Chapter 1

Borrowing

1.1 Introduction

In the previous chapter, we looked quite extensively at a very simple language. In this chapter, we will add some of the more interesting features, namely borrowing and mutability. Borrowing presents us with new challenges, as we need to make sure that the variable we borrow from is not accessible during the borrow, but becomes accessible again after the borrow. We will need to keep track of what variables we have borrowed from. If we do not do so, we cannot free them afterwards.

1.2 Syntax

First, we need to update our syntax to reflect the possibility of borrowing. The possible statements from definition **??** can remain almost the same. We just need to make a new statement for a mutable let. We will also have to update our expressions and types.

Definition 1.2.1. A statement S is defined recursively by:

$$S ::= \texttt{skip} \mid S_1; S_2 \mid \texttt{let}\ x : \tau\ \texttt{in}\ S' \mid \texttt{let mut}\ x : \tau\ \texttt{in}\ S' \mid x = e$$

where e is an expression as defined below, τ is a type as defined below and S_1, S_2 and S' are again statements.

Our definition of a program instruction (Definition **??**) updates accordingly with this new definition of a statement. Again, we will be using S to refer to both, but will make explicit what we are talking about in another way.

Definition 1.2.2. An expression e is defined recursively by:

$$e ::= x \mid i \mid e_1 + e_2 \mid \&\ x \mid \&\texttt{mut}\ x$$

Definition 1.2.3. A type τ is

$$\tau ::= \text{Int} \mid \&\tau$$

1.3 Examples

In order to illustrate what is happening in this chapter and therefore what should be happening in the semantics, we will walk through some small pieces of example code.

1.3.1 Mutability

To start of easy, we will first look at an example of mutability without borrowing.

```
1 let mut y = 0;
2 y = 1;
```

We can desugar this to our own syntax:

```
1 let mut y : Int in
2     y = 0;
3     y = 1;
```

We will walk through the example line by line and try to use the semantics of the previous chapter as much as possible. At the `let`-statement, besides initializing `y` to $\perp$, we need to somehow save that `y` is mutable. We can do that by introducing a new function besides the state, $\mathcal{M}$, which tells us whether a variable is mutable or not. For purposes that will become clear later on, we will choose to let this function return $\{\texttt{mut}\}$ if the variable is mutable and let it return $\emptyset$ if the variable is not mutable.

Then in line 2, we assign a value to `y`, which makes that $s(y) = 0$. Then in line 3, instead of rejecting the program because `y` already has a value and thus is not $\perp$, we see that `y` is mutable and accept the assignment if `y` is either $\perp$ or an element from $\mathbb{Z}$. If `y` were not mutable, only $\perp$ would have sufficed.

1.3.2 Nonmutable borrow

The example of this section looks at just borrowing and no mutability.

```
1 let x = 0;
2 let z;
3 let y = &x;
4 z = y;
```

If we write this in our syntax, we get:

```
1 let x: Int in (
2     x = 0;
```

```
3       let z: &Int in (
4           let y: &Int in (
5               y = & x;
6               z = y;
7           )
8       )
9   )
```

We will again walk through this example step by step. At the start, we have no borrows and every variable is undefined, i.e. $-$.

```
1   let x: Int in
2       x = 0;
```

We will again use a state to keep track of the values of the variables. After the first line, we will say `x` is now $\perp$: $s(x) = \perp$. After line 2, `x` will be equal to $0 \in \mathbb{Z}$. Now we move on to the next `let`-statements.

```
1   let x: Int in
2       x = 0;
3       let z: &Int in
4           let y: &Int in
```

Now `z` has been declared. So we have $s(x) = 0$ and $s(z) = \perp$. For all other variables v, we have $s(v) = -$ after line 3. After line 4, we also have $s(y) = \perp$.

```
1   let x: Int in (
2       x = 0;
3       let z: &Int in (
4           let y: &Int in (
5               y = & x;
6               z = y;
```

Now we have $s(y) = 0$ in line 5 and we have $s(x) = 0$. As you might remember from the chapter about Rust, you can have multiple 'readers' to one location, as long as you cannot change the content of that location (to prevent data races). After line 6, the value that y had will be moved to z. So $s(y) = -$ and $s(z) = 0$.

```
1   let x: Int in (
2       x = 0;
3       let z: &Int in (
```

```
        let y: &Int in (
            y = & x;
            z = y;
        )
```

Now z goes out of scope, so we have $s(z) = -$, while still having $s(x) = 0$.

```
let x: Int in (
    x = 0;
    let z: &Int in (
        let y: &Int in (
            y = & x;
            z = y;
        )
    )
)
```

After the second bracket, nothing happens, as $s(y)$ already was $-$. After the last bracket, x is also again set to $-$.

