Bachelor Thesis
Computing Science

Radboud University

The Semantics of Ownership and Borrowing in the Rust Programming Language

Author:
Nienke Wessel
s4598350

First supervisor:
Dr. Freek Wiedijk
`freek@cs.ru.nl`

Second supervisor:
Marc Schoolderman
`m.schoolderman@cs.ru.nl`

July 10, 2019

Abstract

In order to prove properties of programming languages, we need to have a mathematical representation of the language. This thesis formalizes a subset of the Rust Programming Language. The focus is on ownership and borrowing, properties that are paramount to the memory safety that Rust claims to have. Besides our semantic rules, we have a compile time check system to model that Rust checks programs compile time. We explain our semantic rules and prove some properties of these rules, such as progress and preservation in combination with the compile time checker.

Preface

This document contains the Bachelor Thesis I wrote for the Computing Science Bachelor at Radboud University, Nijmegen.

The topic of this thesis originates from the bachelor courses "Semantics and Correctness" and "Computation Models". For the latter, Sander Hendrix and I did a project on the natural semantics of Rust, mostly concerning memory safety aspects that Rust claims to have. This project sparked my interests for describing the semantics of programming languages and looking into how mathematics and computing science can be combined in this field. The report that we then made formed the basis for this thesis.

There are a lot of people that helped me along the way, and I would like to thank them, so here we go; thanks mom and dad for not getting angry at me that I was not finishing my bachelor's quicker. Thanks friends for accepting that I had to cancel some chillings in order to finally finish the thing. I would like to thank Sander Hendrix for doing the project on Rust together. I want to thank Engelbert Hubbers and Erik Barendsen for their feedback on our project.

Most importantly, I want to thank Freek Wiedijk and Marc Schoolderman for being my supervisors and helping me turn this vague idea ('something with Rust and semantics') into this thesis. And for all the feedback and time, of course!

Anyways, dear reader, have fun reading!

Contents

Chapter 1

Introduction

As computers are found almost everywhere nowadays, we would hope these computers would always do as we expect them to do. However, so far very few programs and programming languages have actually been proven correct. Intensive testing is the best software developers can do. Luckily, much work is being done on the more rigorous *proving* that a computer program does what it is designed to do. This thesis aims to join in those efforts.

The thesis is about the Rust programming language (Matsakis and Klock II, 2014). The makers of this programming language have made several claims about the safety, and specifically memory safety, of their language. Among other things, it should be impossible to get data races or dangling pointers in your programs. All safety aspects are guaranteed at compile time; if your Rust program contains hazards, the compiler will reject the program. Besides that, all checks should finish, even if the actual program contains an infinite loop or does not finish for some other reason. Due to executing these checks compile time, no run time penalty should exist. This makes the language different from, for example, C or Java. In C, no memory safety is guaranteed because arbitrary pointer arithmetic is allowed[1]. In Java, memory safety is guaranteed by run time checks instead of compile time checks.

The combination of memory safety and guaranteeing it compile time is the reason it has been argued that Rust is a good language to use for making a kernel (Levy et al., 2017) or to use in combination with other security measures such as Software Guard Extensions (Ding et al., 2017). Other benefits beyond memory safety have also been attributed to Rust and its features (Balasubramanian et al., 2017).

[1]However, there have been proposals to make extensions or versions of C that are memory safe, even some that rely on compile time/static checks. See for example Dhurjati et al. (2003)

1.1 Previous work

However, little has been done to actually verify all the claims made by the developers of Rust. Much of the previous work concerning Rust has been done on improving the language itself and making it more versatile. See for example Jespersen et al. (2015). Little has been done to actually confirm that it all works.

The most notable project that does look into the actual memory safety of Rust, is the Rustbelt project (Jung et al., 2017). In this project, researchers have formalized a subset of Rust and proved some libraries to be safe in Coq. They use separation logic in order to model the memory of the computer during computation.

Benitez (2016) has also made a formalization of a very small subset of Rust. He modeled references and ownership as capabilities on different value locations. For example, a variable `x` could have the capabilities `read` or `write` on a location to indicate that it could be read or modified respectively. He separated memory locations from computation. In Reed (2015) something similar is done, and some memory safety claims are proven.

1.2 The current work

The current work aims to differentiate from the above work by showing how we can model the memory safety aspects of Rust, whilst using simple semantic rules, instead of more advanced mechanisms such as separation logic. The proofs here should be readable for anyone with only basic background in semantics theory, and gives the reader insight into what is happening in Rust both in practice and on a formal level.

The research question that is central in this thesis is *How does one formally describe ownership in Rust in a simple way?*

In order to answer this question, we looked at several sub-questions:

- What is ownership in Rust?
  - What is it used for?
  - What syntax is relevant to denote ownership?
  - What is borrowing?
  - What are lifetimes?
- What subset of Rust is necessary to describe ownership?
- What type of semantics is the most useful to describe ownership?
- How can we model the compile time checker best?
- Can we prove safety from a set of semantic rules and a compile time checker?

1.3 Scope of this thesis

As for Rust, we will only be focusing on two major aspects of the language; the concepts of *ownership* and *borrowing*. All you need to know and a little bit more is explained in Chapter 2. The version of Rust compiler used for this thesis was 1.22.1.

We will mainly leave the types for what they are and focus on the memory safety aspect. The specific types are not particularly interesting. First of all, the focus of this thesis is on memory safety. Types are not particularly relevant for that. Also, the type system is not very different from other programming languages, which makes that adding them would also not be interesting from a research point of view. Therefore, types are ignored to keep the scope of this thesis limited. We will only look at whether an item is mutable or not, a concept explained in Chapter 2.

Rust also offers the possibility to surpass some of its memory safety guards by giving the user the option to write `unsafe` code. Many of Rust's standard libraries depend on `unsafe` code. However, the developers of Rust claim that if users just make use of the interfaces and do not write unsafe code themselves, their code should be safe. In this thesis, we will not look at unsafe code. Jung et al. (2017) have developed a set of semantic rules to incorporate both safe and unsafe code.

The compiler finishes almost always, even if the program contains an infinite loop. The actual Rust compiler is Turing Complete (Leffler, 2017), but there are no known 'normal' cases where it does not finish. When checking a program compile time, we cannot simply run the program in the check if the program loops infinitely, because the checker should not loop. The compiler needs to have a level of abstraction from the program. We will show how we did that in a later chapter. On the other hand, we will not follow the Rust compiler completely. The compiler often gives warnings if there is something of which it suspects that you did not mean to write it down that way. We will ignore these warnings. Our goal here is to accept the same programs as the Rust compiler accepts, so we only take actual errors into account.

1.3.1 On safety and memory safety

Before moving on, we will quickly walk through our definitions of safety and memory safety. In this thesis, memory safety of a program will be used to indicate that a program is protected from either bugs or vulnerabilities that have to do with memory access, such as buffer overflows or dangling pointers.

When talking about semantics, this translates to having no problems in our model of the memory. For example, that we cannot have an uninitialized variable in an expression or that we cannot assign a variable a value twice, unless it is mutable (see Chapter 2 for an explanation of the term).

Safety means that a program cannot get in trouble in general. For example, we do not want any errors. When talking about semantics, an unsafe program can either have explicit crashes, or can get stuck in the semantics and there is no 'step' to take in a semantic derivation system. Neither should happen in a safe program.

1.4 Why make a formal semantics?

Before looking at the rest of the thesis, we will first briefly discuss why it is important to formalize programming languages and why we want to make formal semantics. Besides giving insight in what is happening in the programming language, there are multiple goals.

First of all, a set of formal rules can help prove specific Rust programs to be correct. In order to prove that a program does what it is supposed to do, we first need to know what the program does exactly. Then we can prove that this is indeed what was intended. A formal description of the language is therefore necessary.

The second goal is to say something about programs that process Rust, such as compilers. Once we know how syntax relates to the meaning of a program, we can prove that the compilers behave correctly. Besides compilers, this could also apply to static analyzers, optimization programs, etc.

1.5 The next chapters

The following chapter is concerned with Rust. We will give a brief introduction to Rust and explain the relevant features. After that, in Chapter 3 *Moving*, we will formalize the most basic form of the ownership concept. All (mathematical) details will be explained rather thoroughly. In the chapter after that, Chapter 4 *Borrowing*, we will add two new features: mutability and borrowing. By then, we have a pretty rigorous basis from Chapter 3, so we will not provide as many details anymore. This gives us the space to actually focus on what is happening there, instead of getting lost in the mathematics. In the final chapter, Chapter 5 *Conclusion & Discussion*, we briefly discuss what we have done and the implications for future work.

Chapter 2

Rust

This chapter provides an introduction to Rust, specifically to the features that are considered interesting for this thesis.

Rust is an imperative programming language, with some functional and object orientated aspects. The language has similarities to C and especially C++, but is different in that it claims to provide memory safety whereas C does not. One of the most important concepts in this *RAII*, which stands for Resource Acquisition Is Initialization. This term was introduced by Stroustrup (1994) when talking about the design of C++.

The idea behind RAII is that a resource (a piece of memory) is allocated during the creation (initialization) of the data/variable and a resource is deallocated during the the destruction of the data/variable. As long as the programmer does not "lose" the data (i.e. a reference to the data), there are no memory leaks. To help the programmer do this, there are several important terms/concepts which will be discussed here. We start with **ownership** and **borrowing**. At the end of this chapter, we briefly also discuss **lifetimes**, as this is an important aspect of memory safety in Rust. We will not be using lifetimes explicitly, but it is helpful for understanding Rust in general and the syntax we chose.

2.1 Ownership

Since variables are in charge of freeing their own resources, resources must have exactly one owner. Otherwise, a resource could be freed more than once by different owners or left to dangle if there are no owners at all. This concept is called **ownership** in Rust terminology.

When programming, this can lead to problems. Sometimes it is convenient to have a variable access the resource of another variable. However, accessing this resource means there are two owners: the variable that is accessing the resource and the original variable that had the resource. This is not allowed in Rust. Luckily, Rust has some ways to deal with these kind of scenarios. The easiest solution is *moving* the ownership of a resource to another variable.

In Rust terminology, this is called a **move**. After moving the ownership, the previous owner can no longer use the resource.

The following example illustrates this.

```
1 let vector = vec![37, 42];
2 let vector2 = vector;
3 println!("vector[0] + vector[1] = {}", vector[0] + vector[1]);
```

This program gives the compile error `use of moved value: ‘vector‘`.

We will briefly walk through the program line by line. The variable `vector` is initialized in line 1. That means `vector` owns a piece of memory to store the values of the vector. Next, the ownership is transferred to `vector2`. That means `vector2` now owns the piece of memory and `vector` cannot access it anymore. Then, when `vector` tries to access that piece of memory in line 3 to print its value, it cannot do so because the ownership has been transferred. When compiling a Rust program such as in this example, the compiler checks whether the ownership rules are abided. If this is not the case, it produces an error and the program does not compile.

Besides assignments, there are also other methods to take ownership of a piece of memory. Another frequently occurring method is passing the variable as an argument to a function. For example:

```
fn function(v : Vec<i32>) {
    // Do something
}

let vector = vec![37, 42];
function(vector);
println!("vector[0] + vector[1] = {}", vector[0] + vector[1]);
```

This gives the same error as before, `error: use of moved value: ‘vector‘`.

