Seminar Report

Rust

Christopher Naschberger

29 July 2015

Supervisor: Dr. Cezary Kaliszyk

Abstract

This seminar report gives a short description about the programming language `Rust`. Its main focus is giving a basic understanding of the language, its advantages in comparison with other common languages and its notable features.

Contents

Table 1: An overview of the history of `Rust`'s development.

Year	Event
2006	personal project of Graydon Hoare
2009	sponsoring through Mozilla
2010	public announcement
2011	self-hosted compiler *(before it was written in Ocaml)*
2015	first stable release
planned	every six weeks a new release

1 Introduction

This document offers an overview of the programming language `Rust`. It will compare it to well-known languages and also introduce some of its more unique features. However, it is far from exhaustive. It is advised to either take a look at the official Rust-Book [7] and/or the website RustByExample [6] for learning everything about its syntax and usage. For instance, it only mentions the subject of writing unsafe code in `Rust` which allows the usage of pointers which are otherwise completely banned. Finally, it is advised to read every subsection of section 4 in order as later subsection build to some extent on the understanding of earlier ones.

2 General

This section provides a general overview of `Rust`. For more in-depth details about its features refer to sections 3 & 4.

2.1 Intention

`Rust` is a relatively new expression based system programming language, which combines paradigms of multiple commonly used languages. Its development was driven by the intention to created a system programming language that uses well-established features found in other languages but not yet implemented in widely-used system languages or only in ones which offer unsafe memory models[1]. For instance it is possible to directly control the memory layout like in `C`, while having better memory safety, and using features from functional programming.

2.2 History

Starting as a personal project of Mozilla employee Graydon Hoare in 2010, `Rust` was officially released in May 2015. Table 1 shows the history of it starting from the beginning. The most interesting event was probably around 2011, when the language was the first time compiled with self-hosted compiler i.e., a compiler written in the language itself.

[1] `http://www.infoq.com/news/2012/08/Interview-Rust` [Online; accessed 25-July-2015]

2.3 Origin of Name

There are a lot of explanations for its name as its creator kept on making up new ones when asked. However, he personally admitted that it was probably because of the fungus of the same name as they are extremely robust just like the language should be[2].

2.4 Development

Even though Mozilla is sponsoring the development of the language, it is open source and highly community driven with a repository on GitHub [1]. The contributors give themselves the nickname 'Rustaceans' [7]. As a result of the open development with input from many different sources a lot of its features have been changed frequently since it has been deceived. Before the official release it was easily possible, that old `Rust`-programs would stop compiling after a new release as an used feature got dropped or changed. For instance at first the object system was depending on garbage collection, but this was later changed to the borrowing system which is one of the defining features of `Rust`. It will be explained in closer details in section 4.1.

Starting with the first official release it has been agreed on avoiding major changes and, if needed, use deprecation to mark old features and only remove them in the next release[3]. It is, consequently, important to check articles, reports, etc. about `Rust` for their publication-date as a lot of things which were applicable months before the officially release may not be any more.

2.5 Guarantee of Memory Safety

The question most people will probably ask themselves, when confronted with a new programming language, is "why should I learn another language?". The main reason for using `Rust` over other languages is its guarantee of being stable and (memory) safe. Because of the strong constraints the programmer has to follow when programming they can make the assumption, that when the program crashes it is the fault of the used libraries not theirs.

The only exception to that would be if, they wrote unsafe code themselves. However, even in that case they can pinpoint the reason for the crash easily as such code has to be inside an unsafe block. Therefore, only that code needs to be checked for ensuring that the used library is at fault. Furthermore, those code blocks do still not allow everything, which makes them safer than normal code in other languages.

3 Language Details

The next paragraphs will take a more specific look at `Rust` and its features.

[2] `http://www.reddit.com/r/rust/comments/27jvdt/internet_archaeology_the_definitive_endall_source/` [Online; accessed 25-July-2015]

[3] `http://blog.rust-lang.org/2015/05/15/Rust-1.0.html` [Online; accessed 25-July-2015]

```rust
fn main() {
   println!("Hello World!");
}
```

Listing 1: The obligatory hello world program in `Rust`. The exclamation mark in `println!()` means that a macro is called.