1.3.3 Mutable borrow

For the third example, we will look at some mutable borrowing. As explained in the chapter about Rust, the following program produces an error, as it makes two mutable references to the same piece of memory. This makes the code prone to data races.

```
let mut x=0;
let y = & mut x;
let z = & mut x;
```

If we write this in our syntax, we get:

```
let mut x: Int in (
    x = 0;
    let y: &Int in (
        y = & mut x;
        let z: &Int in (
            z = & mut x;
        )
    )
)
```

So now we will walk through the code and see where the code should not be accepted.

```
let mut x: Int in (
    x = 0;
```

After the `let`, we have $s(x) = -$. Then after the `x = 0`, we have $s(x) = 0$. This should not be surprising in the light of the previous chapter and previous example. However, it is very important to not that our `x` here is mutable. So we will say $\mathcal{M}(x) = \{\texttt{mut}\}$.

```
let mut x: Int in (
    x = 0;
    let y: &Int in (
        y = & mut x;
```

For `y`, the same story holds as for `x` above, but `y` is not mutable. So at the end, $s(y) = 0$ and $\texttt{mut} \notin \mathcal{M}(y)$. However, we want `x` to know that it has been borrowed, so cannot be changed directly anymore and we want `y` to know where it borrowed from, so we can release the borrow after the scope of `y` ends. Therefore, we need to do some additional bookkeeping. We will say $\mathcal{M}(x) = \{\texttt{bor}, \texttt{mut}\}$ and $\mathcal{M}(y) = \{x\}$. So our $\mathcal{M}$ will have a signature of the form $\mathbf{Var} \to \mathcal{P}(\{\texttt{bor}, \texttt{mut}\} \cup \mathbf{Var})$.

```
let mut x: Int in (
    x = 0;
    let y: &Int in (
        y = & mut x;
        let z: &Int in (
            z = & mut x;
```

An error should occur here, as this will be the second borrow to the mutable variable `x`. This could be discovered by looking at $\mathcal{M}(x)$ and seeing that both `mut` and `bor` are in there. No new borrow could be made.

Looking at the same program, but without the last `let`, we get the following structure (where `|` is used to designate $\bot$):

```
//s(x) = -, s(y) = -, M(x) = {}, M(y) = {}
let mut x: Int in (   //s(x) = |, M(x) = {mut}
    x = 0;            //s(x) = 0
    let y: &Int in (  //s(y) = |
         y = & mut x; //s(y) = 0, M(x) = {bor,mut}, M(y) = {x}
    )                 //s(y) = -, M(y) = {}, M(x) = {mut}
)                     //s(x) = -, M(x) = {}
```

Here, we can see that at the end, everything is freed. After the borrow of `x` ends, we remove the borrow status from our $\mathcal{M}$, as `x` now can be borrowed again.

1.3.4 Another example of mutable borrowing

The following piece of code is incorrect and produces the error `re-assignment of immutable variable`. This is because even though `y` and `z` are mutable, `x` is not, and thus cannot be reassigned to borrow from `z`.

```
let mut y=0;
let mut z=1;
let x = & mut y;
x = & mut z
```

As discussed above, we only accept an assignment of `x` if

- `x` is `mut` and has value $-$ or some value from $\mathbb{Z}$, or;
- `x` is not `mut` and has value $-$.

Neither is the case here, so we reject this piece of code, as does the Rust compiler.

In order to fix this, we make `x` mutable, as was done in the following piece of code. This does compile.

```
let mut y=0;
let mut z=1;
let mut x = & mut y;
x = & mut z
```

Our analysis of this piece of code is the following:

```
//s(y)=s(x)=s(z)=-,M(x)=M(y)=M(z)={}
let mut y : Int in (          //s(y)=|, M(y)={mut}
    y=0;                      //s(y)=0
    let mut z : Int in (      //s(z)=|, M(z)={mut}
        z=1;                  //s(z)=1
        let mut x : &Int in ( //s(x)=|, M(x)={mut}
            x = & mut y;      //s(x)=0, M(x)={y},M(y)={mut,bor}
            x = & mut z;      //s(x)=1, M(x)={y,z},M(z)={mut,bor}
        )                     //s(x)=-, M(x)={},M(y)=M(z)={mut}
    )                         //s(z)=-, M(z)={}
)                             //s(y)=-, M(y)={}
```

It might be surprising to see that after the `x` is reassigned, the borrow of `y` is not undone. However, this is the case in the Rust compiler. The compiler (version 1.22.1) does not accept a reborrow of y even after x has gotten a new value. So adding the line `let v = & mut y` after line 8 leads to a compile error.

