Lily Sanchez and Jacob Valero

Final Rust Report

Introduction

Rust began development in 2006 by researchers at Mozilla research, and a stable version was first released in 2015.

It is multi-paradigm, as it's both functional and imperative.

Rust, as a language, prioritizes speed and security, not using runtime or any sort of garbage collection, and doesn't allow null or dangling pointers. There is also an emphasis on ownership and scope of variables.

Why Use Rust? (Features and Distinctions from other Languages)

As StackOverflow's "most loved language" for four years in a row now, Rust is clearly well-beloved by those who use it. Why is that? Below are some reasons as to why someone might want to develop in Rust.

There is a lot of debate about the preference between statically and dynamically typed languages. While dynamically typed languages are less verbose and easier to understand, statically typed languages allow for faster debugging and runtime efficiency. Rust tries to mitigate the tediousness of other statically typed languages while maintaining their benefits by requiring top-level items (think function arguments and constants) to have explicit types, while allowing type inference in function bodies. The compiler can then infer the types of variables in function bodies from the typing of the function signature.

To increase the speed of runtime, expedited by garbage-collection being performed (i.e. unnecessary memory gets deleted) at compile time, as opposed to runtime. This also saves on memory usage, and saves time on memory access.

To further increase speed, Rust allows developers to have control over low-level details, similar to other systems programming languages like C and C++. For example, developers can choose whether to store data on the stack or the heap. However, Rust has features that are not available in other systems programming languages. For example, Rust tries to have as many abstractions as possible which still are just as efficient as the non-abstracted code. Rust calls these *zero cost abstractions*.

Safe Rust is designed to help the user avoid undefined behavior, and essentially cleans up as the user goes along (again, at compile time), but users are allowed to write in unsafe Rust should they need more freedom in their code.

Rust was designed to be user-friendly, and fast to both write and run, making it invaluable for use in a variety of contexts, from systems programming to app design.

Built-in Types

Rust has four single-value types: integers, floating point values, Booleans, and characters. Integers and floats can be customized when they are declared, but default to being signed and 32-bit (for integers) and 64-bit (for floats). Booleans are one byte in size, and characters are four

bytes, and can include any Unicode character, including accented letters, foreign characters, and emojis.

Rust also has two built-in types that contain multiple values: tuples, which can contain values of different types, and arrays, which cannot. Both tuples and arrays have a fixed size upon initialization. The values are stored at indices 0–*n*-1, where n is the size of the tuple or array, and are accessed with the forms *variableName***.**index and *variableName***[**index**]**, respectively.

Variables

Variables in Rust are immutable by default, meaning that the programmer must specify if they want to be able to bind a new value to a variable upon its declaration, using the keyword **mut**. The names of immutable variables can, however, be reused with the keyword **let**, effectively replacing the previous reference with whatever the programmer has most recently bound to the variable. This is called "shadowing". Unlike mutable variables, shadowed variables can be assigned new types when they are updated, letting the name *x* for example first reference a boolean, then later, a String, so long as the keyword **let** is used when *x* is bound to a String.

For instance:

```
let x = true;
let x = "Hello";
```

will assign the String value "Hello" to x, even though it had been a boolean previously. The name is simply reused.

Meanwhile:

```
let mut y = true;
y = "Hello"
```

will not compile. Mutable or not, variables cannot have their types changed without being redeclared.

Control Structures

Assignment: let <var> = <expression>

As shown above, initial assignment of variables, known as bindings in rust documentation, begin with **let**. For example:

```
let x = 5;
```

Similarly to Go, Rust has type inference and will infer the value of 5 as a 32bit Integer (i32). However we can also declare the typing of the variable as follows:

```
let x: i32 = 5;
```

Note that Rust does not allow for multiple assignments.