`Rust` combines most of the common programming paradigms. It is imperative with functional features and has an object system which allows to some extent object-oriented programming.

> "`Rust` is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety [5]."

This quote taken directly from the official homepage describes what `Rust` wants to achieve. In general a language either has low-level access and is fast, but the memory safety relies on the programmer's use of e.g., pointers, or it is safe but slow as the high-level abstractions which ensure the safety lead to a lot of overhead. An example for the first one would be `C`, for the second one `Java`.

`Rust`, on the other hand, wants to offer safety *and* performance. It has no garbage collection and also does not rely on the programmer to free the data - it does so automatically as soon as it is not needed any more. Furthermore, it is not possible to use pointers, which point to invalid data. As a result, there are no `null`-pointers or so called dangling pointers, which point to resources that have been freed. This is ensured by the feature called 'borrowing', which is explained in detail in section 4.1. Furthermore, it guarantees that it is impossible to create a race condition in `Rust`.

This abstraction, as most of the other ones in this language, are so called zero-cost abstractions i.e., they are all checked at compile time and, consequently, will not slow down the performance of the program. This obviously leads to slower compile times as other languages. However, the performance itself is comparable to a program written in C which has the same safety precautions [9].

As seen in listing 1, `Rust` has a similar syntax to languages such as `C` and `Java`, however, its features differ a lot from theirs. The ones seen here are the different ways of declaring a function and also the way macros are called. In this case `print!()` is a macro that allows text to be outputted and it is called via the exclamation mark. Refer to section 5 for a better comparison to other languages.

`Rust` is an expression based language with only two types of statements. The first one is the binding statement which binds data to a variable and the second one is the expression statement which every expression becomes by adding a semicolon at the end of it. If there is a whole code block, its last line determines its value - if that line is a statement it is of the type unit `()`, otherwise it is the value of the last line.

In general, types can be omitted in `Rust` as long as they are inside a function body because it has a strong type-system. In listing 2 are some examples which show ways to bind data. Besides in the case of variable `b` all types are omitted. Every expression has a value and can, therefore, be bound. Consequently, the if-else-structure can directly be

used instead of a ternary operator most other languages have. It also does not matter how many blocks an expression is encapsulated in as every block will have the value of its last line which is always the expression itself. Furthermore, in for example listing 3 the types of `triple` & `func` could be dropped.

```
let a = 4;              //4 is now bound to a
let b:i32 = 4;          //explicitly stating the type
let x = {{{{a - b}}}};
let y = if x > 0 {a} else {b};
let mut z = 0;          //only the value of this variable can be changed
                        //because all others are immutable
```

Listing 2: Examples which show some possible ways to bind data to a variable.

For following an object-oriented paradigm `Rust` has an object system which uses structures and traits and is similar to the type-system of `Haskell`. The section 4.2 takes a closer look at it. Moreover, polymorphism is allowed via generics similar to `Java`, however, this will not be explained by this report.

Furthermore, the language offers common functional features such as exhaustive pattern matching, tuples, usage of high order functions, i.e., functions that possibly accept and return other functions, and anonymous functions called closures. Examples can be found in listing 3. Here the function `hof` expects as an argument a function which has one integer argument and also returns an integer. For declaring what type of function is needed `Rust`'s generics are used. In the main function a closure is bound. Then the variable `triple` is pattern matched and depending on the result the high order function is called with the bound closure or 'Nothing happens' is printed.

```
fn hof<F>(f:F) -> i32 where F : Fn(i32)-> i32{
    //...
}

fn main(){
    let mut triple:(i32, i32, i32) = (0, 0, 0);
    //...
    let func = |x:i32| -> i32 {x+1};
    match triple {
        (0, _, _) => {hof(func);},
                    //call high-order function with closure
        _         => {println!("Nothing happens");},
   }
}
```

Listing 3: This code shows some of `Rust`'s functional features. In no specific order: high-order functions, anonymous functions, pattern matching and tuples.