Now we will look at one more example before moving on to the semantics. The following piece of code passes through the compile time checker.

```
1 let y=0;
2 let z=1;
3 let mut x = & y;
4 x = & z
```

We will again analyze this piece to see what happens here.

```
1  //s(y)=s(x)=s(z)=-,M(x)=M(y)=M(z)={}
2  let y : Int in (                  //s(y)=|
3      y=0;                          //s(y)=0
4      let z : Int in (              //s(z)=|
5          z=1;                      //s(z)=1
6          let mut x : &Int in (     //s(x)=|,M(x)={mut}
7              x = & y;              //s(x)=0
8              x = & z;              //s(x)=1
9          )                         //s(x)=-,M(x)={}
10     )                             //s(z)=-
11 )                                 //s(y)=-
```

In comparison to the previous example, very little bookkeeping needed to be done. We only need to keep track of the values of the variables, and keep in mind that `x` is mutable. There is no need to remember which variables have been borrowed or not, since we can have zero, one or multiple borrows anyways.

1.4 Semantics: Framework

Just as in the chapter about moving, we need some mathematical definitions. This section is dedicated to updating our previous definitions and including some new definitions.

Variables and expression

We will continue to use $\mathbb{Z}_{ext}$ in the same way we did in the previous chapter.

Definition 1.4.1. We define the function $\mathcal{V} : \mathbf{Exp} \to \mathcal{P}(\mathbf{Var})$ recursively by:

$$\begin{aligned}
\mathcal{V}(i) &= \emptyset \\
\mathcal{V}(x) &= \{x\} \\
\mathcal{V}(e_1 + e_2) &= \mathcal{V}(e_1) \cup \mathcal{V}(e_2) \\
\mathcal{V}(\&x) &= \emptyset \\
\mathcal{V}(\&\texttt{mut}x) &= \emptyset
\end{aligned}$$

This function is used to collect all variables in an expression that are not being borrowed in that expression. We used this function in the previous chapter already.

Definition 1.4.2. We define the function $\mathcal{B} : \mathbf{Exp} \to \mathcal{P}(\mathbf{Var})$ recursively by:

$$\begin{aligned}
\mathcal{B}(i) &= \emptyset \\
\mathcal{B}(x) &= \emptyset \\
\mathcal{B}(e_1 + e_2) &= \mathcal{V}(e_1) \cup \mathcal{V}(e_2) \\
\mathcal{B}(\&x) &= \{x\} \text{ if } \texttt{mut} \in \mathcal{M}(x) \\
\mathcal{B}(\&\texttt{mut}x) &= \{x\} \text{ if } \texttt{mut} \in \mathcal{M}(x)
\end{aligned}$$

This function is used to collect all variables in an expression that are being borrowed in that expression, but only if they are mutable. This is done because, as we saw in the examples above, there can only arise problems if a mutable variable has already been borrowed. This function thus gathers all 'at risk' variables in an expression.

Definition 1.4.3. We define the function $\mathcal{C} : \mathbf{Exp} \to \mathcal{P}(\mathbf{Var})$ recursively by:

$$\begin{aligned}
\mathcal{C}(i) &= \{x\} \\
\mathcal{C}(x) &= \emptyset \\
\mathcal{C}(e_1 + e_2) &= \mathcal{V}(e_1) \cup \mathcal{V}(e_2) \\
\mathcal{C}(\&x) &= \{x\} \\
\mathcal{C}(\&\texttt{mut } x) &= \{x\}
\end{aligned}$$

This function is used to collect all variables in an expression.
We also need to update the evaluation function.

