Rust Programming Language

By: Michael Plekan

Introduction

The Rust programming language is a pretty new language relative to C or Java, only being first created in 2015. My interest in it comes from the combination of features to improve writing speed by shorthand syntax and the execution speed on the same level as C. The overall look and feel is drastically different from Python and Java with colons and arrows being used quite frequently in the syntax. For this project I go over a lot of the major features of Rust and compare them to Java, Python, or C. I then go into a few features in more detail in the deep dive section. Lastly, I talk about the Jacobi iteration program I made during this project.

Abstract

Throughout this paper it will go over some of the main features in Rust like loops, data types, and control flow structures. I use examples to show how some of these features look. I take a closer look at memory management, option types, and match statements in the deep dive section. These are some of the more interesting or tougher to understand features I found. I also talk about the program I wrote, going in different language features I used in it.

Language Overview

This section highlights some of the language features with some examples of the features showing off how they might be used. This section is to give readers a general idea of Rust and some of the differences. All examples shown in this paper can be found within the example codes of the repository.[1]

Data Types

Primitives

Rust has the same standard primitive types but the syntax for them is different. In C there is `int, float, long`, etc, whereas in Rust use `i`,`u`, or `f` followed by `8,32,64,128`, or `size`. The letter specifies whether it is a signed integer, unsigned integer or floating point number and the number specifies the number of bits to use or use the architecture default. This departure from C can be confusing at first but it tells the programmer more information about the variable than the normal naming convention does. This can be nice when debugging overflow errors or trying to reduce memory use to a minimum. I think this is a good thing for both readability and writability after getting used to it.

[1] https://github.com/MikePlekan/Rust_Research

Strings and str

Rust has two types of strings. The first is `String` which is stored in a vector which is the Rust version of an Arraylist or dynamically sized array. This storage method allows them to be growable. The second type of string is `str` which seems to be the more simple almost C like string. The best part about strings in rust is the number of functions that are available to modify them.

```
string.chars().filter(|x|*x=='l').count
```

Just within this simple example the programmer can split the string into chars, filter it and count the number of occurrences that passed the filter. This is only skimming the surface of what Rust has built in to work with and modify strings.

Custom Types

There are two main custom types in Rust, the struct and the enum. The struct type is a lot like the structs in C. They can hold multiple named fields. The fields don't have to match types. Where Rust really differs is with the enum type. They can be used like C enums but they can also do so much more. You can have structs as options in the enum. You can also specify that you want to use the names within the enum without having to use the name of the enum. This would be very useful in the tokenizer and parser because we can use the names but not have to worry about what they equal.

One last feature for both of these is the ability to add functions to them.

```
Coins{
enny=1,
ickel=5,
ime=10,
uater=25,
ollar(i32)


Coins{
n to_dollars(&self)->f64{
    match self{
        Coins::Penny => return 0.01,
        Coins::Nickel => return 0.05,
        Coins::Dime => return 0.1,
        Coins::Quater => return 0.25,
        Coins::Dollar(v) => return *v as f64,
    }
}
```

This example shows an enum called coins where there is a function to convert the values into the dollar amount. The self parameter is very similar to self in python. Overall this extra functionality is nice and can make the code more readable by removing extra words that don't need to be there, like in the tokenizer and parser example.

Modules

The modules in Rust are like classes in Java and python. They can hold functions, structs, and more. They can nest modules within modules. They have a hierarchical system and can manage permissions (private/ public) like Java.

```rust
pub mod fruit {
    pub mod seed {
        pub static NUM:i32 = 10;
        pub fn eat() {
            println!("Eating seed");
        }

    }
    fn eat() {
        println!("Eating fruit");
    }
    pub fn peel() {
        println!("Peeling fruit");
    }
}
```

In the example it shows a fruit module with a sub module and 2 functions. The `pub` keyword is for making functions and variables public.The `eat` function is not public so it will never be used since nothing inside the module calls it.