Finally, `Rust` offers a foreign-function-interface which means, that its code can be compiled with the right options and then its functions may be called from another language. Moreover, it is possible to call code from other languages inside `Rust` as long as it is put inside an unsafe-block.

4 Notable Features

This chapter will take a look at some of `Rust`'s more unique features and describe them in more details, so that the reader has an easier time understanding the important mechanics of the language.

4.1 Memory System

This section is focussed on the defining feature of `Rust`'s stability, the so called 'borrowing', where data can be accessed through more than one variable which is normally not allowed.

In general, languages allow the programmer access to data via pointers. However, those can be pretty dangerous to the memory safety as they may lead to crashes if invalid data is accessed. Therefore, either the programmer needs to be careful of their usage or procedures similar to garbage collecting need to be used to ensure that the program will, eventually, have new space for data. Both of the mentioned options have disadvantages and advantages. `Rust` tries to find a balance with its 'borrowing' system so that it can eliminate most of the weaknesses and still keep the advantages. A more throughout comparison can be found in chapter 5.

4.1.1 Ownership

Whenever data is bound in `Rust` the variable it is bound to becomes its sole owner. This means that, first of all, only this variable has read or write access and, secondly, that at the end of its lifetime, which is normally at the end of its scope, the compiler can free the data without any complications. It is possible to rebind data to a new variable, however, that also means the ownership is moved and only the new variable may now access it. If the programmer tries to use the old variable the program will not compile and the compiler will complain about using the moved value.

There is one exception to this rule. For all the primitive types and all types that implement the trait `Copy` (for explanation of traits see section 4.2) the value will be copied and the ownership stays the same i.e., both variables can still be used afterwards, but changing the value of one will *not* change the value of the other.

4.1.2 Borrowing

The biggest issue with `Rust`'s memory system with the current knowledge is, that all function calls are call-by-value and, as a result, as soon as one is called the ownership is moved to the function variables. The lifetimes of those only apply to the function, which means that at the end of it the data is freed and, consequently, unreachable afterwards,

which, in general, will not be wanted. To avoid this, either every time the still needed variables have to be returned (possibly inside a tuple with the result), or the programmer needs to use 'borrowing' when calling the function.

Borrowing means that variables get access to data which they are not the owner of. Depending on the access (read/write) they get, they may also stop the owner from accessing it. At the end of the variables' scopes the owner gets back the full access to the data. There are two types of borrows. First of all *immutable references* (`&T`) which only allow read access and secondly *mutable references* (`&mut T`) which also allow write access. To ensure that there is neither a memory nor a concurrency issue, there are two rules.

- a borrow must last for a smaller scope
- only one kind of reference is possible:
  - 0..N immutable references
  - exactly 1 mutable reference

The reason for the first rule is obvious. If a borrow would last longer than the scope of the owner, it is possible that the borrower tries to access data which is already freed. The second rule ensures, that it is not possible to create a race condition, see section 4.1.3 for an explanation.

In listing 4 an example of borrowing is given. First a vector is created via the macro `vec!` and then a function is called with it. This function adds all of its elements together and returns the result. Normally, as a vector is not a primitive type and also does not implement the trait `Copy`, it would be freed at the end of the function. However, in this case it is only borrowed, which means it can still be printed in the `main`-function.

```
fn sum(vector:&Vec<i32>) -> i32{
    let mut sum = 0;
    for num in vector{
        sum = sum + num;
    }
    sum
}

fn main(){
    let vector = vec![1,2,3];
    let sum = sum(&vector);
    println!("{} is sum of {:?}", sum, vector);
    //prints '6 is sum of [1, 2, 3]'
}
```

Listing 4: An example for borrowing, where the sum of all elements of a vector and the vector itself are printed.

4.1.3 Concurrency

For concurrency `Rust` follows the paradigm that functions calls are not synchronized but the data itself is locked. In the simplest case this means that the ownership or the borrow of data needs to be send between threads and the threads currently without ownership or a borrow are locked out from accessing it. Looking at the second rule for borrowing, which only allows one type of borrow at a time, it is easy to see why this will prevent race conditions.