Definition 1.4.4. The evaluation function $\mathcal{A} : \mathbf{Exp} \times \mathbf{State} \to \mathbb{Z}_{ext}$ is defined by:

$$\begin{aligned}
\mathcal{A}[\![i]\!]s &= \mathcal{N}[\![i]\!] \\
\mathcal{A}[\![x]\!]s &= s(x) \\
\mathcal{A}[\![e_1 + e_2]\!]s &= \mathcal{A}[\![e_1]\!]s + \mathcal{A}[\![e_2]\!]s \qquad \text{if } \mathcal{A}[\![e_1]\!]s \in \mathbb{Z} \text{ and } \mathcal{A}[\![e_2]\!]s \in \mathbb{Z} \\
\mathcal{A}[\![e_1 + e_2]\!]s &= - \qquad \text{otherwise} \\
\mathcal{A}[\![\& \ x]\!]s &= \mathcal{A}[\![x]\!]s \\
\mathcal{A}[\![\&\texttt{mut } x]\!]s &= \mathcal{A}[\![x]\!]s
\end{aligned}$$

Notice it is the same as in the previous chapter, except for the added last lines.

1.5 Semantics: Small step

In this chapter, we will move to the small step semantics right away and skip the big step semantics. This is because the latter, as mentioned in the previous chapter, allows for better separation in a semantic derivation system and a compile time check system. The semantic rules are of similar form to the previous chapter, except that we have rules of the form $\langle S, L, s, \mathcal{M}\rangle \Rightarrow \langle S', L', s', \mathcal{M}'\rangle$.

S and S' are again statements as defined in Definition 1.2.1. L and L' are again lists as defined in the previous chapter. s and s' are again states as defined in the previous chapter. $\mathcal{M}$ is a function with signature $\mathbf{Var} \to \mathcal{P}(\{\texttt{bor}, \texttt{mut}\} \cup \mathbf{Var})$.

Definition 1.5.1. We define the following semantic derivation rules (name on the left):

$$[\text{load}_{\text{sosb}}] \quad \langle \texttt{skip}, I :: L, s, \mathcal{M}\rangle \Rightarrow \langle I, L, s, \mathcal{M}\rangle$$

$$[\text{comp}_{\text{sosb}}] \quad \langle S_1; S_2, L, s, \mathcal{M}\rangle \Rightarrow \langle S_1, S_2 :: L, s, \mathcal{M}\rangle$$

$$[\text{ass}_{\text{sosb}}] \quad \langle \texttt{x = } e, L, s, \mathcal{M}\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -], \mathcal{M}[\forall y \in \mathcal{B}(e), y \mapsto \mathcal{M}(y) \cup \{\texttt{bor}\}][x \mapsto \mathcal{M}(x) \cup \mathcal{B}(e)]\rangle$$

$$[\text{let}_{\text{sosb}}] \quad \langle \texttt{let } x : \tau \texttt{ in } S, L, s\rangle \Rightarrow \langle S, (x, s(x), \mathcal{M}(x)) :: L, s[x \mapsto \bot], \mathcal{M}\rangle$$

$$[\text{letmut}_{\text{sosb}}] \quad \langle \texttt{let mut } x : \tau \texttt{ in } S, L, s, \mathcal{M}\rangle \Rightarrow \langle S, (x, s(x), \mathcal{M}(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle$$

$$[\text{set}_{\text{sosb}}] \quad \langle (x, v, m), L, s, \mathcal{M}\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto v], \mathcal{M}[\forall y \in \mathcal{M}(x) \cap \mathbf{Var} \setminus m, y \mapsto \mathcal{M}(y) \setminus \{\texttt{bor}\}][x \mapsto m]\rangle$$

Note that $\langle \texttt{skip}, \texttt{Nil}, s, \mathcal{M}\rangle$ has no derivation; this is the/an end state.

The $[\text{load}_{\text{sosb}}]$ rule and the $[\text{comp}_{\text{sosb}}]$ rule are assumed to be self-explanatory. As for the $[\text{ass}_{\text{sosb}}]$ rule, besides updating the state like we used to, we need to add all borrowed variables to the $\mathcal{M}$ of x. We do this by gathering all those variables with the $\mathcal{B}$ function. For all those variables, their $\mathcal{M}$ needs to include `bor` from now on as they have been borrowed.

For the $[\text{let}_{\text{sosb}}]$ and $[\text{letmut}_{\text{sosb}}]$ rules, we added the $\mathcal{M}$-function to the tuple, because we also need to reset that one in the reset. Also, in the $[\text{letmut}_{\text{sosb}}]$ rule, we need $\mathcal{M}$ to reflect that x is mutable.