Control Flow

Loops

We all know about the for and while loops and Rust has them like any other language but it also has some neat features for them as well. There is a `loop` keyword for infinite loops instead of having to do `while(true)` or a variation of that. You can also label your loops as shown in the example below.

```rust
'outer: while x<10 {//doesn't reach 10 b
    println!("Entered the outer loop");
    x+=1;
    'inner: loop {
```

I think this is a very useful feature because it not only allows readers to refer to different loops easier, it also lets you say exactly where you want a break to go. This improves both readability and writability. Another nice feature of the loops is that you can assign a variable to the result almost like a function call.

```rust
let mut i = 2;
let solution:i32 = loop{
    if i >127 {
        break i;
    }
    i*=2;
};
```

I think this is an interesting feature because on one hand it shortens the code and makes it clear what you're trying to get from a loop but it can also make it harder to spot the assignment statements if you're not used to seeing them before loops.

If statements

The if statements are basically the same as any other language except for a feature we saw in loops. This is the ability to assign a variable to the result of an if statement. It is sort of like a ternary operator taken to the extreme. This feature drastically increases the readability over other languages that can only match this using ternary operators.

Match statements

The match statements in Rust are alot like the switch statements of Java and C but like a lot of other features they have extra functionality. They can handle many different types of things, like strings, numbers, enums, etc. They can also match arrays, tuples, and structs. There are a lot of interesting things that can be done with them but that will be covered in the deep dive section.

Functions

Full Functions

These are very similar to your functions in any other language. An arrow(->) is used to define the return type instead of putting in front of the function name as seen in the example function header below.

```
fn sum(x: i32, y: i32) -> i32 {
    x + y
}
```

The biggest difference is that functions in Rust can't just use variables that live outside the function. I think this has to do with memory management.

Closures

These are seemingly shorthand functions and a lot of the time you can treat them like a shorthand function. One major difference is that these can grab variables from outside of the closure. A fair warning when doing this, the closure burrows that variable until the last time the closure is called. This can limit what you can do with the variable during that time.

Memory Management

The way Rust handles memory management is one of the unique features to Rust. All of the memory safety checking and handling is figured out at compile time. Which for performance is really good since you don't need to add a garbage collector which runs during runtime. This way of dealing with memory safety comes with a few new implementation considerations. The borrow checker/ system. As a programmer you get a few different types of ways to pass variables around. There is a pass by value, pass by reference and a pass by mutable reference. These are pretty similar to most other languages but the restrictions that they place on the variable is where the difference is. I will get more into each one in the deep dive section.

Deep Dive

For the deep dive section, it will be looking into a few topics more than what was covered in the overview section.

Match statements

The match statements have a bit more to offer with binding and guardian which makes the match statement that much more powerful. First is binding which allows the programmer to bind

the input to a variable for a case. As shown in the example, it binds character ‘a’ to l,u, or n

```
let b = match 'a' {
    l @ 'a' ..= 'z' => format!("im the lowercase letter {}", l),
    u @ 'A' ..= 'Z' => format!("IM THE UPPERCASE LETTER {}", u),
    n => format!("im not a letter {}", n),
};
```

depending on the case. This allows you to use it within the case. The next feature is guardian which allows you to add more conditions to cases using an if statement as seen in the example below.

```
match [1,2,3,4,5,6,7,8,9,10] {
    [a, b, c, ..] if a < 0 => println!("a = {}, b = {}, c = {}", a, b, c),
    [a, b, .., c] if c == a => println!("a = {}, b = {}, c = {}", a, b, c),
    [a, b, ..] if a > 0 => println!("a = {}, b = {}", a, b),

    _ => println!("anything"),
}
```

Option types

The Option type is an enum with `some` and `none`. This is used for making tolerant code. You can wrap any type or struct within the option. This is so you can check the option for `some` or `none` instead of having to check for null every time. You can use them in if let Statements, let else Statements, while let loops which are all control flow structures that work with the option type to make the code easier to write and understand. For example if let statements allow you to unwrap the option type if it is some and bind it to a variable. The option type seems really cool but coming from C I still am getting used to where to use them over a null check.