Conditional Statements - If-elif-else

Here is the general structure for if statements, which resemble other programming languages and is similar to Go in that the conditional statement does not need to be surrounded by parentheses while the block is enclosed in braces:

```
if <condition> {
     <statement(s)>
} elif <condition> {
     <statement(s)>
} else {
     <statement(s)>
}
```

Conditional statements - match, if let, while let

match works very similarly to switch statements from languages like Java, C, and Go. However unlike Java and C, there is no default fallthrough behavior.

```
match <variable> {
     <possible instance of the same type> => <expression>
     <possible instance of the same type> => <expression>
     ...
     _ => <expression> //This is the default case
```

For example:

```
let number = 5;
match number {
     1 => println!(“One”);
     5 => println!(“Five”);
     _ => println!(“Not one or five.”);
```

Rust also allows you to set the outcome of a match to a variable as such:

```
let result = match boolean {
     //etc.
}
```

if let and **while let** have similar functionality to match, except for that they allow for non-exhaustive case catching with **if let**, and adding while loop functionality to match with **while let**. These structures allow for more concise expressions when compared to their **match** counterparts. They follow the structure below:

```
if let <data> = <case> {
     <statement>
} else if { // Optional: Alternate case
     <statement>
} else { // Optional: Default failure case
     <statement>
}

while let <data> = <case> { // Think of this first line as the
```

```
    <statements>          // conditional in a while loop
}
```

If you're wondering why these control structures exist, **match** and its concise relatives **if let** and **while let** exist for rust's goal of speed, as simply checking for equality is faster than evaluating boolean expressions.

Loops - loop, while, and for loops

Rust provides three different types of loops defined by their keywords: **loop**, **while** and **for** loops. **loop** is an infinite loop, and **while** and **for** loops resemble the usage of many other programming languages where **while** continues until its condition is not true and **for** loops iterate over a range of values, which can include a data structure that is iterable, such as arrays. See templates below:

```
loop {
    <statements>
}
```

```
while <condition> {
    <statements>
}
```

```
for <variable> in <range or iterable> { //To specify range: x..y where
    <statements>                        //x < y
}
```

Parameter Passing

The parameter passing scheme used by Rust is strictly pass-by-value. The example below demonstrates that no change to n is made by the function and that 100 is printed.

```
fn main() {
  let n: usize = 100;
  inc(n);               // Run inc().
  println!("{}", n);    // No change to n
}

fn inc(n_inc: usize) -> usize {
  n_inc + 1
}
```

It is important to note that references in Rust are first-class citizens and therefore can be passed into functions. However, as Rust is strictly pass-by-value, the reference's value is copied to the subroutine's stack instead of the reference itself. In the example below, inc_ref() (which is identical to inc() above except that it takes a reference typed parameter instead) prints the same output as the above example.

```
fn main() {
  let n_ref: &usize = &100;
  inc_ref(n_ref);
  println!("{}", n_ref);
}

fn inc_ref(n_inc_ref: &usize) -> usize {
  n_inc_ref + 1
}
```

If we consider Rust's goal of safety it makes sense that a pass-by-value parameter scheme is used as it protects the original data used by the caller of the function.

Exception Handling

There are a variety of exception handlers with different purposes in Rust. The examples used below are found in “Rust by Example” from the Rust documentation except when noted otherwise.

panic

panic is the simplest error handler. It will print an error message and usually exit the program.

```
fn drink(beverage: &str) {
    // You shouldn't drink too much sugary beverages.
    if beverage == "lemonade" { panic!("AAAaaaaa!!!!"); }

    println!("Some refreshing {} is all I need.", beverage);
}

fn main() {
    drink("water");
    drink("lemonade");
}
```

Option and unwrap

Option and **unwrap** allow for the exception handling of multiple cases and None. Option<T> is a predefined enum where T is a type defined by the user, Option<T> has two available options. Some(T) is used when an element of type T is found and None means no element was found. Here is a basic example from “Learning Rust”.

```
fn get_an_optional_value() -> Option<&str> {

    //if the optional value is not empty
    return Some("Some value");

    //else
    None
```

```
}
```

Then, using **match,** a user can explicitly handle different cases.

unwrap() is an associated function which simply returns the element if it exists, and will **panic** if given a None type.

```
// The commoner has seen it all, and can handle any gift well.
// All gifts are handled explicitly using `match`.
fn give_commoner(gift: Option<&str>) {
    // Specify a course of action for each case.
    match gift {
        Some("snake") => println!("Yuck! I'm putting this snake back
in the forest."),
        Some(inner)   => println!("{}? How nice.", inner),
        None          => println!("No gift? Oh well."),
    }
}

// Our sheltered royal will `panic` at the sight of snakes.
// All gifts are handled implicitly using `unwrap`.
fn give_royal(gift: Option<&str>) {
    // `unwrap` returns a `panic` when it receives a `None`.
    let inside = gift.unwrap();
    if inside == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", inside);
}

fn main() {
    let food  = Some("cabbage");
    let snake = Some("snake");
    let void  = None;

    give_commoner(food);
    give_commoner(snake);
    give_commoner(void);

    let bird = Some("robin");
    let nothing = None;
```

```
    give_royal(bird);
    give_royal(nothing);
}
```