For the [set_{sosb}] rule, besides resetting s and $\mathcal{M}$, we also need to let all the variables x has borrowed from know that the borrow has stopped. This should not happen when a similarly named variable x already borrowed the variable in the higher scope. This is done by removing all variables in m from the set under consideration.

Determinism

Again, we would like our semantics to be deterministic. First, we need the following Lemma.

Lemma 1.5.1. *For every statement S, for every list L, for every state s, s', if $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$, then it must be the case that $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, (x, s(x)) :: L, s''\rangle \Rightarrow \langle (x, s(x)), L, s''\rangle \Rightarrow \langle \texttt{skip}, L, s'\rangle$ for some state s''.*

Proof. The proof is similar to lemma **??** and therefore omitted. ♢

Now we can prove that the semantics are deterministic.

Theorem 1.5.2. *For every program instruction S, every state s, s', s'', every list L, if $\langle S, L, s, \mathcal{M}\rangle \Rightarrow^* \langle \texttt{skip}, L, s', \mathcal{M}'\rangle$ and $\langle S, L, s, \mathcal{M}\rangle \Rightarrow^* \langle \texttt{skip}, L, s'', \mathcal{M}'\rangle$, then $s' = s''$ and $\mathcal{M}' = \mathcal{M}''$.*

Proof. The proof proceeds by induction on the structure of S, the program instruction. We will only prove the case for $\texttt{let mut}\ x : \tau\ \texttt{in}\ S'$. The other cases are similar to this case and to the proof in the previous chapter.

- $S = \texttt{let mut}\ x : \tau\ \texttt{in}\ S'$: assume $\langle \texttt{let mut}\ x : \tau\ \texttt{in}\ S', L, s, \mathcal{M}\rangle \Rightarrow^* \langle \texttt{skip}, L, s', \mathcal{M}'\rangle$ and $\langle \texttt{let mut}\ x : \tau\ \texttt{in}\ S', L, s, \mathcal{M}\rangle \Rightarrow^* \langle \texttt{skip}, L, s'', \mathcal{M}''\rangle$. The only rule that can be applied is [$\text{letmut}_{\text{sosb}}$]. This gives us

$$\langle \texttt{let mut}\ x : \tau\ \texttt{in}\ S', L, s, \mathcal{M}\rangle \Rightarrow$$

$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^* \langle \texttt{skip}, L, s', \mathcal{M}'\rangle$$

and

$$\langle \texttt{let mut}\ x : \tau\ \texttt{in}\ S', L, s, \mathcal{M}\rangle \Rightarrow$$

$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^* \langle \texttt{skip}, L, s'', \mathcal{M}''\rangle$$

By Proposition 1.5.1, we know that the latter $\Rightarrow^*$ can be broken up into smaller parts, so we have

$$\langle \texttt{let mut}\ x : \tau\ \texttt{in}\ S, L, s, \mathcal{M}\rangle \Rightarrow$$

$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^*$$
$$\langle \texttt{skip}, (x, s(x)) :: L, s''', \mathcal{M}'''\rangle \Rightarrow$$
$$\langle (x, s(x)), L, s''', \mathcal{M}'''\rangle \Rightarrow \langle \texttt{skip}, L, s', \mathcal{M}'\rangle$$

and

$$\langle \texttt{let mut } x : \tau \texttt{ in } S, L, s, \mathcal{M}\rangle \Rightarrow$$
$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^*$$
$$\langle \texttt{skip}, (x, s(x)) :: L, s'''', \mathcal{M}''''\rangle \Rightarrow$$
$$\langle (x, s(x)), L, s'''', \mathcal{M}''''\rangle \Rightarrow \langle \texttt{skip}, L, s'', \mathcal{M}''\rangle$$

Now we can apply the induction hypothesis to

$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^* \langle \texttt{skip}, (x, s(x)) :: L, s''', \mathcal{M}'''\rangle$$

and

$$\langle S', (x, s(x)) :: L, s[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]\rangle \Rightarrow^* \langle \texttt{skip}, (x, s(x)) :: L, s'''', \mathcal{M}''''\rangle$$

That gives us $s''' = s''''$ and $\mathcal{M}''' = \mathcal{M}''''$. Then $\langle (x, s(x)), L, s''', \mathcal{M}'''\rangle \Rightarrow \langle \texttt{skip}, L, s'''[x \mapsto v], \mathcal{M}'''[\forall y \in \mathcal{M}(x) \cap \mathbf{Var} \setminus m, y \mapsto \mathcal{M}(y) \setminus \{\texttt{bor}\}][x \mapsto m]\rangle$, so $s' = s'''[x \mapsto v] = s''$ and $\mathcal{M}' = \mathcal{M}'''[\forall y \in \mathcal{M}(x) \cap \mathbf{Var} \setminus m, y \mapsto \mathcal{M}(y) \setminus \{\texttt{bor}\}][x \mapsto m] = \mathcal{M}''$, which means $s' = s''$ and $\mathcal{M}' = \mathcal{M}''$.