In this case, the ownership is moved to the variable `v` in the function `function`. When `function` is finished, all local variables release their resources. So `v` releases the memory space it had for its vector. Later on, `vector` wants access to the resource, but it gave away ownership in the function call, so it can no longer access the resource. Also, the resource was removed after the function returned, so there is also nothing left to access. Again, this error is produced compile time instead of run time.

Moving and ownership are the focus of the next chapter, Chapter 3. There, we will derive semantic rules for these concepts.

2.2 Borrowing

Always moving the variable seems unpractible in many common programming situations. Especially function calls become unpractical. Luckily, Rust also has another way of dealing with the ownership problem.

If you want a variable to temporarily give up its access rights, you can perform a **borrow**. This way, the original owner temporarily gives up its access to the resource to have it returned later on. This ensures that only one variable owns a resource at a time. The checks to make that happen all happen at compile time.

The code below illustrates how borrowing is done in Rust. The code is similar to the example above, but `function` only borrows the resource behind `vector`, which is indicated by the `&` in front of `vector` in the function call. Also, the function signature now contains an `&` to indicate that the function in the end will return the borrowed resource.

```rust
fn function(v : &Vec<i32>) {
// Do something
}

let vector = vec![37, 42];
function(&vector);
println!("vector[0] + vector[1] = {}", vector[0] + vector[1]);
```

This code will give no error, as the piece of memory used by `vector` was only borrowed by `function` and returned to `vector` when `function` was done.

2.2.1 Mutability

So far, we have only looked at what in Rust are called **immutable** variables. That means we cannot change what these variables point to. Doing so results in a (compile time) error. This is also not convenient when programming: we often want to update our variables. To be able to do so, one makes the variables **mutable**.

Some very simple examples can illustrate the difference.

```rust
let x = 0;
x = 1;
```

This is not allowed as `x` is immutable, but the program tries to mutate it anyways. It gives the error `re-assignment of immutable variable ‘x‘`.

```rust
let mut x = 0;
x = 1;
```

This program is allowed, as `x` is now mutable, which is shown by the `mut` in front of it.

The following example shows what happens when we combine mutability with borrowing. Here, the function tries to alter `vector` while it is immutable.

```rust
fn function(v : &Vec<i32>) {
    v.push(1337);
}

let vector = vec![37, 42];
function(&vector);
println!("vector[0] + vector[1] = {}", vector[0] + vector[1]);
```

This gives the compile time error: `cannot borrow immutable borrowed content '*v' as mutable`. This means that by calling `push` on `vector`, vector is passed as if it were mutable, but `vector` was not passed as mutable to `function`.

One can fix this by letting `function` borrow `vector` as a mutable reference.

```rust
fn function(v : &mut Vec<i32>) {
    v.push(1337);
}

let mut vector = vec![37, 42];
function(&mut vector);
println!("vector[0] + vector[1] = {}", vector[0] + vector[1]);
```

This is allowed and compiles.

2.2.2 Rules for borrowing

One can wonder why one would not simply always perform a borrow instead of a move. And if so, why does Rust even have moving and does it not do borrow as a default (as is often done in other programming languages)?

Unlike other programming languages, it is not always possible to borrow a resource in Rust. Borrowing is only allowed in specific cases. These rules were introduced to eliminate data races. They are as follows:

1. The scope of the borrower cannot outlast the scope of the original owner
2. Only one of the following can be true at any moment:
   (a) there are one or more references to a non-mutable resource
   (b) there is exactly one reference to a mutable resource

The first rule is to make sure that the original owner is able to clean up the resource without problems after it is done. If the original owner does not exist anymore when the resource is returned, it cannot clean up the resource. The second rule exists to eliminate data races.

For a data race to happen, there need to be two pointers to the same piece of memory at the same time, at least one of them needs to be writing and their operations are not synchronized. By following the second rule for borrowing, one makes sure a data race cannot exist. If the first part of the second rule holds, the shared resource cannot be written to, as it is immutable. If the the second part of the second rule holds, there are no two pointers in the first place. For a more semantic justification of these rules/this idea, we refer to Boyland (2003).

This means that the following piece of code is allowed (multiple borrows to one immutable owner):

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

The next piece of code is not allowed and produces the error `cannot borrow ‘y‘ as mutable more than once at a time`

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

This concludes the section on borrowing. We will look at some more examples in Chapter 4, when we derive the semantic rules for it.

2.3 Lifetimes

The final important concept in Rust's ownership system is that of **lifetimes**. While this thesis does not use the lifetimes in the semantic rules, a description of Rust's system would not be complete without a section on lifetimes.

Lifetimes are often used by the compiler to keep track of what was mentioned above, when simple scopes are not sufficient. Before we look at the theory of lifetimes, we first take a look at what lifetimes are and what they look like in Rust.

In all code mentioned before, we did not write down the lifetimes explicitly. This is because the Rust compiler is able to deduce the rules in ‘trivial’ cases. Later on, there will be a bit of an explanation on what makes a case trivial for the compiler.

In the following piece of code, it is shown what it would have looked like if we had written the lifetimes explicitly. You can do this for all Rust functions that you write, so that the compiler does not have to derive the lifetimes. Note that it is the same `function` as above.

```rust
fn function<'a>(v : &'a mut Vec<i32>) {
    v.push(1337);
}
```

Strictly speaking, lifetimes are a form of generics, which is why they have generics-like syntax. However for all lifetimes, you are required to place an apostrophe `'`. Just as with generics, one can use any name for a lifetime[1]. However, it is customary that you use `'a`, `'b`, `'c` etc.

To see when lifetimes are useful, we can take a look at a more interesting case.

```rust
fn function(v : &Vec<i32>, w : &Vec<i32>) -> &Vec<i32> {
    if v[0] == 37 {
        v
    }
    else {
        w
    }
}

let vector = vec![37, 42];
let another_vector = vec![42, 37];
let f = function(&vector, &another_vector);
println!("f[0] + f[1] = {}", f[0] + f[1]);
```

This program defines two vectors and passes them to a function. That function then returns one of them. The result is used to print something. Even though this looks like a good program, the Rust compiler will reject it. It will give an error asking for lifetime parameters: `error: missing lifetime specifier`. It is saying that the function's return type contains a borrowed value, but the signature doesn't say what it was borrowed from. The problem here is that rust does not know how the lifetimes of the input and the output are related to each other. The borrow checker now cannot make sure that the rules mentioned above are enforced.

It is important to realize that by adding lifetime notations you cannot actually change the lifetime of any variable. They are a way of telling the compiler how the lifetimes of variables are related. We are basically saying to the Rust compiler that any variables passed to a function that do not adhere to the rules should result in an error.

In order to make the previous example work, we need to add lifetime annotations to the function `function` in such a way that the compiler understands how the lifetimes of the input variables and output variables relate. In this case, it is pretty straightforward:

[1]Except for `'static`, as that has a special meaning,

```
fn function<'a>(v : &'a Vec<i32>, w : &'a Vec<i32>) -> &'a Vec<i32>{
    if v[0] == 37 {
        v
    }
    else {
        w
    }
}

let vector = vec![37, 42];
let another_vector = vec![42, 37];
let f = function(&vector, &another_vector);
println!("f[0] + f[1] = {}", f[0] + f[1]);
```

We are telling the compiler that we have a lifetime `'a` and that the function gets two arguments that will live at least as long as `'a`. The function also returns a variable that will also live at least as long as `'a`. If you would want to use the function but do something different than specified here, this will result in a compile time error.

When concrete vectors are passed to `function` the lifetime that will be substituted for `'a` is the part of the scopes of `v` and `w` that overlap. Since scopes are always nested, this comes down to that `'a` is the smaller of the lifetimes of `v` and `w`. So, the returned reference will be guaranteed to be valid at least as long as the shorter of `v` and `w`.

However, we can also manipulate the order in which variables are declared. This will change the lifetimes. For example, the following piece of code *is* allowed.

```
fn function<'a>(v : &'a Vec<i32>, w : &'a Vec<i32>) -> &'a Vec<i32>{
    if v[0] == 37 {
        v
    }
    else {
    w
    }
}

let vector = vec![37, 42];
{
    let another_vector = vec![42, 37];
    let f = function(&vector, &another_vector);
    println!("f[0] + f[1] = {}", f[0] + f[1]);
}
```

The curly brackets indicate a scope here. The code will compile. The result of `function` does not live longer than the input variables of `function`, so nothing can go wrong. However, if we shuffle the order in which variables are declared, we get a problem. This is shown in the following example.

```rust
fn function<'a>(v : &'a Vec<i32>, w : &'a Vec<i32>) -> &'a Vec<i32>{
    if v[0] == 37 {
        v
    }
    else {
        w
    }
}

let vector = vec![37, 42];
let f;
{
    let another_vector = vec![42, 37];
    f = function(&vector, &another_vector);
}
println!("f[0] + f[1] = {}", f[0] + f[1]);
```

We now declare `f` before the vector `another_vector` is declared. This means `f` will live longer than `another_vector`. If `function` now returns the resource of `another_vector`, `f` lives longer than the variable it borrowed from. As said, before, this is a problem, because the original owner can no longer clean up after it is returned ownership. The program gives the compile time error `error: ‘another_vector‘ does not live long enough`. We, as humans, can see that there will be no problem if you do run the code, since `function` will return the resource of `vector` and not that of `another_vector`. The Rust compiler, however, still does not allow it.

2.3.1 Lifetime elision

As mentioned above, it is not necessary to explicitly state lifetimes in all cases. The compiler can sometimes infer the lifetimes. Not explicitly stating lifetimes is called **elision**. At the time of writing, there are only very few specific cases in which the compiler can infer the lifetimes. It is certainly possible that the amount of cases in which this can be done grows as newer versions of the language are made available. There was no elision in earlier versions of the language.

The compiler has a set of rules it follows to determine what the lifetimes in a function signature should be. If, after following these rules, the compiler still has references for which it cannot figure out the lifetimes, there will be an error.

These rules are the following (as taken from the website ”The Rust Programming Language” (The Rust Project, 2017b)):

1. Each parameter that is a reference gets its own lifetime parameter. In other words, a function with one parameter gets one lifetime parameter: `fn foo<'a>(x: &'a i32)`, a function with two arguments gets two separate lifetime parameters: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`, and so on.
2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: `fn foo<'a>(x: &'a i32) -> &'a i32`.
3. If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method[2], then the lifetime of `self` is assigned to all output lifetime parameters.

All of the cases in the first two parts of this chapter had only one input lifetime parameter and no return lifetime parameter. This means that, following the rules, no lifetime parameters are left and the compiler has inferred the right lifetimes.

2.3.2 Lifetimes of statements

In this section we have only looked at lifetimes of variables that are passed to functions and how they relate. However, the compiler also keeps track of the lifetimes of variables, even when we use no functions. In most of this thesis, we will be looking at declarations only, so it is worthwhile to look at how the lifetimes work there.

The example is taken from The Rustonomicon which is a piece of documentation about advanced and unsafe Rust programming (The Rust Project, 2017a).