A race condition happens, if two or more threads are accessing a resource without synchronization and one or more is currently writing. In `Rust`'s case either just one is writing and no one else is accessing it or there are possibly more than one accessing it, however, none of them has a write access. So in both situations one of the three conditions does not hold and, therefore, no data race can happen.

Normally, however, it is needed that more than one thread can access data at the same time. For that `Rust` offers types in the standard library for instance `AtomicBool` which allows atomic access and `Arc` which more or less allows synchronized access to the data. The important part is, that as long as the programmer is not using the unsafe-syntax for writing code, they can be sure that there will not be a data race. If there happens to be one, it is not their fault but the fault of the developer of the library they used.

4.2 Object System

The next paragraphs will describe the object system in more detail. It can be used in a similar way to for instance the class-based one of `Java`. It is, however, closer to the typesystem in `Haskell`.

One of `Rust`'s more high level features is the usage of an object system. Similar to generics in other languages you can define function such that any object can be given and/or returned as long as it fits a certain constraint. In for instance `Java` such constraint would be that the object gotten must derive from a certain class or implement a certain interface. In `Rust` it is via so called traits, which a structure must implement.

4.2.1 Structures

In every `Rust` program it is possible to group one or more types together into a so called structure. Those types get an unique name (key) and are the fields of the structure. It is possible that a structure is the field of another structure, however, they cannot be the same structure. The reason for this is as for initializing an object a value must be provided for every type it holds. Because there exists no `null` in `Rust`, it would not be possible to create an 'initial'-object which holds nothing.

Another possibility for initializing is by giving an already initialized object and only the key-value pairs that should be different. If objects are often initialized with similar values there are two solutions. Either a wrapper is used which fills the object with default values in case not all were given or a default object is created and used every time.

The important thing to remember while using structures is that the mutability of the

fields is not determined when the structure is defined. Instead it is determined when the binding to a variable is done. This means that either all fields are mutable or none.

4.2.2 Implementation of Methods

To make calling functions on structures easier it is possible to attach functions to objects. Similar to for instance `Java` those methods can then be called directly on it. Normally, those methods have as the first parameter either `self`, `&self` or `&mut self` which refers to or is the object the methods are called on.

In listing 5 a structure is defined and three methods are attached to it. The first one does not have the `self` reference, which means that method is only callable via the type itself. This is done to allow that a function name can occur more than once - inside the scope of the trait and outside of it (possibly inside another trait). Via the operator `::` it is possible to define the scope of the function to be called. In this case this is the one of the `Foo` implementation.

This operator is also used in case an object implements two traits which have a method with the same name. By specifying the scope it is then possible to chose the right one (for example deciding between the `doSomething()`-method of the traits `Foo` and `Bar` - `Foo::doSomething(obj)` or `Bar::doSomething(obj)`).

The other two methods in the listing get a mutable reference to the object as they do not change anything and just return a new object.

Theoretically, in this situation `self` could be used in both cases. In general, however, this may not be wanted, as using this means, that the object will be freed at the end of the function call and, therefore, unusable afterwards (the reasons for this are explained in section 4.1). Even more, it would probably be better to implement all three methods in the same way without `self` as the object is never needed, but for the sake of the example it is used here in this way.

Eventually, in the `main`-function an object is initialized by calling the `new`-function and the `bar()`-method is called on it directly. This call returns another object of the type `Foo` on which the `baz()`-method is called, eventually.

```
struct Foo{
    x: i32,
}

imp Foo{
    fn new()      -> Foo {Foo {x:0}}
    fn bar(&self) -> Foo {Foo {x:1}}
    fn baz(&self) -> Foo {Foo {x:2}}
}

fn main(){
    let foo = Foo::new();
    foo.bar().baz();
    //if not attached the call would look similar to this:
    //baz(bar(foo));
}
```

Listing 5: In this example a structure is defined and methods, which are, eventually, called on an initialized object, get attached to it.

4.2.3 Traits

In `Rust` it is also possible to group structures together by using traits. This allows polymorphism to work as the traits can put constraints on parameter for function calls. If a structure implements a trait it simply means that it implements all of the functions defined in that trait. If an implementation will be similar for all structures that implement it, traits offer the possibility of defining a default implementation which can be overridden if needed. In some situation it may be wanted that a trait can only be implemented if another trait is implemented too. In this case it is possible to define that a trait inherits from another. There are two restrictions when implementing traits.