This proves the theorem. ♢

1.5.1 Compile time check

We can now look at the compile time checker. In the following definition we show the actual rules. An explanation will be given below.

Definition 1.5.2. The *compile time checker* is a derivation system that has the following rules

$$\begin{aligned}
[\texttt{skip}, \texttt{Nil}, r, \mathcal{M}] &\rightarrow \texttt{true} \\
[\texttt{skip}, P :: L, r, \mathcal{M}] &\rightarrow [P, L, r, \mathcal{M}] \\
[S_1; S_2, L, r, \mathcal{M}] &\rightarrow [S_1, S_2 :: L, r, \mathcal{M}] \\
[\texttt{let } x : \tau \texttt{ in } S, L, r, \mathcal{M}] &\rightarrow [S, (x, r(x), \mathcal{M}(x)) :: L, r[x \mapsto \bot], \mathcal{M}] \\
[\texttt{let mut } x : \tau \texttt{ in } S, L, r, \mathcal{M}] &\rightarrow [S, (x, r(x), \mathcal{M}(x)) :: L, r[x \mapsto \bot], \mathcal{M}[\texttt{x} \mapsto \{\texttt{mut}\}]] \\
[(x, v, m), L, r, \mathcal{M}] &\rightarrow [\texttt{skip}, L, r[x \mapsto v], \mathcal{M}[\forall y \in \mathcal{M}(x) \cap \mathbf{Var}, \\
&\qquad y \mapsto \mathcal{M}(y) \setminus \{\texttt{bor}\}][\texttt{x} \mapsto \texttt{m}]
\end{aligned}$$

Also, if $\texttt{mut} \in \mathcal{M}(x), r(x) = \bot$ or $r(x) = \star, \forall y \in \mathcal{C}(e), r(y) = \star$ and $\forall y \in \mathcal{B}(e), \neg\texttt{bor} \in \mathcal{M}(e)$, **OR** if $\neg\texttt{mut} \in \mathcal{M}(x), r(x) = \bot, \forall y \in \mathcal{C}(e), r(y) = \star$ and $\forall y \in \mathcal{B}(e), \neg\texttt{bor} \in \mathcal{M}(e)$, we have the rule

$$[x = e, L, r, \mathcal{M}] \rightarrow$$
$$[\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -], \mathcal{M}[\forall y \in \mathcal{B}(e), y \mapsto \mathcal{M}(y) \cup \{\texttt{bor}\}][x \mapsto \mathcal{M}(x) \cup \mathcal{B}(e)]]$$

If neither of these conditions hold, we have the rule

$$[x = e, L, r, \mathcal{M}] \rightarrow \texttt{false}$$

Explanation

The rules are almost identical to those of the semantics. The difference here is the set of conditions that should apply in an assignment.

First of all, we distinguish two cases: whether x is mutable or not. If x is mutable, then $r(x)$ should be either $\bot$ or $\star$ as that means x is at least initialized and might or might not have a value. Also, all variables in the expression should be $\star$. Otherwise the expression e has no well-defined value. Lastly, for all variables that are being borrowed in e and are mutable (so $\mathcal{B}(e)$), we check that they have not been borrowed before. If they have, they cannot be borrowed again after all.

The other case is that x is not mutable. Then we only accept it if x has been initialized but has not been assigned a value yet. So $r(x) = \bot$. The rest is the same as above.

Termination

We again want to prove that this compile time checker always terminates.

Theorem 1.5.3. *The compile checker from definition 1.5.2 always terminates.*

Proof. This proof is almost completely similar to that of **??**, if we define $|\texttt{let mut } x : \tau \texttt{ in } S| = |S| + 3$. We therefore omit the details. ♢

Progress and preservation

Just like in the previous chapter, we will prove progress and preservation. We start with preservation.