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

We are now going to ‘desugar’ this piece of code and write out all lifetimes explicitly. Note that the following is not valid syntax in Rust. We only use it to see what is going on here. Also note that the compiler will try to minimize the extent of a lifetime. That means desugaring will result in the following:

```rust
// NOTE: `'a: {` and `&'b x` is not valid syntax!
'a: {
    let x: i32 = 0;
    'b: {
```

[2]In Rust, methods are functions attached to objects

```
        // lifetime used is 'b because that's good enough.
        let y: &'b i32 = &'b x; // this is only valid if x lives
                                // at least as long as y,
                                // which is clearly the case
        'c: {
            // in the following line, it is stated that y has to
            // live at least as long as z, which is clearly
            // the case.
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

We can do the same for a small program where we pass references to an outer scope:

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

In this example, we first declare `z` before we declare `y`. After declaring `y` we finally assign a value to `z`. This example desugars to:

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // Must use 'b here because this reference is
            // being passed to that scope.
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

For our last example program, we alias a mutable reference. That means the rules for borrowing are relevant.

```
let mut vector = vec![1, 2, 3];
let x = &vector[0];
vector.push(4);
println!("{}", x);
```

This program has a *mutable* vector `vector` and a variable `x` that points to a part of `vector`. This means we have a problem: `vector` is mutable but `x` also points to the same piece of memory. That means the second rule of borrowing is broken and the program is rejected.

The compiler will reject the program, but for completely different reasons. It does not know that `x` is a reference to a part of `vector`, because it does not know what the `vec` macro does. We desugar the program to see what is happening.