Instead of using **match** statements with **Option**, users can also use the **?** operator. When **?** is used with an **Option**, then it will return the value if the correct type is given, otherwise it will terminate the current function and return None. For example:

```
fn next_birthday(current_age: Option<u8>) -> Option<String> {
     // If `current_age` is `None`, this returns `None`.
     // If `current_age` is `Some`, the inner `u8` gets assigned to
`next_age`
    let next_age: u8 = current_age?;
    Some(format!("Next year I will be {}", next_age))
}
```

Result

Result is very similar to **Option** but provides more options for explicitly handling cases. Just like **Option**, Result<T, E> specifies the types which are expected and unexpected. Ok(T) is used when an element of type T is found while Err(E) is used when an element of type E is found. **unpack()** can also be used with **Result**, which will either return the element T or **panic**. **?** can also be used and will either return the element T when the **Result** is of type T or return None when **Result** is of type E. Below is a basic example of **Result** from "Learning Rust":

```
 fn function_with_error() -> Result<u64, String> {

    //if error happens
    return Err("The error message".to_string());

    // else, return valid output
    Ok(255)
}
```

Summary and Use Cases

panic is useful for quick testing and for providing guidance for unrecoverable errors.

Option is best for when a value is optional or when None is not an error condition as it lets you specify how those cases are handled.

Result is best for recoverable errors as it allows users to specify valid and invalid inputs, which can then be dealt with by the caller accordingly.

Data Abstraction

In Rust, **generics** are used to abstractly define functions and structs. Generics allow for easy duplication of a type's structure, should the programmer want to make multiple more-specific types that have the same properties and relationships to other generic-based types.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Here, the generic struct Point has been defined by replacing the data type, between the two brackets < >, with T. New Point objects can be instantiated with specified types of x and y.

Rust also uses **traits**, which are used to define methods that can be called on a type with that trait. This is similar to an interface in Java.

```
pub trait Summary {
    fn summarize(&self) -> String;
```

```
}
```

Here, the trait Summary is defined, and may now be implemented for specific types. All types for which the trait is implemented will have, as defined above, a method called “summarize” that takes the object as its parameter (`&self`) and returns a value of type `String`.

Subroutines

In Rust, subroutines take the form of **functions**, which are defined with the keyword `fn`. (Note: they are called functions, but do not necessarily take parameters or return a value.) To denote that a function should take a parameter, in this case a 32-bit integer is being taken and assigned to the variable x, the following syntax is used:

```
fn example_function(x: i32) {
      //function body here
}
```

As there is no default type for parameters passed into a function, each parameter must have its type declared. For functions with multiple parameters, the following syntax is used:

```
fn example_function(x: i32, y: String, z: f32) {
      //function body here
}
```

Rust does not support variable numbers of arguments in functions like languages such as Java, Go, or Python. Optional arguments can be accepted as parameters, however:

```
fn optional_args(x: Option<boolean>) -> Option<boolean> {
      match x {
            None => None,
            Some(b) => Some(!b),
      }
}
```

To denote that a function should return a value, in this case a 64-bit float, the following syntax is used:

```
fn example_function() -> f64 {
      //function body here
}
```

In Rust, functions defined as having return values, as above, automatically return the value of the final expression in the body of the function. Alternatively, the keyword `return` can be used to explicitly define a value to return. Any type can be returned from a function, including user-defined types, primitive types, composite types, as well as another function (including closures, which are essentially lambda functions). Rust cannot return multiple values from a single function, though. The best way this can be approximated is by using a tuple:

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

References

https://www.rust-lang.org/
https://doc.rust-lang.org/book/
https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/#:~:text=One%20of%20the%20biggest%20benefits,and%20can%20be%20cleaned%20up.
https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html
https://doc.rust-lang.org/book/ch03-02-data-types.html
https://doc.rust-lang.org/rust-by-example/
https://blog.ryanlevick.com/posts/rust-pass-value-or-reference/
https://doc.rust-lang.org/rust-by-example/error.html