Theorem 1.5.4. *For all program instructions S, S', lists L, L', states s, s', reduced states r, and for all $\mathcal{M}$, $\mathcal{M}'$: if $[S, L, r, \mathcal{M}] \to^* \texttt{true}$ and $\langle S, L, s, \mathcal{M}\rangle \Rightarrow \langle S', L', s', \mathcal{M}'\rangle$ with r related to s, then there is an r' that is related to s' and we have $[S', L', r', \mathcal{M}'] \to^* \texttt{true}$.*

Proof. We assume $\langle S, L, s, \mathcal{M}\rangle \Rightarrow \langle S', L', s', \mathcal{M}'\rangle$. We look at the different possible rules $\Rightarrow$.

- $\langle \texttt{skip}, I :: L, s, \mathcal{M}\rangle \Rightarrow \langle I, L, s, \mathcal{M}\rangle$. We assume $[\texttt{skip}, I :: L, r, \mathcal{M}] \to^* \texttt{true}$. We try to determine which derivation steps could have let to `true`. The only possible step is

  $$[\texttt{skip}, I :: L, r, \mathcal{M}] \to [I, L, r, \mathcal{M}]$$

  Then we have our $r' = r$, and obviously we also have $[I, L, r, \mathcal{M}] \to^* \texttt{true}$.
- $\langle S_1; S_2, L, s, \mathcal{M}\rangle \Rightarrow \langle S_1, S_2 :: L, s, \mathcal{M}\rangle$. We assume $[S_1; S_2, L, r, \mathcal{M}] \to^* \texttt{true}$. We try to determine which derivation steps could have led to `true`. The only possible step is

  $$[S_1; S_2, L, r, \mathcal{M}] \to [S_1, S_2 :: L, r, \mathcal{M}]$$

  Then we have our $r' = r$ and obviously we also have $[S_1, S_2 :: L, r, \mathcal{M}] \to^* \texttt{true}$.

- The other cases are similar and therefore omitted.

This proves our theorem. ♢

Theorem 1.5.5. *For all program instructions S, reduced states r, lists L, and for all $\mathcal{M}$'s: if $[S, L, r, \mathcal{M}] \to^* \texttt{true}$, then $S = \texttt{skip}$ and $L = \texttt{Nil}$ or we have $\langle S, L, s, \mathcal{M}\rangle \Rightarrow \langle S', L', s', \mathcal{M}'\rangle$ for some S', L', s' and M', and every s related to r.*

Proof. We assume $[S, L, r, \mathcal{M}] \to^* \texttt{true}$. We look at the possible derivation steps that could have led to this form.

- $[\texttt{skip}, \texttt{Nil}, r, \mathcal{M}] \to \texttt{true}$. This means $S = \texttt{skip}$ and $L = \texttt{Nil}$ and that means we are done.
- $[\texttt{skip}, P :: L, r, \mathcal{M}] \to [P, L, r, \mathcal{M}]$. This means we have $\langle \texttt{skip}, P :: L, s, \mathcal{M}\rangle$ with s any state related to r. We can then apply the rule $[\text{load}_{\text{sos}}]$, to get $\langle \texttt{skip}, P :: L, s, \mathcal{M}\rangle \Rightarrow \langle P, L, s, \mathcal{M}\rangle$, which means we are done.
- The other cases are similar and therefore omitted.

This proves our theorem ♢

Bibliography

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamari, Z., and Ryzhyk, L. (2017). System programming in Rust: Beyond safety. *ACM SIGOPS Operating Systems Review*, 51(1):94–99.

Benitez, S. (2016). Short paper: Rusty Types for Solid Safety. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 69–75. ACM.

Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. (2003). Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, 38(7):69–80.

Ding, Y., Duan, R., Li, L., Cheng, Y., Zhang, Y., Chen, T., Wei, T., and Wang, H. (2017). Poster: Rust sgx sdk: Towards memory safety in intel sgx enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2491–2493. ACM.

Jespersen, T. B. L., Munksgaard, P., and Larsen, K. F. (2015). Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 13–22. ACM.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2017). Rustbelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66.

Leffler, S. (2017). *Rust's Type System is Turing-Complete*. https://sdleffler.github.io/RustTypeSystemTuringComplete/.

Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., and Levis, P. (2017). The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 1. ACM.

Matsakis, N. D. and Klock II, F. S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.

Reed, E. (2015). Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*.