```rust
'a: {
    let mut vector: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index<'b>(&'b vector, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push<'c>(&'c mut vector, 4);
        }
        println!("{}", x);
    }
}
```

The compiler does know that `x` has to live as long as `'b` to be printed. The signature of `Index::index` says that `vector` has to live as long as `'b` as well. When `push` is then called, Rust sees that we try to make a mutable reference to `vector` with lifetime `'c`. Rust knows that `'c` is contained within `'b`, so `&'b data` must still be live. Therefore, when trying to do `Vec::push`, the program is rejected.

Chapter 3

Moving

3.1 Introduction

In the following two chapters, we will look into two different subsets of Rust. This first chapter is concerned with a very simple language where we will just focus on *ownership* and *moving*. We will first look into the syntax of this small language and then move on to the semantics.

In the next chapter, we will add some interesting features, and again look at both the syntax and semantics of those features. This first chapter will go into detail at every step, while the following chapter will not do so anymore and assume that what was mentioned here is known to the reader.

The subset of Rust that interests us in this chapter will be dubbed as 'move only'. In this subset of Rust, you can only assign variables. If you do so, the ownership of the resource will be moved to that variable. You cannot borrow and nothing is mutable. That means everything is constant. While this does not seem like a very useful language, it will provide a good basis to work from when adding new features.

3.1.1 A note on types

In this chapter and the following chapter, we will assume that all programs are typed correctly. We will not develop a type system here, as that is not particularly interesting for what we are investigating now: the compile timed check of moving, borrowing and mutability. Therefore, the assumption is that we are only dealing with well-typed programs.

3.2 Syntax

We will first look at the syntax of our 'move only' language. We have the following defintions.

Definition 3.2.1. A statement S is defined recursively by:

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

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

Definition 3.2.2. An expression e is defined recursively by:

$$e ::= x \mid i \mid e_1 + e_2$$

Definition 3.2.3. A type τ is

$$\tau ::= \texttt{Int}$$

This syntax is very simple, but enough for us to say something useful about ownership. We use only one data type, as more data types will add no additional interesting facts for ownership. However, this syntax can be expanded to include more types. The data type we use is Int to denote all integers.

Our syntax for `let`-statements is different from the usual Rust syntax. We split up a statement such as `let x = 5` in `let x in (x = 5)` (brackets added for clarity). This can be done for every type of let-statement. It is done to show that there are actually two steps in a statement such as `let x = 5`. First of all, the variable `x` is declared. Then, a value is assigned to the resource `x` owns.

Besides splitting up a `let`-statement if necessary, we include a type in our `let`-statements. While this is not done in Rust, it makes things more clear for the reader, especially in the following chapter where we will include borrows.

In order to illustrate this, the following program (taken from 2.3.2) is written in our syntax.

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

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

We will often put the closing bracket at the same indenting height as the opening brackets, contrary to normal programming standards. This is done to clearly show the scopes.

Our syntax looks like the desugaring in 2.3.2, but we removed the lifetimes in order to simplify the syntax. We do not need the lifetimes for the very simple features of Rust we will be looking at.

3.3 Semantics: Framework

In this chapter, we will look at two types of semantics. The first is a big step semantics and the second a small steps semantics. Both are based on the exposition in Nielson and Nielson (1992). We will try to use as much of the same notation as used in Nielson and Nielson (1992) as possible.

Our big step semantics is called 'natural semantics'. Its rules are of the form:

$$\langle S, s\rangle \rightarrow s'$$

S is a statement as defined in the previous section. s and s' are states, for which an exact definition will be given shortly.

Our small step semantics is called 'structural operational semantics'. Its rules are of the form:

$$\langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$$

Where S, S' and s, s' are again statements and states respectively. L and L' are lists of program parts, which will also be defined later on.

In order to work with these rules, we need some more mathematical definitions for the different kinds of sets and functions we are interested in. Therefore, we will first set up a mathematical framework in this section, and then move on to the actual rules in the following two sections.

In this section we will need to distinguish actual definitions from pseudo-definitions. The latter means we give an intended interpretation for a symbol or a set in natural language. This is done to show what the symbol or set is supposed to represent in the real world. However, the actual meaning of the symbol or set will depend on how we will use it in other definitions later on. The pseudo-definitions are added to help the reader navigate through the many difficult notations introduced in the chapter and to give them some intuition to what is happening. To distinguish between actual definitions and these pseudo-definitions, the latter is preceded by 'Informal', as below.

Variable values

The symbols in this subsection will be defined informally.

Informal definition 3.3.1. We will need the following symbols

i) $-$ will be used to indicate that a variable has not been declared at all.
ii) $\perp$ will be used to indicate that a variable has been declared using `let`, but has not been assigned a value.

Intuitively, these are some special kind of values a variable can have.

As is conventional, the set $\mathbb{Z}$ will be used to denote the set of integers. We extend $\mathbb{Z}$ with two new symbols mentioned in the previous section, to get $\mathbb{Z}_{ext}$. Therefore:

Definition 3.3.1. $\mathbb{Z}_{ext} := \mathbb{Z} \cup \{\perp, -\}$.

Expressions

Definition 3.3.2. We have the following sets:

i) **Var** denotes the set of all variables in Rust.
ii) **Num** denotes the set of all numbers in the form of how they are represented in Rust.
iii) **Add** denotes the set of all tuples from **Exp**, i.e. $\mathbf{Add} = \mathbf{Exp} \times \mathbf{Exp}$.
iv) $\mathbf{Exp} = \mathbf{Num} \cup \mathbf{Var} \cup \mathbf{Add}$

Notice that we have a recursive definition of **Exp**. **Exp** should be interpreted as a representation for all possible syntactic expressions. Note that elements from **Add** will usually be denoted as "$e_1 + e_2$" instead of "(e_1, e_2)".

As **Exp** is a inductive set, it will turn out to be useful to also be able to gather all the variables that are in an expression e. We do this by defining a new function $\mathcal{V}$.

Definition 3.3.3. We define the function $\mathcal{V} : \mathbf{Exp} \to \mathcal{P}(\mathbf{Var})$ recursively by:

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

We see that this definition recursively walks through the expression to gather all variables mentioned and join them together into a single set.

We also need a way to transform a Rust representation of a number into an actual number.

Informal definition 3.3.2. $\mathcal{N}$: $\mathbf{Num} \to \mathbb{Z}$ translates a Rust representation for a number to the actual number

We leave the details of this function out, as it is cumbersome and not relevant for the remainder of this thesis. We will simply assume that we have such a function and that it behaves as we would expect. For a detailed exposition on how one could make such a function, we refer the reader to Nielson and Nielson (1992).

State and evaluation

The state is a function that maps a variable to a value. Therefore, we can define the set **State** which contains all possible states.

Definition 3.3.4. **State** is the set of all functions $s : \mathbf{Var} \to \mathbb{Z}_{ext}$

We also have an evaluation function $\mathcal{A}$, which takes an element from **Exp** and a state s and returns the corresponding value. The value is an element of $\mathbb{Z}_{ext}$.

Definition 3.3.5. The evaluation function $\mathcal{A} : \textbf{Exp} \times \textbf{State} \to \mathbb{Z}_{ext}$ is defined by:

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

We now have sufficient apparatus to continue with the actual semantic rules.

3.4 Semantics: Big step

As mentioned earlier on, our big step semantics are called natural semantics with rules of the form

$$\langle S, s\rangle \to s'$$

Where S is a statement and s, s' are states. As we have defined both statements and states in the previous section, we can now start defining the actual rules.

3.4.1 Natural semantics rules

Definition 3.4.1. We define the following natural semantics rules. The names on the left are used to later refer to the specific rules.

$$[\text{skip}_{\text{ns}}] \qquad \langle \texttt{skip}, s\rangle \to s$$

$$[\text{comp}_{\text{ns}}] \qquad \frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

$$[\text{let}_{\text{ns}}] \qquad \frac{\langle S, s[x \mapsto \perp]\rangle \to s'}{\langle \texttt{let } x : \tau \texttt{ in } S, s\rangle \to s'[x \mapsto s(x)]}$$

$$[\text{ass}_{\text{ns}}] \qquad \langle x = e, s\rangle \to s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -] \qquad \text{if } \mathcal{A}[\![x]\!]s = \perp,\ \mathcal{A}[\![e]\!]s \neq \perp \text{ and } \mathcal{A}[\![e]\!]s \neq -$$

$\mathcal{V}(e) \mapsto -$ is an abbreviation for 'for all $x \in \mathcal{V}(e)$, $x \mapsto -$'.

The rules for skip and composition are assumed to be self-explanatory (see also Nielson and Nielson (1992)). The rule for `let` is derived from the idea that a `let`-statement should put the variable x at $\perp$ (declared but not assigned). When the body of a let-statement is finished, x should be put at whatever it was before it encountered the `let`-statement.

As for the assignment rule, there are several conditions that need to apply before we can consider the rule. If one of these conditions does not apply, we cannot apply this rule and that means we cannot make a derivation of the program. The program is incorrect, then. The first rule states that `x` must have been declared, but not assigned. The second and third rules state that the expression must result in a value. In practice that means that every variable in the expression must be assigned some value. For example, take a look at the following program:

$$\texttt{let } x : \tau \texttt{ in } x = y$$

y has not been given a value (it has not even been declared), so we cannot rightly assign y's value to x.

Note that these rules are important to model Rust in the best possible way. It is not possible to assign to a value that has not been declared and it is also not possible to assign a value to a non-mutable variable that already has a value.

3.4.2 Properties of our natural semantics rules

In this subsection, we will look at, and prove some interesting properties of our natural semantics.

Determinism

The first property that interests us is determinism. This means that if we run a program multiple times (even infinitely many times) with the same starting state, the end state will always be the same. This is, of course, a desirable property in most programming languages.

Theorem 3.4.1. *For every statement S, every state s, s', s'', if $\langle S, s\rangle \to s'$ and $\langle S, s\rangle \to s''$, then $s' = s''$.*

Proof. The proof proceeds by induction on the structure of S. We distinguish the following cases:

- $S = \texttt{skip}$: if $\langle \texttt{skip}, s\rangle \to s'$ and $\langle \texttt{skip}, s\rangle \to s''$, then this can only be achieved by [skip$_{\text{ns}}$], which means $s = s'$ and $s = s''$, so $s' = s''$.
- $S = x = e$: if $\langle x = e, s\rangle \to s'$ and $\langle x = e, s\rangle \to s''$, then the rule that must have been applied is [ass$_{\text{ns}}$]. This means $s' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$ and $s'' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$, so $s' = s''$.
- $S = S_1; S_2$: if $\langle S_1; S_2, s\rangle \to s'$ and $\langle S_1; S_2, s\rangle \to s''$, the rule that must have been applied is [comp$_{\text{ns}}$]. This gives us

$$\frac{\langle S_1, s\rangle \to s''' \qquad \langle S_2, s'''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'} \text{ and } \frac{\langle S_1, s\rangle \to s'''' \qquad \langle S_2, s''''\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

  We can now apply the induction hypothesis to $\langle S_1, s\rangle \to s'''$ and $\langle S_1, s\rangle \to s''''$, so we have $s''' = s''''$. We can also apply the induction hypothesis to $\langle S_2, s'''\rangle \to s'$ and $\langle S_2, s''''\rangle \to s''$, so we have $s' = s''$.
- $S = \texttt{let } x : \tau \texttt{ in } S'$: if $\langle \texttt{let } x : \tau \texttt{ in } S', s\rangle \to s'$ and $\langle \texttt{let } x : \tau \texttt{ in } S', s\rangle \to s''$, then it must be the case that the rule [let$_{\text{ns}}$] has been applied. That gives us

$$\frac{\langle S', s[x \mapsto \perp]\rangle \to s'''}{\langle \texttt{let } x : \tau \texttt{ in } S', s\rangle \to s'''[x \mapsto s(x)]}$$

where $s'''[x \mapsto s(x)] = s'$ and

$$\frac{\langle S', s[x \mapsto \perp]\rangle \to s''''}{\langle \texttt{let } x : \tau \texttt{ in } S', s\rangle \to s''''[x \mapsto s(x)]}$$

where $s''''[x \mapsto s(x)] = s''$. We can apply the induction hypothesis to $\langle S', s[x \mapsto \perp]\rangle \to s'''$ and $\langle S', s[x \mapsto \perp]\rangle \to s''''$, so we have $s''' = s''''$. This must also mean that $s' = s'''[x \mapsto s(x)] = s''''[x \mapsto s(x)] = s''$, so $s' = s''$.

This exhausts all possible cases and hence proves our theorem. $\diamondsuit$

Variable allocation

While this first theorem is one that is generally considered useful to prove, the next is very specific to Rust and the way we defined our semantics. It states that a program no longer has any variables in memory after it terminates.

Theorem 3.4.2. *For every statement S, every state s, s', if $\langle S, s\rangle \to s'$, then for all variables y, if $\mathcal{A}[\![y]\!]s = -$ then $\mathcal{A}[\![y]\!]s' = -$.*

Proof. The proof proceeds by induction on the structure of S. We distinguish the following cases:

- $S = \texttt{skip}$: if $\langle \texttt{skip}, s\rangle \to s'$, then this can only be achieved by [skip$_{\text{ns}}$], which means $s = s'$. Therefore if $\mathcal{A}[\![y]\!]s = -$ then $\mathcal{A}[\![y]\!]s' = -$.
- $S = x = e$: assume $\langle x = e, s\rangle \to s'$. Then the only rule that could have been applied is [ass$_{\text{ns}}$], so s' must be $s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$ and $\mathcal{A}[\![x]\!]s = \perp$, $\mathcal{A}[\![e]\!]s \neq \perp$ and $\mathcal{A}[\![e]\!]s \neq -$. If one of these things had not been the case, we could not have applied any rules, and therefore our assumption '$\langle S, s\rangle \to s'$' would be false. So we can assume these things to be true. Now we look at a variable y, chosen arbitrarily. We distinguish two cases:
  - $y = x$: if $\mathcal{A}[\![x]\!]s \neq -$ there is nothing to prove. So assume $\mathcal{A}[\![x]\!]s = -$. However, this is contradictory to our assumption $\mathcal{A}[\![x]\!]s = \perp$. So this is not possible.
  - $y \neq x$: we again assume $\mathcal{A}[\![y]\!]s = -$, as otherwise there is nothing to prove. Then we can again distinguish two cases:
    * $y \in \mathcal{V}(e)$ [1] : in s', we have $\mathcal{V}(e) \mapsto -$, so $y \mapsto -$, which means $\mathcal{A}[\![y]\!]s = -$.
    * $y \notin \mathcal{V}(e)$: as $x \neq y$ and $y \notin \mathcal{V}(e)$, the value of y is not any different from s in $s' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$, so $\mathcal{A}[\![y]\!]s' = -$

  This exhausts all possible cases and therefore proves the case where S is of the form $x = e$.

[1]Note: in practice, it will not be possible for y to be both an element of $\mathcal{V}(e)$ and $\mathcal{A}[\![y]\!]s = -$, as one of the assumptions of the rule [ass$_{\text{ns}}$] assumes $\mathcal{A}[\![e]\!]s \neq -$. However, we would need to prove this with induction on the shape of an expression and it turns out not to be necessary as the proof is rather simple anyways.

- $S = \texttt{let } x : \tau \texttt{ in } S$: assume $\langle \texttt{let } x : \tau \texttt{ in } S, s\rangle \to s'$. Then the rule that has been aplied last must be [let$_{\text{ns}}$]. This has the following derivation tree
$$\frac{\langle S, s[x \mapsto \perp]\rangle \to s''}{\langle \texttt{let } x : \tau \texttt{ in } S, s\rangle \to s''[x \mapsto s(x)]}$$
This means $s' = s''[x \mapsto s(x)]$. We look at an arbitrary variable y and assume $\mathcal{A}[\![y]\!]s' = -$ (otherwise we have nothing to prove). We again distinguish two cases:
  - $y = x$: as $\mathcal{A}[\![x]\!]s = -$ and x in s'' maps to $s(x)$, we can conclude $\mathcal{A}[\![x]\!]s'' = -$
  - $y \neq x$: we are now interested in $\mathcal{A}[\![y]\!]s''$. We can apply the induction hypothesis to $\langle S, s[x \mapsto \perp]\rangle \to s''$ as $\mathcal{A}[\![y]\!]s[x \mapsto \perp] = \mathcal{A}[\![y]\!]s = -$. So $\mathcal{A}[\![y]\!]s'' = -$. As $s' = s''[x \mapsto s(x)]$, the value of y is not changed and we can conclude $\mathcal{A}[\![y]\!]s' = -$
- $S = S_1;\, S_2$: assume $\langle S_1; S_2, s\rangle \to s'$ (NB: the accents $'$ are different than in the definition of the natural semantics). Then the last rule that has been applied must be [comp$_{\text{ns}}$], which gives the following derivation tree
$$\frac{\langle S_1, s\rangle \to s'' \qquad \langle S_2, s''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'}$$
Now assume y to be an arbitrary variable and $\mathcal{A}[\![y]\!]s = -$. We can apply the induction hypothesis to $\langle S_1, s\rangle \to s''$, so we know $\mathcal{A}[\![y]\!]s'' = -$. We can then again apply the induction hypothesis to $\langle S_2, s''\rangle \to s'$. So we know $\mathcal{A}[\![y]\!]s' = -$, which was what was to be proven.

This exhausts all possible shapes of our statement S, and hence proves the theorem. $\diamondsuit$

Note that this means that after a program terminates, none of the results are available anymore. To make use of the data, you would have to write it to a file or some other IO device. This is a significant break from the exposition of Nielson and Nielson (1992) where the state of a terminated program shows the values of the variables at the end and you can thus 'access' the results of the calculations. However, this is not how a computer normally works: after a program terminates, all memory will have to be freed.

3.4.3 Conclusion

This concludes our section of the big step semantics. As these semantics are rather coarse-grained, we will shift our focus to the small step semantics from now on. In the end, we will prove that the two are equivalent.

3.5 Semantics: Small step