Ownership and Borrowing

The ownership model in Rust is weird coming from C or Java. As mentioned in the overview there are a few different ways to pass variables around. To go through the different ways, let's assume the variable is getting passed into a function as a parameter. Passing by value will transfer ownership to the function. This can be a problem because now when the function returns the variable will be freed since it was a parameter and owned by the function. Passing by value without transferring ownership is possible by calling a clone function. Next pass by reference which borrows the variable for the lifetime of the function, then ownership is returned to the original variable. Finally the mutable reference is basically the same but it allows the function to edit it. There is one big difference though, the permissions you have

with the variable while it is being burrowed is restricted.

Jacobi Program

While learning Rust, I built a Jacobi iteration program in Rust. I made it using a bunch of different language features. Rust does have the ability to do multithreading for a program like this but it requires a higher level of knowledge than I could gain within this project. The memory safety of Rust makes it a bit more tricky than doing it in C.

Struct

I created a struct for holding the data in each position of the matrix.

```
struct Field {
    value: f64,
    permanent: bool,
}
```

The `value` field holds the numeric value and the `permanent` field is set true for spots that shouldn't be changed. This is very straightforward if you have made a struct in C. Where this differs is using the `derive` keyword as seen below.

```
#[derive(Debug, Clone, Copy)]
struct Field {
```

This keyword along with the fields inside are things that I haven't seen in other languages in this format. The derive is telling the compiler to build the functions or other things needed to implement `Debug`, `Clone`, and `Copy`.

`Debug` is used for getting a simple print format of the struct. `Clone` and `Copy` are for moving the data around. This is a part of the language I didn't dive into that much but it definitely seems like it could be very useful.

Enums

I used an enum inorder to manage the start condition.

```
//create an enum for the start type
enum Start {
    Boundary,//all edges are 1
    Diagonal,//diagonal is 1
    Random,//random points are 1
}
```

There are 3 types of start positions that the program can do. The boundary puts ones along all the edges and sets them all to permanent as the struct describes. The diagonal sets a line of ones going diagonally through the matrix and set to permanent. The last is just a random amount and at random positions get set to 1 and set to permanent. This enum lets me cleanly use the match later on.One last thing to mention is this line below.

```
fn main() {
    use Start::*;
```

This is very similar to C++ where you can say use a namespace for this section so you don't have to type it out everytime. The line above does this for the enum I created.

Macros

I used 2 different macros in this program. One for printing the matrix in a nice format and the other for setting the fields within the field structure as seen below.

```
macro_rules! set {
    ($x:expr, $y:expr) => {
        matrix1[$x][$y].value = 1.0;
        matrix1[$x][$y].permanent = true;
    };
}
```

The macros in Rust are alot like C with a slightly different syntax. I think they can be very useful for repetitive tasks that don't need the full functionality of a function or closure.

Match statement

The start condition is handled using the enum and this match statement evaluates the enum and actually modifies the matrix accordingly. As seen in the example this is where I used the macro shown in the last section. I think this is a very readable way of organizing this.

```
//match the start type
match start {
    Boundary => {
        for i in 0..size {
            set!(0, i);
            set!(size-1, i);
            set!(i, 0);
            set!(i, size-1);
        }
    },
    Diagonal => {
        for i in 0..size {
            set!(i, i);
        }
    },
    Random => {
        use rand::Rng;
        let mut rng = rand::thread_rng();
        for i in 0..size {
            for j in 0..size {
                if rng.gen() {
                    set!(i, j);
```

Conclusion

In the future there are so many more features and use cases for Rust that I want to explore. Even though I went through a lot of features in this paper, it just skims the surface. They have an entire book like C for the Rust language, if that gives an idea of how big the language is. Also there is the package manager which expands the language to do pretty much whatever you want. An area of Rust that I would like to explore more is the multithreaded appellation of Rust. I'm curious to figure out if the safety features of Rust stop it from getting some of the same performance results as C.

Citations:

Introduction - Rust By Example

The Rust Programming Language

Rust Docs