- either the trait or the type must be user-defined
- traits only apply in the scope they are defined

The first rule allows that, theoretically, even primitive types can implement a user-defined trait. However, this is seen as poor style and should, therefore, be avoided. It also allows the implementation of already defined traits like `Copy` for user-defined structures. The second one ensures that even though someone may implement user-defined traits for a primitive type, it does not matter for a programmer as long as they do not use the corresponding traits [7].

The example in listing 6 shows two traits with `Square` inheriting from `Shape` and two structures. The structure `Point` has no fields and its implementation of `Shape` does not override the default method and is, consequently, empty. On the other hand `MySquare` overrides the default implementation for `Shape`. The important thing to see here is that,

if a structure wants to implement `Square` it also has to implement `Shape` as `Square` inherits from that trait. Otherwise the program cannot be compiled.

```
trait Shape{
    fn calcArea(&self) -> i32 {0}
}
trait Square : Shape{
    fn getX(&self) -> i32;
}
struct Point;
struct MySquare{
    x: i32,
}
impl Shape for Point{}
impl Shape for MySquare{
    fn calcArea(&self) -> i32 {self.x * self.x}
}
impl Square for MySquare{
    fn getX(&self) -> i32 {self.x}
}
```

Listing 6: In this code segment two traits and two structures are defined. The two structures then implement the traits.

Theoretically, it is possible to follow object-oriented paradigm while using `Rust`. Traits can be seen as a mixture of interfaces and abstract classes, classes are the combination of structures, traits and attached methods and, finally, inheritance can be simulated through trait's inheritance. However, there is no need to do so and in some cases it may even be a good idea to not follow it as it may cause some problems to be harder.

5 Comparison to other Languages

This section will compare `Rust` with imperative and functional languages and will try to show some of its advantages.

5.1 Imperative Languages

`Rust`'s syntax is extremely similar to the one used in `C` and `Java`, but besides that it differs in a lot of aspects. The most distinct ones are the different ways memory management is handled.

5.1.1 Memory-Management in C

C uses a system where the programmer is the one responsible for its safety as it is close to the machine and, therefore, allows a lot of optimizations. That means on the one hand that programs can be written that run fast because there can be almost no overhead, but on the other hand one invalid pointer could make the whole program crash. This can happen quite often if the programmer is not careful as it is extremely easy to free data that is still pointed to from elsewhere.

5.1.2 Memory-Management in Java

Java counts every reference to an object and, in general, only frees it if it can be sure that nothing points to it. This is done by the so called garbage collector which runs from time to time or when memory gets low. Objects may have no references at all but still exist as long as no garbage collecting has happened yet. It is possible that objects are freed which still have references to them if for instance memory gets critically low, however, this is very unlikely. Garbage collecting can, when executed, depending on the number of objects, currently running calculations, etc. slow the program down.

Overall, that means that Java-programs are not prone to crash because of bad memory access, but their performance can be bad, even considering that the code is executed on a virtual machine and not really optimized for the hardware used.

5.1.3 Memory-Management in Rust

Rust tries to find a middle way between those two systems. As data is automatically freed when not needed any more, there is no need for garbage collecting, which means it is a lot faster than Java. Furthermore, the programmer cannot, besides if they are using unsafe code, write code which will create an invalid pointer. It is, in general, slower as C as there is a little bit of overhead and also some tricks which are used in C to improve the performance by directly manipulating memory addresses are not possible.

5.1.4 Usage of Code-Analysis-Tools

There exists software, e.g., code-analysis tools, which makes it possible to find memory errors like invalid pointers in C and other languages, however, Rust is still preferable as it was already developed with that in mind. First of all there is no guarantee that every programmer will use those tools. So even though they may exist, they may never be used in commercial code for different reasons e.g., to save time, not well enough known and so on.