We now move on to a small step semantics. We mostly use the same mathematical framework as in the big step semantics, unless stated otherwise. Our small step semantics, as stated above, has rules of the form $\langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$, where S, S' are statements and s, s' are states as defined above. To properly define what L and L' are, we first need to define what is meant by 'a program instruction'.

Definition 3.5.1. A program instruction I has the following form:

$$I ::= S \mid (x, v)$$

where S is a statement, x a variable from **Var** and v an element from $\mathbb{Z}_{ext}$.

Informally, a program instruction is either a statement, or some specific instruction that cannot be programmed by a user, but will be used by us to do some bookkeeping along the way. In the remainder of the thesis, we will mostly use S as a letter for both, but will explicitly state whether that is a program instruction or statement.

We can now look at our *lists*:

Definition 3.5.2. A *list* L is of the form

- $L = \texttt{Nil}$
- $L = I :: L'$, where I is a program instruction and L' is again a list.

We can now state that our L and L' from the operational semantic rules should be lists as defined in definition 3.5.2. We will use this list as a sort of stack to keep track of what program parts need to be executed later on. This idea is not new. It has also been proposed by, for example, Appel and Blazy (2007).

3.5.1 Operational semantics rules

Definition 3.5.3. We define the following semantic derivation rules (name on the left):

[load$_{\text{sos}}$] $\langle \texttt{skip}, I :: L, s\rangle \Rightarrow \langle I, L, s\rangle$

[comp$_{\text{sos}}$] $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$

[ass$_{\text{sos}}$] $\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$

[let$_{\text{sos}}$] $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle \Rightarrow \langle S, (x, s(x)) :: L, s[x \mapsto \perp]\rangle$

[set$_{\text{sos}}$] $\langle (x, v), L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto v]\rangle$

Where $\mathcal{V}(e) \mapsto -$ is an abbreviation for 'for all $x \in \mathcal{V}(e)$, $x \mapsto -$'.

Note that we cannot take a step anymore when in $\langle \texttt{skip}, \texttt{Nil}, s\rangle$; this is the end state.

The [load$_{\text{sos}}$] rule is used to pop an instruction from the stack. The [comp$_{\text{sos}}$] rule pushes the second part of the composition to the list. The idea is that you can then freely execute the first part and pop the second part when the first part finishes. The [ass$_{\text{sos}}$] rule changes the state in a similar way as was done in the natural semantics. The [let$_{\text{sos}}$] rule changes the state to reflect that x now has been declared. It also pushes a tuple onto the list stack. After the body of the `let` terminates, x should be reset to what it was before the `let`-statement. Pushing this tuple to the stack is a way to make that happen. When that tuple is popped from the list, the [set$_{\text{sos}}$] rule resets the value in the state.

Informal definition 3.5.1. We will be using $\langle S, L, s\rangle \Rightarrow^* \langle S', L', s'\rangle$ to indicate we can go from $\langle S, L, s\rangle$ to $\langle S', L', s'\rangle$ in zero or more steps, as defined in 3.5.3.

3.5.2 Properties of our semantic rules

Concerning lists and recursive statements

In this section, we will prove some properties about the lists in combination with the semantics. For example, we will see that it is irrelevant what is in the list for a statement that is currently executing. In order to prove these properties, we need some helping lemmas. These lemmas help us break down the recursive statements: the composition and the `let`-statement. Proving this now will save us some work later on, also in the following sections.

The first of these lemmas states that if we have a derivation from a composition to a `skip`, we must have first pushed the second part to the list and then have executed the first part. Then you pop the second part from the list and execute that.

Lemma 3.5.1. *For every statement S_1, S_2, for every list L, for every state s, s', if $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$, then it must be the case that $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ for some state s''.*

Proof. Assume $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$. Now look at $\langle S_1; S_2, L, s\rangle$. We know there must be a derivation. The only possible rule is [comp$_{\text{sos}}$], so we have $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$. Now we see that our list has grown by one item. However we know there is a derivation and it ends with the list L, instead of $S_2 :: L$. So at some point in the derivation this should be removed from the list. The only possible rule that could have done that is [load$_{\text{sos}}$]. So we must have $\langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle$ at some point in the derivation. Therefore we have $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$, which was to be proven. ♢

We have a similar lemma to the `let`-statement. We will not be proving that one as rigorously, as the proof is very similar to the one of the previous lemma.

Lemma 3.5.2. *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 3.5.1. Note that we can actually say some more about the relation between s'' and s' because of how the rule [set$_{\text{sos}}$]works. $\diamondsuit$

We are now ready to prove that when executing a program instruction, it is irrelevant what is in the list.

Proposition 3.5.3. *For every program instruction S, every state s, s', if $\langle S, L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$ for some list L', then we have $\langle S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ for all lists L.*

Proof. We know that there must be a derivation from $\langle S, L', s\rangle$ to $\langle \texttt{skip}, L', s'\rangle$, so S must be of one of the forms from definition 3.5.3. We therefore prove the proposition by induction on the form of the program instruction S.

- $S = \texttt{skip}$: we look at $\langle \texttt{skip}, L, s\rangle$. This is already of the correct form, as $s = s'$, so we can evaluate to the desired state in zero steps, which proves this case.
- $S = x = e$: assume $\langle x = e, L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$. Then we know the rule that must have been applied first is [ass$_{\text{sos}}$], so we have

$$\langle x = e, L', s\rangle \Rightarrow \langle \texttt{skip}, L', s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$$

  which is of the desired form. So it must be that $s' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$. Now we look at $\langle x = e, L, s\rangle$. The only rule we can apply is [ass$_{\text{sos}}$], so we get

$$\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$$

  which is of the desired form, and has the right s'. This proves this case.
- $S = (v, e)$: assume $\langle (v, e), L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$. Then we know the rule that must have been applied first was [set$_{\text{sos}}$]. So we get

$$\langle (v, e), L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s[x \mapsto v]\rangle$$

  This is of the desired form, so it must be the case that $s' = s[x \mapsto v]$. Now we look at $\langle (v, e), L, s\rangle$. The only rule that can be applied is [set$_{\text{sos}}$], so we get

$$\langle (v, e), L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto v]\rangle$$

  which is of the desired form, and has the right s'. This proves this case.
- $S = S_1; S_2$: assume $\langle S_1; S_2, L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$. Then by [comp$_{\text{sos}}$]and lemma 3.5.1, we have

$$\langle S_1; S_2, L', s\rangle \Rightarrow \langle S_1, S_2 :: L', s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L', s''\rangle$$

$$\Rightarrow \langle S_2, L', s''\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle \ (*)$$

  Now we look at $\langle S_1; S_2, L, s\rangle$ with L arbitrary. We can apply the rule [comp$_{\text{sos}}$]to see

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$$

Now we can apply the induction hypothesis to $\langle S_1, S_2 :: L, s\rangle$, as we have $\langle S_1, L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s''\rangle$ by $(*)$. So we get

$$\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle$$

Then by [load$_{\text{sos}}$],

$$\langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle$$

Now we can apply the induction hypothesis to $\langle S_2, L, s''\rangle$, as we have $\langle S_2, L', s''\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$ by $(*)$. We get

$$\langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

That means we can form the derivation sequence

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle$$

$$\Rightarrow \langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

This gives us

$$\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

which was to be proven.

- $S = \texttt{let}\ x : \tau\ \texttt{in}\ S'$: assume $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S', L', s\rangle \Rightarrow^* \langle \texttt{skip}, L', s'\rangle$. There is only one rule that could have been applied first, [let$_{\text{sos}}$], which gives us

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

By lemma 3.5.2 we get

$$\langle \texttt{let}\ x : \tau\ \texttt{in}\ S', L', s\rangle \Rightarrow \langle S, (x, s(x)) :: L', s[x \mapsto v]\rangle$$

$$\Rightarrow^* \langle \texttt{skip}, (x, s(x)) :: L', s''\rangle \Rightarrow \langle (x, s(x)), L', s''\rangle \Rightarrow \langle \texttt{skip}, L', s'\rangle\ (*)$$

From this and the rule [set$_{\text{sos}}$], we can conclude that $s' = s''[x \mapsto v]$. Now we look at $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S', L, s\rangle$ for some list L. Then we can apply [let$_{\text{sos}}$]to get

$$\langle \texttt{let}\ x : \tau\ \texttt{in}\ S', L, s\rangle \Rightarrow \langle S, (x, s(x)) :: L, s[x \mapsto v]\rangle$$

We can now apply the induction hypothesis to $\langle S, (x, s(x)) :: L, s[x \mapsto v]\rangle$, as we know $\langle S, (x, s(x)) :: L', s[x \mapsto v]\rangle \Rightarrow^* \langle \texttt{skip}, (x, s(x)) :: L', s''\rangle$ by $(*)$. We get

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

By the rules [load$_{\text{sos}}$] and [set$_{\text{sos}}$], we get

$$\langle \texttt{skip}, (x, s(x)) :: L, s''\rangle \Rightarrow \langle (x, s(x)), L, s''\rangle \Rightarrow \langle \texttt{skip}, L, s''[x \mapsto v]\rangle$$

This is of the desired form and has the right s'. This proves this case.

This exhausts all possible cases and hence proves the theorem. $\diamondsuit$

Intuitively, this makes sense. If we look at the rules, we see that we can only move something from the list to the front if the front only contains a `skip` statement, and that thus everything that had been there, must have finished evaluating.

We will now do a simple proof to show that in a composition, the execution of S_1 does not depend on what S_2 entails.

Proposition 3.5.4. *For every statement S_1, S_2, for every list L, for every state s, s', if $\langle S_1, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s\rangle$, then $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle S_2, L, s\rangle$*

Proof. Assume $\langle S_1, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s\rangle$ and look at $\langle S_1; S_2, L, s\rangle$. Then by [comp$_{\text{sos}}$]

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$$

Now by 3.5.3 and the fact that $\langle S_1, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s\rangle$, we know $\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s\rangle$. Then from [load$_{\text{sos}}$]

$$\langle \texttt{skip}, S_2 :: L, s\rangle \Rightarrow \langle S_2, L, s\rangle$$

Therefore, we get

$$\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle S_2, L, s\rangle$$

which was to be proven. $\diamondsuit$

Determinism

Another major desirable property of our semantics is *determinism.* This means that running the program will always have the same results. Therefore, we want our derivation system to always return the same result. The proof is similar to that of the natural semantics.

Theorem 3.5.5. *For every program instruction S, every state s, s', s'', every list L, if $\langle S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $\langle S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$, then $s' = s''$.*

Proof. The proof proceeds by induction on the structure of S, the program instruction. We distinguish the following cases:

- $S = \texttt{skip}$: if $\langle \texttt{skip}, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $\langle \texttt{skip}, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$, then no steps have to be taken, or could have been taken, and we obviously have $s' = s''$.
- $S = x = e$: $\langle x = e, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $\langle x = e, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$, then the rule that must have been applied is [ass$_{\text{sos}}$]. This means that exactly one step has been taken and that $s' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$ and $s'' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$, so $s' = s''$.
- $S = (x, v)$: if $\langle (x, v), L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $\langle (x, v), L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$, then the rule that must have been applied next is [set$_{\text{sos}}$]. This means that exactly one step has been taken and that $s' = s[x \mapsto v]$ and $s'' = s[x \mapsto v]$, so $s' = s''$.

- $S = S_1; S_2$: if $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$, then the only applicable rule is [comp$_{\text{sos}}$]. This means that we have

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

and

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$$

By Proposition 3.5.1, we know that the latter $\Rightarrow^*$ can be broken up into smaller parts, so we have

$$\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s'''\rangle \Rightarrow \langle S_2, L, s'''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

and

$$\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''''\rangle \Rightarrow \langle S_2, L, s''''\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$$

for some s''' and s''''. Now we can apply the induction hypothesis to $\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s'''\rangle$ and $\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''''\rangle$, so we have $s''' = s''''$. We then get

$$\langle S_2, L, s'''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

and

$$\langle S_2, L, s'''\rangle \Rightarrow^* \langle \texttt{skip}, L, s''\rangle$$

to which we again can apply the induction hypothesis to get $s' = s''$.

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

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

and

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

By Proposition 3.5.2, we know that the latter $\Rightarrow^*$ can be broken up into smaller parts, so we have

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

and

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

Now we can apply the induction hypothesis to

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

and

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

That gives us $s''' = s''''$. Then $\langle (x, s(x)), L, s'''\rangle \Rightarrow \langle \texttt{skip}, L, s'''[x \mapsto v]\rangle$, so $s' = s'''[x \mapsto v] = s''$, which means $s' = s''$.

This exhausts all possible cases and hence proves the theorem. $\diamondsuit$

3.5.3 Conclusion

This concludes our section on the small step semantics. We saw that they are similar to the big step semantics, but do not include the specific conditions in the assignment. In order to make sure that is adhered to those conditions, we define a second derivation system, which is the focus of the following section.

3.6 Compile time check

As mentioned before, all checks are done compile time. However, a semantic description is a description of what is supposed to happen run time. In the natural semantics section we combined the two, but to really model Rust properly, it is necessary to separate the two.

In this section, we look at a compile time checker derivation system to model what happens in the Rust compiler. The checker will always result in either `true` or `false` to indicate whether the program is correct according to Rust's rules. This checker will have a lot of similarities to the semantics, but will not perform actual computations.

First, we will introduce some new definitions, then give the derivation system and lastly prove some properties of the system and of the system in combination with the semantics of the previous section.

3.6.1 Definitions

Because we will not be doing actual computations, but instead just bookkeeping of whether something is a value, we do not need a state as in the previous section, but will rather use what we will call a *reduced state*.

Informal definition 3.6.1. $\star$ will be used to indicate that a variable has some numerical value from $\mathbb{Z}$, but we are not interested in what this value is.

Definition 3.6.1. **RState** is the set of functions $r : \mathbf{Var} \to \{-, \bot, \star\}$

The interpretation of $\star$ here is that the variable has a value. However, for compile time checking, it is irrelevant what that value is, so we leave that out. We just want to know that the variable has been assigned a certain value.

However, in proofs further on, it will prove to be desirable to be able to talk about a state and a reduced state being *related*.

Definition 3.6.2. A state $s \in \mathbf{State}$ and a reduced state $r \in \mathbf{RState}$ are said to be *related* if for all $x \in \mathbf{Var}$, we have

$$r(x) = \bot \iff s(x) = \bot$$

and

$$r(x) = - \iff s(x) = -$$

This means $r(x) = \star \iff s(x) \in \mathbb{Z}$.

We are now ready for the actual compile time checker.

Definition 3.6.3. The *compile time checker* is a derivation system that has the following rules

$$\begin{aligned}
[\texttt{skip}, \texttt{Nil}, r] &\to \texttt{true} \\
[\texttt{skip}, P :: L, r] &\to [P, L, r] \\
[S_1; S_2, L, r] &\to [S_1, S_2 :: L, r] \\
[x = e, L, r] &\to [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]] \\
&\quad \text{if } r(x) = \perp \text{ and } \forall y \in \mathcal{V}(e), r(y) = \star \\
&\ \to \texttt{false} \text{ otherwise} \\
[\texttt{let } x : \tau \texttt{ in } S, L, r] &\to [S, (x, r(x)) :: L, r[x \mapsto \perp]] \\
[(x, v), L, r] &\to [\texttt{skip}, L, r[x \mapsto v]]
\end{aligned}$$

3.6.2 Properties

We will now look at some interesting properties of the compile time checker. First of all, we will prove that the checker always finishes, independent on what the program that needs to be checked is. While a program might run forever, the compiler should always finish. In order to prove this, we will assign a (non-negative) ‘length’ to every possible input of the checker and show that with every step the checker takes, this ‘length’ decreases. This in turn gives us a decreasing sequence of non-negative numbers. This sequence must always be finite, since there is no infinite decreasing sequences of non-negative numbers.

Theorem 3.6.1. *The compile checker from definition 3.6.3 always terminates.*

Before we proof this theorem, we first define the aforementioned ‘length’.

Definition 3.6.4. The *length* is a function $|.| : \textbf{Lists} \times \textbf{ProgInstr} \to \mathbb{N}$ which is defined by $|(P, L)| = |P| + |L|$.

