let\_ref}^{\odot}_{\text{ns}}]}{\langle \text{let y = \&x; let z = \&x, } s_1\rangle, \to (s_3, \odot)}\,[\text{block}_{\text{ns}}]}{\langle \text{ \{ let y = \&x; let z = \&x \}, } s_1\rangle, \to (s_4, \odot)}\,[\text{comp}^{\odot}_{\text{ns}}]\quad \dfrac{}{\langle \text{let e = \&mut x, } s_4\rangle, \to (s_5, \odot)}\,[\text{let\_refm}^{\odot}_{\text{ns}}]}{\langle \text{ \{ let y = \&x; let z = \&x \}; let e = \&mut x, } s_1\rangle, \to (s_5, \odot)}\,[\text{comp}^{\odot}_{\text{ns}}]}{\langle \text{ let mut x = 42; \{ let y = \&x; let z = \&x \}; let e = \&mut x, } s_0\rangle, \to (s_5, \odot)}\,[\text{comp}^{\odot}_{\text{ns}}]$$

where

s_0: all variables $\mapsto$ (∇, false, false, none)
s_1: $x \mapsto$ (42, true, false, none), $\{y, z, e\} \mapsto$ (∇, false, false, none)
s_2: $x \mapsto$ (42, true, false, ref), $y \mapsto$ (x, false, false, none), $\{z, e\} \mapsto$ (∇, false, false, none)
s_3: $x \mapsto$ (42, true, false, ref), $y \mapsto$ (x, false, false, none), $z \mapsto$ (x, false, false, none), e $\mapsto$ (∇, false, false, none)
s_4: $x \mapsto$ (42, true, false, none), $\{y, z\} \mapsto$ (∇, false, false, none), e $\mapsto$ (∇, false, false, none)
s_5: $x \mapsto$ (42, true, false, mut), $\{y, z\} \mapsto$ (∇, false, false, none), e $\mapsto$ (42, false, false, none)

We see here that `let` e = `&mut` x no longer fails. At the end of the block both y and z are invalidated and x is restored to its original borrow status. In other words: borrow(s_4 x) = none, instead of ref. We have no borrow of x, so now we *can* take a mutable reference (`&mut T`).

References

[1] Nielson, H.R., F. Nielson (1992). *Semantics with Applications*, Wiley, Chichester. Revised edition (1999), available at `www.daimi.au.dk/~hrn`, retrieved from `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf`

[2] Barendsen, E. (2010) *Semantics of abrupt completion: a case study in natural semantics*, lecture notes for *Semantiek en Correctheid*, Radboud University Nijmegen, retrieved from `https://www.cs.ru.nl/~hubbers/courses/bm/`

[3] Various, *The Rust Programming Language* [Digital book, first edition], Retrieved from `https://doc.rust-lang.org/book/`