Rust on the other hand has no way around it. Everyone who writes code in this language will deal with memory safety. This implies also the second advantage of Rust. As long as the programmer does not use unsafe blocks extensively, which would also be far from the intended usage, the other developer can be sure that the program will not crash because of memory issues, or at least they know where to look for the source of

those errors. This makes it far easier for beginners to start working on projects as there is less pressure because there are less things to get wrong.

5.2 Functional Languages

Additionally, `Rust` has far more functional features than `Java` or `C`. None of them allow for instance the usage of tuples or pattern matching. It is possible to write pure-functional programs in `Rust`. However, if used normally, it is no *pure*-functional language as functions can have side-effects and, moreover, it is possible to mutate data. A direct comparison of `Rust`, `Ocaml` and `Haskell` can be found on the homepage of Raphael Poss[4]. It is, however, over one year old and as a result a bit outdated. Nevertheless, it is still usable - all code examples given should work as long as the types are corrected (`int`/`uint` to `isize`/`usize`). The last chapter 'Lifetime and storage, and managed objects' should, however, be completely ignored as most of its content is not applicable any more.

6 Conclusion

This final chapter will summarize the report and give hints for easing learning `Rust`.

Overall, `Rust` offers some interesting features which are useful when safe parallel execution is wanted. An example usage for that would be a web-browser which is highly complex and depends on a lot of parallel tasks. Mozilla's experimental web browser 'Servo' has been written in `Rust`[5].

The biggest obstacle when using the language is understanding borrowing. At first the compiler will seem to complain about things that should work but do not. However, after some time it gets more and more obvious how the mechanic behind borrowing works. Eventually, it becomes clear how favourable the guarantee is it offers.

Learning the language is probably best done by following the official Rust-book [7] and using the website 'Rust by Example' [6] which has, as the name already suggests, a lot of examples for all possible features of `Rust`.

For the first smaller programs it will be enough to use the online compiler called 'Rust Playground'[6] but, eventually, a switch to an editor or IDE is advised. Most common editors offer a plugin for syntax-highlighting e.g., 'Vim', 'Emacs', 'Atom' and so on. If auto-completion is wanted, there exists RACER (**R**ust **A**uto-**C**omplete-**er**)[7], which can be used as a plugin too. At the moment of writing, there is only one real IDE available made specifically for `Rust` which is called 'SolidOak'[8]. It offers a simple user interface and uses 'Neovim' as editor. It comes with both plugins already installed.

Learning or at least trying `Rust` is definitely worth doing, as its unique features need a different way of thinking and, as a result, often offer new insights on problems.

[4] `http://science.raphael.poss.name/rust-for-functional-programmers.html` [Online; accessed 25-July-2015]
[5] `https://github.com/servo/servo` [Online; accessed 29-July-2015]
[6] `https://play.rust-lang.org/` [Online; accessed 25-July-2015]
[7] `https://github.com/phildawes/racer` [Online; accessed 25-July-2015]
[8] `https://sekao.net/solidoak/` [Online; accessed 25-July-2015]

References

[1] GitHub. Rust on GitHub. https://github.com/rust-lang/rust, 2015. [Online; accessed 25-July-2015].

[2] Steve Klabnik. The Story of Rust. http://www.steveklabnik.com/fosdem2015/, 2015. [Online; accessed 25-July-2015].

[3] Niko Matsakis. Baby Steps. http://smallcultfollowing.com/babysteps/, 2015. [Online; accessed 25-July-2015].

[4] Reddit. Rust's Subreddit. http://reddit.com/r/rust, 2015. [Online; accessed 25-July-2015].

[5] Rustaceans. Official Homepage. http://www.rust-lang.org/, 2015. [Online; accessed 25-July-2015].

[6] Rustaceans. Rust by Example. http://rustbyexample.com/, 2015. [Online; accessed 25-July-2015].

[7] Rustaceans. The Rust Book. https://doc.rust-lang.org/stable/book/, 2015. [Online; accessed 25-July-2015].

[8] StackOverflow. Rust on StackOverflow. http://stackoverflow.com/questions/tagged/rust, 2015. [Online; accessed 25-July-2015].

[9] Wikipedia. Rust (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Rust_%28programming_language%29, 2015. [Online; accessed 25-July-2015].