The length on a list $|L|$ is recursively defined by

$$\begin{aligned}
|\texttt{Nil}| &= 0 \\
|B :: L| &= |B| + |L|
\end{aligned}$$

Futhermore, for every program instruction we have

$$\begin{aligned}
|\texttt{skip}| &= 1 \\
|S_1, S_2| &= |S_1| + |S_2| + 1 \\
|x = e| &= 2 \\
|\texttt{let } x : \tau \texttt{ in } S| &= |S| + 3 \\
|(x, v)| &= 2
\end{aligned}$$

Lastly, we define that $|\texttt{false}|$ and $|\texttt{true}|$ are 0.

Together this gives us a definition of the length of a program and list combined.

Proof. Of theorem 3.6.1. We will show termination by proving that with every step from 3.6.3, the length as defined in 3.6.4 decreases. We will look at the different possible cases.

- $[\texttt{skip}, \texttt{Nil}, r] \to \texttt{true}$ gives $1 \to 0$
- $[\texttt{skip}, P :: L, r] \to [P, L, r]$ gives $1 + |P| + |L| \to |P| + |L|$
- $[S_1; S_2, L, r] \to [S_1, S_2 :: L, r]$ gives $|S_1| + |S_2| + 1 + |L| \to |S_1| + |S_2 :: L| = |S_1| + |S_2| + |L|$
- $[x = e, L, r] \to [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]]$ gives $2 + |L| \to 1 + |L|$
- $[x = e, L, r] \to \texttt{false}$ gives $2 + |L| \to 0$
- $[\texttt{let}\ x : \tau\ \texttt{in}\ S, L, r] \to [S, (x, r(x)) :: L, r[x \mapsto \perp]]$ gives $|S| + 3 + |L| \to |S| + 2 + |L|$
- $[(x, v), L, r] \to [\texttt{skip}, L, r[x \mapsto v]]$ gives $2 + |L| \to 1 + |L|$

As every one of these steps shows a decrease in the length, we can conclude that the derivation must terminate (by the reasoning explained above). $\diamondsuit$

3.6.3 Equivalence with our small step semantics

Now we have both a big step semantics and a small step semantics, one would hope these two to be equivalent. First of all, one should notice that the big step semantics has the compile check rules embedded in the semantics, while the small step semantics has them separated in the form of the compile time check. That makes that in the proof, we will prove the big step semantics to be equivalent to the small step semantics combined with the compile time checker.

In order to prove the equivalence, we will first prove that one implies the other, and then prove that the other implies the first.

Lemma 3.6.2. *For all statements S, S', all states s, s', reduced states r, r' with r related to s, r' related to s', for all lists L: if $\langle S, s\rangle \to s'$ then $\langle S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$. Also, if $[\texttt{skip}, L, r'] \to^* \texttt{true}$, then $[S, L, r] \to^* \texttt{true}$.*

Proof. The proof proceeds by induction on the shape of the derivation tree.

- We assume the last rule applied was $[\text{skip}_{\text{ns}}]$. This means $\langle \texttt{skip}, s\rangle \to s$. Then for any list L, we have $\langle \texttt{skip}, L, s\rangle$ leading in zero steps to the required end state.
  Now assume $[\texttt{skip}, L, r] \to \texttt{true}$ with s related to r. This is already what needed to be proven in the second part of the lemma.
- We assume the last rule applied was $[\text{ass}_{\text{ns}}]$. This means $\langle x = e, s\rangle \to s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$. The conditions state that $\mathcal{A}[\![x]\!]s = \perp$, $\mathcal{A}[\![e]\!]s \neq \perp$ and $\mathcal{A}[\![e]\!]s \neq -$. Then we start with $\langle x = e, L, s\rangle$ for any L. Then we can apply $[\text{ass}_{\text{sos}}]$to get

$$\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$$

This is the required end state.
Now assume $[\texttt{skip}, L, r'] \to^* \texttt{true}$ and we know from the first part that $r' = r[x \mapsto \star][\mathcal{V}(e) \mapsto -]$. We know $\mathcal{A}[\![x]\!]s = \perp$, $\mathcal{A}[\![e]\!]s \neq \perp$ and $\mathcal{A}[\![e]\!]s \neq -$. Then we also know that

$$[x = e, L, r] \to [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]] \to^* \texttt{true}$$

as $\mathcal{A}[\![e]\!]s \neq \perp$ and $\mathcal{A}[\![e]\!]s \neq -$ implies that $\forall y \in \mathcal{V}(e), r(y) = \star$.

- We assume the last rule applied was $[\text{comp}_{\text{ns}}]$. This means

$$\frac{\langle S_1, s\rangle \to s'' \qquad \langle S_2, s''\rangle \to s'}{\langle S_1; S_2, s\rangle \to s'}$$

If we now look at $\langle S_1; S_2, L, s\rangle$ for some list L, we see that we can apply $[\text{comp}_{\text{sos}}]$to get

$$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$$

We can now apply the Induction Hypothesis to $\langle S_1, S_2 :: L, s\rangle$ as we have $\langle S_1, s\rangle \to s''$. We get

$$\langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle$$

Then by $[\text{load}_{\text{sos}}]$, we have

$$\langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle$$

Now we can apply the Induction Hypothesis again, as we have $\langle S_2, s''\rangle \to s'$. This gives us

$$\langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

This all combined gives us

$$\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

which was to be proven.
The second part of the lemma is a bit more tricky. Assume $[\texttt{skip}, L, r'] \to^* \texttt{true}$ $(*)$ for some list L in the above derivation. We are interested in the derivation of $[S_1; S_2, L, r]$. We can immediately move a step, so

$$[S_1; S_2, L, r] \to [S_1, S_2 :: L, r]$$

This leaves us with the question on what we can say about $[S_1, S_2 :: L, r]$. It seems logical to apply the same Induction Hypothesis (IH) as was done above the first time (on $\langle S_1, s\rangle \to s''$). This IH tells us that if $[\texttt{skip}, S_2 :: L, r''] \to^* \texttt{true}$, then $[S_1, S_2 :: L, r] \to^* \texttt{true}$. Therefore, in order to be able to apply the IH, we need to say something about $[\texttt{skip}, S_2 :: L, r'']$. By the loading rule, we get

$$[\texttt{skip}, S_2 :: L, r''] \to [S_2, L, r'']$$

This still does not tell us whether it will derive to `true`. However, it seems that we could again use the IH, in the same way it was done above the *second* time (on $\langle S_2, s''\rangle \to s'$). This IH states that if $[\texttt{skip}, L, r'] \to^* \texttt{true}$, then $[S_2, L, r''] \to^* \texttt{true}$. The condition is true per assumption $(*)$. Therefore, we have $[S_2, L, r''] \to^* \texttt{true}$. So that gives us

$$[\texttt{skip}, S_2 :: L, r''] \to [S_2, L, r''] \to^* \texttt{true}$$

and per the first application of the IH. Finally we have

$$[S_1; S_2, L, r] \to [S_1, S_2 :: L, r] \to^* \texttt{true}$$

which was to be proven.

- We assume the last rule applied was $[\text{let}_{\text{ns}}]$. This means

$$\frac{\langle S, s[x \mapsto \perp]\rangle \to s'}{\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, s\rangle \to s'[x \mapsto s(x)]}$$

If we now look at $\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle$ for some list L, we see that we can apply $[\text{let}_{\text{sos}}]$ to get

$$\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle \Rightarrow \langle S, (x, s(x)) :: L, s[x \mapsto \perp]\rangle$$

We can now apply the IH to $\langle S, (x, s(x)) :: L, s[x \mapsto \perp]\rangle$, as we have $\langle S, s[x \mapsto \perp]\rangle \to s'$. This gives us

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

Then per $[\text{load}_{\text{sos}}]$ and $[\text{set}_{\text{sos}}]$ we have

$$\langle \texttt{skip}, (x, s(x)) :: L, s'\rangle \Rightarrow \langle (x, s(x)), L, s'\rangle \Rightarrow \langle \texttt{skip}, L, s'[x \mapsto s(x)]\rangle$$

In total, this gives us

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

which was to be proven.
For the second part of the lemma, we follow a reasoning similar to the previous case. Assume $[\texttt{skip}, L, r'[x \mapsto r(x)] \to^* \texttt{true}\ (*)$ for some list L in the above derivation. We are interested in the derivation of $[\texttt{let}\ x : \tau\ \texttt{in}\ S, L, r]$. We can immediately move a step to get

$$[\texttt{let}\ x : \tau\ \texttt{in}\ S, L, r] \to [S, (x, r(x)) :: L, r[x \mapsto \perp]]$$

It seems sensible to want to apply the IH (on $\langle S, s[x \mapsto \perp]\rangle \to s'$). The IH in this case states that if $[\texttt{skip}, (x, r(x)) :: L, r'] \to^* \texttt{true}$, then $[S, (x, r(x)) :: L, r[x \mapsto \perp]] \to^* \texttt{true}$. Therefore, we are interested in $[\texttt{skip}, (x, r(x)) :: L, r']$. By the derivation rules we have

$$[\texttt{skip}, (x, r(x)) :: L, r'] \to [(x, r(x)), L, r'] \to [\texttt{skip}, L, r'[x \mapsto r(x)]]$$

Of this last one, we know it derives to $\texttt{true}$, by $(*)$. Therefore, we know $[\texttt{skip}, (x, r(x)) :: L, r'] \to^* \texttt{true}$, and we can apply the IH as mentioned above. Therefore, we know $[S, (x, r(x)) :: L, r[x \mapsto \bot]] \to^* \texttt{true}$. This gives us

$$[\texttt{let}\ x : \tau\ \texttt{in}\ S, L, r] \to [S, (x, r(x)) :: L, r[x \mapsto \bot]] \to^* \texttt{true}$$

which was to be proven.

The lemma now follows by induction. ♢

To prove the other way around, we first need two helping lemmas.

Lemma 3.6.3. *For all program instructions S, S', lists L, L', statements s, s': if $\langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$ and $[S, L, r] \to^*$ **true**, then $[S, L, r] \to [S', L', r']$ for exactly one r' related to s'.*

Proof. We will check the different possibilities for the rules $\Rightarrow$.

- $\langle \texttt{skip}, I :: L, s\rangle \Rightarrow \langle I, L, s\rangle$. Assume $[\texttt{skip}, I :: L, r] \to^* \texttt{true}$ with r related to s. We can only do $[\texttt{skip}, I :: L, r] \to [I, L, r]$. Our r is of the right form and the only such r, which proves this case.
- $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$. Assume $[S_1; S_2, L, r] \to^* \texttt{true}$ with r related to s. We can only do $[S_1; S_2, L, r] \to [S_1, S_2 :: L, r]$. Our r is of the right form and the only such r, which proves this case.
- $\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$. Assume $[x = e, L, r] \to^* \texttt{true}$ with r related to s. We can only do $[x = e, L, r] \to [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]]$, because if the conditions of the rule were not met, the compile check could not have terminated in $\texttt{true}$, but would have terminated in $\texttt{false}$. Our r is of the right form, because $\mathcal{A}[\![e]\!]s \in \mathbb{Z}$ and the only such r. This proves this case.
- The other cases are similar and therefore omitted.

This proves our Lemma. ♢

Lemma 3.6.4. *For all program instructions S, S', lists L, L', states s, s': if $\langle S, L, s\rangle \Rightarrow^* \langle S', L', s'\rangle$ and $[S, L, r] \to^*$ **true**, then $[S, L, r] \to^* [S', L', r']$ for exactly one r' related to s'.*

Proof. By induction on the result from 3.6.3. ♢

Lemma 3.6.5. *For all statements S and lists L, for all states s, s', for all reduced states r, with r related to s: if $\langle S, L, s\rangle \Rightarrow^* \langle$**skip**$, L, s'\rangle$ and $[S, L, r] \to^*$ **true**, then $\langle S, s\rangle \to s'$. Note that S is a statement here, and not a program instruction.*

Proof. We do some form of induction. We will look at the possible forms of S.

- $\texttt{skip}$: assume $\langle \texttt{skip}, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $[\texttt{skip}, L, r] \to^* \texttt{true}$. Then s must be equal to s'. Then we have $\langle \texttt{skip}, s\rangle \to s$, which was to be proven.

- $x = e$: assume $\langle x = e, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $[x = e, L, r] \rightarrow^* \texttt{true}$. Then $[\text{ass}_{\text{sos}}]$ must have been applied, so

  $$\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

  This must mean $s' = s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$. We could make $\langle x = e, s\rangle \rightarrow s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$ per $[\text{ass}_{\text{ns}}]$, if we can assure that $\mathcal{A}[\![x]\!]s = \perp$, $\mathcal{A}[\![e]\!]s \neq \perp$ and $\mathcal{A}[\![e]\!]s \neq -$. In $[x = e, L, r] \rightarrow^* \texttt{true}$, we know that

  $$[x = e, L, r] \rightarrow [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]] \rightarrow^* \texttt{true}$$

  and we only could have applied that rule if if $r(x) = \perp$ and $\forall y \in \mathcal{V}(e), r(y) = \star$. If r and s related, this means that the conditions are fulfilled and we can apply $[\text{ass}_{\text{ns}}]$.
- $S_1; S_2$: assume $\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $[S_1; S_2, L, r] \rightarrow^* \texttt{true}$. Then per 3.5.1 we have

  $$\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^* \langle \texttt{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$$

  Now if $[S_1, S_2 :: L, r] \rightarrow^* \texttt{true}$ we can apply the Induction Hypothesis (IH). As we have $[S_1; S_2, L, r] \rightarrow^* \texttt{true}$, we can make

  $$[S_1; S_2, L, r] \rightarrow [S_1, S_2 :: L, r] \rightarrow^* \texttt{true}$$

  so we can indeed apply the IH. This gives us $\langle S_1, s\rangle \rightarrow s''$.
  Now, if also $[S_2, L, r''] \rightarrow^* \texttt{true}$, we can again apply the IH to get $\langle S_2, s''\rangle \rightarrow s'$. We know $[S_1, S_2 :: L, r] \rightarrow^* \texttt{true}$. The last step before $\texttt{true}$ must be $[\texttt{skip}, \texttt{Nil}, r''']$, as that is the only rule that leads to $\texttt{true}$. So we know the list needs to be emptied again. Therefore, the derivation needs to be something like the following:

  $$[S_1; S_2, L, r] \rightarrow [S_1, S_2 :: L, r] \rightarrow^* [\texttt{skip}, S_2 :: L, r''''] \rightarrow [S_2, L, r''''] \rightarrow^*$$

  $$[\texttt{skip}, \texttt{Nil}, r'''] \rightarrow \texttt{true}$$

  Because of 3.6.4 we know that r'''' must be related to s'' and must be equal to r''. This means that we can indeed conclude that $[S_2, L, r''] \rightarrow^* \texttt{true}$ and we therefore have $\langle S_2, s''\rangle \rightarrow s'$.
  Now we can apply $[\text{comp}_{\text{ns}}]$ to give us $\langle S_1; S_2, s\rangle \rightarrow s'$, which was to be proven.
- $\texttt{let}\ x : \tau\ \texttt{in}\ S$: This case is very similar to that of the composition, but with only one application of the Induction Hypothesis. Therefore, we leave out the details.

$\diamondsuit$

Theorem 3.6.6. *For all statements S, S', states s, s', reduced states r: $\langle S, s\rangle \rightarrow s'$ if and only if for some L $\langle S, L, s\rangle \Rightarrow^* \langle \texttt{skip}, L, s'\rangle$ and $[S, L, r] \rightarrow^* \texttt{true}$ (with r related to s).*

Proof. This follows from the proofs for Lemmas 3.6.2 and 3.6.5. $\diamondsuit$

3.6.4 Safety

We now move on to the more interesting proofs where we will actually prove safety properties. Here, we apply an idea originally introduced by Wright and Felleisen (1994) in a slightly different context. The original idea is that when you have a type system, you prove *preservation* and *progress*, and that this combines to safety. However, in our case, we do not have a type system, but we do have our compile time checker. The following pages detail these proofs.

Preservation

The first thing we will prove is *preservation*, where we will show that if the compile time checker says a program is okay, and you take a step in the semantics, the checker will say it is still okay. In general this means that the result of the checker does not depend on where you are in the program and does not change throughout running the program.

Theorem 3.6.7. *For all program instructions S, S', lists L, L', states s, s', reduced states r: if $[S, L, r] \rightarrow^* \texttt{true}$ and $\langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$ with r related to s, then there is an r' that is related to s' and we have $[S', L', r'] \rightarrow^* \texttt{true}$.*

Proof. We assume $[S, L, r] \rightarrow^* \texttt{true}$. We look at the different possible rules $\Rightarrow$.

- $\langle \texttt{skip}, I :: L, s\rangle \Rightarrow \langle I, L, s\rangle$. We assume $[\texttt{skip}, I :: L, r] \rightarrow^* \texttt{true}$. We try to determine which derivation steps could have let to $\texttt{true}$. The only possible step is $[\texttt{skip}, I :: L, r] \rightarrow [I, L, r]$. Then we have our $r' = r$, and obviously we also have $[I, L, r] \rightarrow^* \texttt{true}$.
- $\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$. We assume $[x = e, L, r] \rightarrow^* \texttt{true}$. We try to determine which derivation steps could have led to $\texttt{true}$. The only possible step is $[x = e, L, r] \rightarrow [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]]$. Then we have our $r' = r[x \mapsto \star][\mathcal{V}(e) \mapsto -]$ and obviously we also have $[\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]] \rightarrow^* \texttt{true}$ then.
- $\langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle$. We assume $[S_1; S_2, L, r] \rightarrow^* \texttt{true}$. We try to determine which derivation steps could have led to $\texttt{true}$. The only possible step is $[S_1; S_2, L, r] \rightarrow [S_1, S_2 :: L, r]$. Then we have our $r' = r$ and obviously we also have $[S_1, S_2 :: L, r] \rightarrow^* \texttt{true}$.
- The other cases are similar and therefore omitted.

This proves our theorem. ♢

Progress

The second property we are interested in is called *progress*. This means that if the program passes through the compile time checker, we can always do a step in the semantics (or we are already in the `skip` state).

Theorem 3.6.8. *For all program instructions S, reduced states r, lists L: if $[S, L, r] \rightarrow^* \texttt{true}$, then $S = \texttt{skip}$ and $L = \texttt{Nil}$ or we have, for all s related to r, $\langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$ for some statement S', list L' and state s'.*

Proof. We assume $[S, L, r] \to^* \texttt{true}$. Since it must derive to `true`, we know that derivation steps must be taken. We walk through all the possible derivation steps.

- $[\texttt{skip}, \texttt{Nil}, r] \to^* \texttt{true}$. This means $S = \texttt{skip}$ and $L = \texttt{Nil}$ and that means we are done.
- $[\texttt{skip}, P :: L, r] \to \texttt{true}$. This means we want to say something about $\langle \texttt{skip}, P :: L, s\rangle$ for any s related to r. We can apply the rule [load$_{\text{sos}}$]. This gives us $\langle \texttt{skip}, P :: L, s\rangle \Rightarrow \langle P, L, s\rangle$, which proves this case.
- $[x = e, L, r] \to^* \texttt{true}$. This means we want to say something about $\langle x = e, L, s\rangle$ for any s related to r. We can apply [ass$_{\text{sos}}$]to get $\langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle$, which proves this case.
- The other cases are similar and therefore omitted.

This proves the theorem. $\Diamond$

Chapter 4

Borrowing

4.1 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.

4.2 Syntax

First, we need to update our syntax to reflect the possibility of borrowing. The possible statements from definition 3.2.1 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 4.2.1. 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 3.5.1) 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 4.2.2. An expression e is defined recursively by:

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

Definition 4.2.3. A type τ is

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

4.3 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.

4.3.1 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 $\{$`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.

4.3.2 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 $-$.

4.3.3 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 $\textbf{Var} \to \mathcal{P}(\{\texttt{bor}, \texttt{mut}\} \cup \textbf{Var})$.

```
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 $\perp$):

```
//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.

4.3.4 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.

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

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

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

4.4 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 4.4.1. 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 4.4.2. 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 4.4.3. 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 4.4.4. The evaluation function $\mathcal{A} : \mathbf{Exp} \times \mathbf{State} \to \mathbb{Z}_{ext}$ is defined by:

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

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

4.5 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 4.2.1. 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 4.5.1. We define the following semantic derivation rules (name on the left):

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

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

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

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

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

[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 [load$_{\text{sosb}}$] rule and the [comp$_{\text{sosb}}$] rule are assumed to be self-explanatory. As for the [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 [let$_{\text{sosb}}$] and [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 [letmut$_{\text{sosb}}$] rule, we need $\mathcal{M}$ to reflect that x is mutable.

For the [set$_{\text{sosb}}$] 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 4.5.1. *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 3.5.1 and therefore omitted. ♢

Now we can prove that the semantics are deterministic.

Theorem 4.5.2. *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 [letmut$_{\text{sosb}}$]. This gives us

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

and

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

By Proposition 4.5.1, we know that the latter $\Rightarrow^*$ can be broken up into smaller parts, so we have

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

and

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

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

Now we can apply the induction hypothesis to

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

and

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

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

This proves the theorem. ♢

4.5.1 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 4.5.2. The *compile time checker* is a derivation system that has the following rules

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

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

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

If neither of these conditions hold, we have the rule

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

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 $\perp$ 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 $r(x) =\perp$. The rest is the same as above.

Termination

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

Theorem 4.5.3. *The compile checker from definition 4.5.2 always terminates.*

Proof. This proof is almost completely similar to that of 3.6.1, if we define $|\texttt{let mut } x : \tau \texttt{ in } S| = |S| + 3$. We therefore omit the details. $\diamondsuit$

Progress and preservation

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

Theorem 4.5.4. *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 $\texttt{true}$. 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 $\texttt{true}$. 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. $\diamondsuit$

Theorem 4.5.5. *For all program instructions S, reduced states r, lists L, and for all $\mathcal{M}$'s: if $[S, L, r, \mathcal{M}] \rightarrow^* \texttt{true}$, 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 $[S, L, r, \mathcal{M}] \rightarrow^* \texttt{true}$. We look at the possible derivation steps that could have led to this form.

- $[\texttt{skip}, \texttt{Nil}, r, \mathcal{M}] \rightarrow \texttt{true}$. This means $S = \texttt{skip}$ and $L = \texttt{Nil}$ and that means we are done.
- $[\texttt{skip}, P :: L, r, \mathcal{M}] \rightarrow [P, L, r, \mathcal{M}]$. 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 [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 $\diamondsuit$

Chapter 5

Conclusion and discussion

In the previous chapters, we have seen how ownership, borrowing and mutability work together to provide memory safety and how we write these concepts down in relatively simple semantic rules. We have also proved preservation and progress, to prove safety in a more classical way.

5.1 Discussion and future work

5.1.1 On safety and memory safety: revisited

In the introduction, we shortly talked about safety and memory safety. In the previous chapters, we proven a safety aspect of our semantics: it cannot get stuck if the programs adhere to some condition (our compile time check). We have also looked very briefly at memory safety when we proved that all memory should be cleaned up after a program terminates. Of course, many other properties could be proven. For example, one could prove (in Chapter 4) that every variable is in at most one $\mathcal{M}$ of another variable at all times. More of these properties could be formulated and proved to say more about memory safety. Of course, one should not forget that of several ideas, a more general proof already has been given, such as for a justification of the borrow rules (Boyland, 2003).

5.1.2 Other Rust features

While we have looked at some important concepts of Rust, there is much more to Rust, which we did not look at extensively. In this section, we discuss some of these features and propose some angles for investigation of these features.

While loops

We did not include loops in our very limited language, even though loops have been kept in mind while designing the derivation rules for both the semantics and compile time checker.

It is interesting to add some form of looping, as that makes it possible for a program to not ever finish. As said in the introduction of this thesis, we want the compile time check to always finish, even if the program itself does not finish.

For example, the very simple program

```
fn main() {
    while true {
        print!("Hello world!")
    }
}
```

produces no error.[1]

During the making of the thesis, we briefly looked at how we could add while loops to our syntax and semantics.

Definition 5.1.1. A statement S is defined recursively by:

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

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

For a formal definition of Booleans b, we refer to Nielson and Nielson (1992).

Besides this, we would have semantic rules of the following form.

Definition 5.1.2. In addition to the rules in 4.5.1, we define the following two semantic derivation rules:

$$\langle \texttt{while } b \ \{S'\}, L, s, \mathcal{M}\rangle \Rightarrow \langle S', \texttt{while } b \ \{S'\} :: L, s, \mathcal{M}\rangle$$

if b evaluates to `true`. (named [whilet$_{\text{os}}$])

$$\langle \texttt{while } b \ \{S'\}, L, s, \mathcal{M}\rangle \Rightarrow \langle \texttt{skip}, L, s, \mathcal{M}\rangle$$

if b evaluates to `false`. (named [whilef$_{\text{os}}$])

Of course, the intersting part is how we will deal with the while loop in the compile time check. The Rust compiler simply does not accept any non-mutable variable assignments in a while loop, even if the assignment happens only once or not even at all, as in the following program.

```
let x;
while false {
    x = 0
}
```

[1]It does give the warning `denote infinite loops with ‘loop ... ‘`, but as said in the introduction, we will only be trying to model actual errors.

This program gives the error `re-assignment of immutable variable ‘x‘`.

So we will have to keep track of whether we are in a while loop or not. We can do that, for example, by adding an extra bit of information to our derivation system, so that our derivation system has rules of the form

$$[S, L, s, \mathcal{M}, c] \to [S', L', s', \mathcal{M}', c']$$

The c is a sort of counter that registers the depth of the nested loops.

We will not provide a complete definition for the whole checker, but will sketch what this definition will have to keep track off. First of all, we need to make sure that the compile time checker does not actually run through the while loop, as this might result in an infinite loop, while the checker itself should always finish. This calls for a rule of form similar to

$$[\texttt{while}\ b\ \{S\}, L, r, \mathcal{M}, c] \to [S, \texttt{dec} :: L, r, \mathcal{M}, c + 1]$$

The `dec` in this rule refers to decreasing the counter after we have walked through the body of the loop.

It is important to note that there should be no immutable assignment within a loop, as that would make it possible for an assignment to happen multiple times. As we saw before, the compile checker of the actual Rust Programming Language does not keep track of whether an assignment actually will happen multiple times, so neither do we. This simplifies the process.

As mentioned before, we add a counter to the tuple that will be incremented once we enter a while loop and decremented once we exit the while loop. If the compile time checker comes accross an immutable assignment when the counter is strictly larger than zero, it should reject the program.

The compile time checker pushes a decrease operation to our stack/list to make sure that after the while loop finishes, the counter will be reduced by one.

We need a counter and cannot simply do with for example a boolean, as there might be multiple nested while loops. After the inner while loop, you might naively put the boolean to false when you finish, but then this would not be correct as you are still in the outer while loops. You could make an extra program instruction with the value of the boolean when you enter the loop to keep track of what it was, but this is extra bookkeeping and more difficult than our proposed solution.

The progress and preservation proofs get more complicated when we add while loops to our language. This is because our semantic rules no longer match as perfect with our compile time check anymore as in the previous chapters. This will mean we will have to do some extra work. Also the formulation of the theorems sometimes has to be changed, as there are program instruction types in the compile time checker that do not exist in the semantics.

Lifetimes

Due to time constraints, we did not really include lifetimes in this thesis. We mostly used scope properties instead of the lifetimes themselves. A more extensive investigation could look into how to incorporate lifetimes in the current semantics. This would include choosing a suitable mathematical representation for lifetimes.

A possible choice would be to make lifetimes a lattice. Not all lifetimes are necessarily comparable, but those within one function should be. There should be an upper bound and lower bound.

Later research could look into this possibility.

Types

In this thesis we did not look at more types than simply `Int`. While we could have chosen pretty much any type to demonstrate our ideas, integers proved to be easy for the reader, while being complex enough to allow for expressions with more than one variable. Future research should look into providing a more diverse typing system for Rust.

Also, the current thesis did not provide a type check at all. With the syntax of section 4.2 we can make programs that are not typed correctly. As stated there, we simply assumed all programs under consideration were typed correctly, but this is obviously not very satisfactory when looking at the bigger picture. A type checker could be incorporated in the compile check or could be a separate check. Either way, a more complete description of Rust needs a type checker.

Functions

Functions are important to make a usable programming language. From Chapter 2, it seems likely that lifetimes will be needed to make a successful semantic implementation of this.

Other features of Rust

Rust contains many more features that are interesting to look into. Think of vectors or other more complicated data structures, concurrency and I/O. All of these will be interesting, because differences with other programming languages will occur due to the ownership system.

5.1.3 Formalization

All of the work in this thesis is done on paper and lacks computer verification. While a start was made to formalize the ‘move only’-language (Chapter 3) in the Coq Theorem Prover (Barras et al., 1997), this was not finished due to time constraints. The first attempt can be found on the GitHub page of the author (Wessel, 2019).

In order to make completely sure that all proofs presented in this thesis are correct, they should be verified in Coq or a similar program in the future.

5.1.4 Correctness of translation

While we proved some minor properties of our compile time checker in the previous chapters, this is in no way a guarantee that the translation is a 100% correct and completely models the Rust language. We did run quite a few experiments with the Rust compiler to see what is accepted and what is rejected. Some of these experiments are in this thesis as examples. However, a more thorough investigation could certainly be useful.

5.2 Conclusion

In the previous chapters we looked at a small but interesting part of Rust. After a walkthrough of the interesting features of Rust in Chapter 2, we started out with a big step semantics describing the moving behaviour. Later on, we moved on to a small step semantics, where we separated the checks the Rust compiler does from the semantic rules.

In the chapter after that, we added more features to the subset under consideration: borrowing and mutability. In both chapters, we proved safety by proving progress and preservation. We also looked at other properties of our semantic rules and compile check, such as determinism.

Bibliography

Appel, A. W. and Blazy, S. (2007). Separation logic for small-step Cminor. In *International Conference on Theorem Proving in Higher Order Logics*, pages 5–21. Springer.

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.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. (1997). *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria.

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.

Boyland, J. (2003). Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer.

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.

Nielson, H. R. and Nielson, F. (1992). *Semantics with applications*, volume 104. Springer.

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*.

Stroustrup, B. (1994). *The design and evolution of C++*. Pearson Education India.

The Rust Project (2017a). *Lifetimes*. https://doc.rust-lang.org/beta/nomicon/lifetimes.html.

The Rust Project (2017b). *Validating References with Lifetimes*. https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html.

Wessel, N. (2019). *Coq Formalization of moving in Rust*. https://github.com/NienkeWessel/BachelorThesis/tree/master/coq.

Wright, A. K. and Felleisen, M. (1994). A syntactic approach to type soundness. *Information and computation*, 115(1):38–94.